IBM Engineering and Scientific Subroutine Library
for Linux on POWER

Version 6.1

ESSL Guide and Reference

IBM

Note

Before using this information and the product it supports, read the information in "Notices" on page 1249 .

This edition applies to Version 6 Release 1 of the IBM® Engineering and Scientific Subroutine Library (ESSL) for Linux on POWER® licensed program (product number 5765-L61) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

List of Tables

About this information

This document provides an overview of ESSL and guidance information for designing, coding, and processing your program, as well as migrating existing programs, diagnosing problems, and reference information for coding each ESSL calling sequence.

This documentation is written for a wide class of ESSL users: scientists, mathematicians, engineers, statisticians, computer scientists, and system programmers. It assumes a basic knowledge of mathematics in the areas of ESSL computation. This document also assumes that users are familiar with Fortran, C, and C++ programming.

Related information

The related Web sites listed below may be useful to you when using ESSL.

Product	Web site URL
Linux	For general information and documentation on Linux: `http://www.tldp.org/` For information about the standard Linux installation procedure using the RPM Package Manager (RPM): `http://www.rpm.org/` For information about IBM-related offerings for Linux: `http://www.ibm.com/linux/`
C and C++ XL Fortran	`http://www.ibm.com/support/knowledgecenter/`, under 'Rational' on the left hand pane.
NVIDIA	`http://www.nvidia.com` For information about CUDA, see: `http://developer.nvidia.com/cuda-toolkit` For the CUDA Toolkit Documentation site, see: `http://docs.nvidia.com/cuda/#axzz3VafCSAvr`

Using bibliography references

Special references are made throughout this documentation to mathematical background publications and software libraries, available through IBM, publishers, or other companies. All of these are described in detail in the bibliography. A reference to one of these is made by using a bracketed number. The number refers to the item listed under that number in the bibliography. For example, reference [1] cites the first item listed in the bibliography.

Interpreting the subroutine names with a prefix underscore

A name specified with an underscore (_) prefix, such as _GEMUL, refers to all the versions of the subroutine with that name. To get the entire list of subroutines that name refers to, substitute the first letter for each version of the subroutine. For example, _GEMUL, refers to all versions of the matrix multiplication subroutine: SGEMUL, DGEMUL, CGEMUL, and ZGEMUL. You do **not** use the underscore in coding the names of the ESSL subroutines in your program. You code a complete name, such as SGEMUL. For details about these names, see "The Variety of Mathematical Functions" on page 2.

IBM Request for Enhancement (RFE) Community

The IBM Requests for Enhancements (RFEs) Community provides an opportunity to collaborate directly with the IBM product development teams and other product users on RFEs.

You can submit ESSL RFEs at the Servers and Systems Software RFE Community:

```
https://www.ibm.com/developerworks/rfe/?BRAND_ID=352
```

Special Terms

Standard data processing and mathematical terms are used in this documentation. Terminology is generally consistent with that used for Fortran. See the Glossary for definitions of terms used.

Short and Long Precision

Because ESSL can be used with more than one programming language, the terms **short precision** and **long precision** are used in place of the Fortran terms **single precision** and **double precision**.

Subroutines and Subprograms

An ESSL **subroutine** is a named sequence of instructions within the ESSL product library whose execution is invoked by a call. A subroutine can be called in one or more user programs and at one or more times within each program. The ESSL subroutines are referred to as **subprograms** in the area of linear algebra subprograms. The term subprograms is used because it is consistent with the BLAS. Many of the linear algebra subprograms correspond to the BLAS; these are listed in Appendix A, "Basic Linear Algebra Subprograms (BLAS) and C BLAS (CBLAS)," on page 1225.

Abbreviated Names

The abbreviated names used are defined below.

Table 1: Abbreviated names.

Short Name	Full Name
BLAS	Basic Linear Algebra Subprograms (see Appendix A, "Basic Linear Algebra Subprograms (BLAS) and C BLAS (CBLAS)," on page 1225)
CBLAS	C interface to the BLAS (see Appendix A, "Basic Linear Algebra Subprograms (BLAS) and C BLAS (CBLAS)," on page 1225)
CUDA	Parallel computing platform and programming model invented by NVIDIA
ESSL	IBM Engineering and Scientific Subroutine Library
FFTW	Fastest Fourier Transform in the West (see Appendix C, "FFTW Version 3.1.2 to ESSL Wrapper Libraries," on page 1239)
GPU	Graphics processing unit
HTML	Hypertext Markup Language

Table 1: Abbreviated names. (continued)

Short Name	Full Name
LAPACK	Linear Algebra Package (see Appendix B, “LAPACK and LAPACKE,” on page 1231)
LAPACKE	C Interface to LAPACK (see Appendix B, “LAPACK and LAPACKE,” on page 1231)
OpenMP	Open Multi-Processing
SL MATH	Subroutine Library—Mathematics
SMP	Symmetric Multi-Processing
SSP	Scientific Subroutine Package

Conventions and terminology used

Table 2 on page xxv describes the typographic conventions used.

Table 2: Summary of typographic conventions.

Typographic	Usage
Bold	**Bold** words or characters represent system elements that you must use literally, such as commands, flags, and path names.
Italic	• *Italic* words or characters represent variable values that you must supply. • *Italics* are also used for book titles and for general emphasis in text.
`Constant width`	Examples and information that the system displays appear in `constant width` typeface.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices. (In other words, it means “or.”)
< >	Angle brackets (less-than and greater-than) enclose the name of a key on the keyboard. For example, `<Enter>` refers to the key on your terminal or workstation that is labeled with the word Enter.
...	An ellipsis indicates that you can repeat the preceding item one or more times.
`<Ctrl-`*x*`>`	The notation `<Ctrl-`*x*`>` indicates a control character sequence. For example, `<Ctrl-c>` means that you hold down the control key while pressing `<c>`.
\	The continuation character is used in coding examples for formatting purposes.

Conventions that are consistent with traditional mathematical usage are followed.

Fonts

A variety of special fonts are used to distinguish between many mathematical and programming items. These are defined below.

Special Font	**Example**	**Description**
Italic with no subscripts	*m*, *inc1x*, *aux*, *iopt*	Calling sequence argument or mathematical variable
Italic with subscripts	x_1, a_{mn}, $x_{j1,j2}$	Element of a vector, matrix, or sequence
Bold italic lowercase	***x***, ***y***, ***z***	Vector or sequence
Bold italic uppercase	***A***, ***B***, ***C***	Matrix
Gothic uppercase	A, B, C, AGB `IM=ISMAX(4,X,2)`	Array Fortran statement

Special Notations and Conventions

This explains the special notations and conventions used to describe various types of data.

Scalar Data

Following are the special notations used in the examples for scalar data items. These notations are used to simplify the examples, and they do not imply usage of any precision. For a definition of scalar data in Fortran, C, and C++, see Chapter 4, “Coding programs,” on page 125.

Data Item	**Example**	**Description**
Character item	`'T'`	Character(s) in single quotation marks
Hexadecimal string	`X'97FA00C1'`	String of 4-bit hexadecimal characters
Logical item	`.TRUE. .FALSE.`	True or false logical value, as indicated
Integer data	`1`	Number with no decimal point
Real data	`1.6`	Number with a decimal point
Complex data	`(1.0,-2.9)`	Real part followed by the imaginary part
Continuation	$1.666\overline{6}$	Continue the last digit (1.6666666... and so forth)

Vectors

A vector is represented as a single row or column of subscripted elements enclosed in square brackets. The subscripts refer to the element positions within the vector:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} \qquad \begin{bmatrix} x_1 & x_2 & x_3 & \ldots & x_n \end{bmatrix}$$

For a definition of vector, see "Vectors" on page 71.

Matrices

A matrix is represented as a block of elements enclosed in square brackets. Subscripts refer to the row and column positions, respectively:

$$\begin{bmatrix} a_{11} & \cdot & \cdot & \cdot & a_{1n} \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ a_{m1} & \cdot & \cdot & \cdot & a_{mn} \end{bmatrix}$$

For a definition of matrix, see "Matrices" on page 77.

Sequences

Sequences are used in the areas of sorting, searching, Fourier transforms, convolutions, and correlations. For a definition of sequences, see "Sequences" on page 120.

One-Dimensional Sequences

A one-dimensional sequence is represented as a series of elements enclosed in parentheses. Subscripts refer to the element position within the sequence:

$(x_1, x_2, x_3, \ldots, x_n)$

Two-Dimensional Sequences

A two-dimensional sequence is represented as a series of columns of elements. (They are represented in the same way as a matrix without the square brackets.) Subscripts refer to the element positions within the first and second dimensions, respectively:

$$\begin{array}{cccc}
a_{11} & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} & & a_{2n} \\
\cdot & \cdot & & \cdot \\
\cdot & \cdot & & \cdot \\
\cdot & \cdot & & \cdot \\
a_{m1} & a_{m2} & \cdots & a_{mn}
\end{array}$$

Three-Dimensional Sequences

A three-dimensional sequence is represented as a series of blocks of elements. Subscripts refer to the elements positions within the first, second, and third dimensions, respectively:

$$\begin{array}{cccc}
a_{111} & a_{121} & \cdots & a_{1n1} \\
a_{211} & a_{221} & & a_{2n1} \\
\cdot & \cdot & & \cdot \\
\cdot & \cdot & & \cdot \\
\cdot & \cdot & & \cdot \\
a_{m11} & a_{m21} & \cdots & a_{mn1}
\end{array}
\quad
\begin{array}{cccc}
a_{112} & a_{122} & \cdots & a_{1n2} \\
a_{212} & a_{222} & & a_{2n2} \\
\cdot & \cdot & & \cdot \\
\cdot & \cdot & & \cdot \\
\cdot & \cdot & & \cdot \\
a_{m12} & a_{m22} & \cdots & a_{mn2}
\end{array}
\quad \cdots \quad
\begin{array}{cccc}
a_{11p} & a_{12p} & \cdots & a_{1np} \\
a_{21p} & a_{22p} & & a_{2np} \\
\cdot & \cdot & & \cdot \\
\cdot & \cdot & & \cdot \\
\cdot & \cdot & & \cdot \\
a_{m1p} & a_{m2p} & \cdots & a_{mnp}
\end{array}$$

Arrays

Arrays contain vectors, matrices, or sequences. For a definition of array, see "How Do You Set Up Your Arrays?" on page 47.

One-Dimensional Arrays

A one-dimensional array is represented as a single row of numeric elements enclosed in parentheses:

```
                    (1.0, 2.0, 3.0, 4.0, 5.0)
```

Elements not significant to the computation are usually not shown in the array. One dot appears for each element not shown. In the following array, five elements are significant to the computation, and two elements not used in the computation exist between each of the elements shown:

```
        (1.0, . , . ,2.0, . , . ,3.0, . , . ,4.0, . , . ,5.0)
```

This notation is used to show vector elements inside an array.

Two-Dimensional Arrays

A two-dimensional array is represented as a block of numeric elements enclosed in square brackets:

```
                    | 1.0  11.0  5.0  25.0 |
                    | 2.0  12.0  6.0  26.0 |
                    | 3.0  13.0  7.0  27.0 |
                    | 4.0  14.0  8.0  28.0 |
```

Elements not significant to the computation are usually not shown in the array. One dot appears for each element not shown. The following array contains three rows and two columns not used in the computation:

```
| .    .    .    .    .    . |
| .    .    .    .    .    . |
| .   1.0  2.0  5.0  4.0  . |
| .   2.0  3.0  6.0  3.0  . |
| .   3.0  4.0  7.0  2.0  . |
| .   4.0  5.0  8.0  1.0  . |
| .    .    .    .    .    . |
```

This notation is used to show matrix elements inside an array.

Three-Dimensional Arrays

A three-dimensional array is represented as a series of blocks of elements separated by ellipses. Each block appears like a two-dimensional array:

```
| 1.0  11.0  5.0  25.0 |   | 10.0  111.0  15.0  125.0 |       | 100.0  11.0  15.0  25.0 |
| 2.0  12.0  6.0  26.0 |   | 20.0  112.0  16.0  126.0 |  ...  | 200.0  12.0  16.0  26.0 |
| 3.0  13.0  7.0  27.0 |   | 30.0  113.0  17.0  127.0 |       | 300.0  13.0  17.0  27.0 |
| 4.0  14.0  8.0  28.0 |   | 40.0  114.0  18.0  128.0 |       | 400.0  14.0  18.0  28.0 |
```

Elements not significant to the computation are usually not shown in the array. One dot appears for each element not shown, just as for two-dimensional arrays.

Special Characters, Symbols, Expressions, and Abbreviations

The mathematical and programming notations used are consistent with traditional mathematical and programming usage. These conventions are explained below, along with special abbreviations that are associated with specific values.

Item	**Description**
Greek letters: α, σ, ω, Ω	Symbolic scalar values
$\|a\|$	The absolute value of a
$\boldsymbol{a} \bullet \boldsymbol{b}$	The dot product of $\boldsymbol{a}$ and $\boldsymbol{b}$
x_i	The i-th element of vector $\boldsymbol{x}$
c_{ij}	The element in matrix $\boldsymbol{C}$ at row i and column j
$x_1 \dots x_n$	Elements from x_1 to x_n
$i = 1, n$	i is assigned the values 1 to n
$\boldsymbol{y} \leftarrow \boldsymbol{x}$	Vector $\boldsymbol{y}$ is replaced by vector $\boldsymbol{x}$
$\boldsymbol{xy}$	Vector $\boldsymbol{x}$ times vector $\boldsymbol{y}$
$\boldsymbol{AX} \cong \boldsymbol{B}$	$\boldsymbol{AX}$ is congruent to $\boldsymbol{B}$
a^k	a raised to the k power
e^x	Exponential function of x
$\boldsymbol{A}^T$; $\boldsymbol{x}^T$	The transpose of matrix $\boldsymbol{A}$; the transpose of vector $\boldsymbol{x}$
$\bar{\boldsymbol{x}}$; $\overline{\boldsymbol{A}}$	The complex conjugate of vector $\boldsymbol{x}$; the complex conjugate of matrix $\boldsymbol{A}$

Item	Description
$\bar{x}_i$; $\bar{c}_{jk}$	The complex conjugate of the complex vector element x_i, where: $\text{if } x_i = (a_i, b_i),$ $\text{then } \bar{x}_i = (a_i, -b_i)$ The complex conjugate of the complex matrix element c_{jk}
$\boldsymbol{x}^H$; $\boldsymbol{A}^H$	The complex conjugate transpose of vector $\boldsymbol{x}$; the complex conjugate transpose of matrix $\boldsymbol{A}$
$\sum_{i=1}^{n} x_i$	The sum of elements x_1 to x_n
$\sqrt{a+b}$	The square root of $a+b$
$\int_a^b f(x)\,dx$	The integral from a to b of $f(x)\,dx$
$\lVert\boldsymbol{x}\rVert_2$	The Euclidean norm of vector $\boldsymbol{x}$, defined as: $\sqrt{\sum_{j=1}^{n} \lvert x_j\rvert^2}$
$\lVert\boldsymbol{A}\rVert_1$	The one norm of matrix $\boldsymbol{A}$, defined as: $\max\left\{\sum_{i=1}^{m} \lvert a_{ij}\rvert,\ 1 \le j \le n\right\}$
$\lVert\boldsymbol{A}\rVert_2$	The spectral norm of matrix $\boldsymbol{A}$, defined as: $\max\{\lVert\boldsymbol{Ax}\rVert_2 : \lVert\boldsymbol{x}\rVert_2 = 1\}$
$\lVert\boldsymbol{A}\rVert_F$	The Frobenius or Euclidean norm of matrix $\boldsymbol{A}$, defined as: $\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} \lvert a_{ij}\rvert^2}$
$\lVert\boldsymbol{A}\rVert_\infty$	The infinity norm of matrix $\boldsymbol{A}$, defined as: $\max\left\{\sum_{j=1}^{n} \lvert a_{ij}\rvert, 1 \le i \le m\right\}$

Item	Description
$\boldsymbol{A}^{-1}$	The inverse of matrix ***A***
$\boldsymbol{A}^{-T}$	The transpose of ***A*** inverse
|***A***|	The determinant of matrix ***A***
m by *n* matrix ***A***	Matrix ***A*** has *m* rows and *n* columns
sin *a*	The sine of *a*
cos *b*	The cosine of *b*
SIGN (*a*)	The sign of *a*; the result is either + or -
address {*a*}	The storage address of *a*
max(***x***)	The maximum element in vector ***x***
min(***x***)	The minimum element in vector ***x***
ceiling(*x*)	The smallest integer that is greater than or equal to *x*
floor(*x*)	The largest integer that is not greater than *x*
int(*x*)	The largest integer that is less than or equal to *x*
x mod(*m*)	*x* modulo *m*; the remainder when *x* is divided by *m*
∞	Infinity
π	Pi, 3.14159265...

How to Interpret the Subroutine Descriptions

This explains how to interpret the information in the subroutine descriptions.

Description

Each subroutine description begins with a brief explanation of what the subroutine does. When we combine the description of multiple versions of a subroutine, we give enough information to enable you to easily tell the differences among the subroutines. Differences usually occur in either the function performed or the data types required for each subroutine.

For subroutines with CBLAS and LAPACKE calling sequences, the Data Types table lists only the Fortran name. The data types used for the CBLAS and LAPACKE are the same as that used for Fortran.

Syntax

This section shows the syntax for subroutines as follows:

- Table 3 on page xxxii shows the syntax for the Fortran, C and C++ calling sequences. All subroutines will include this information.
- Table 4 on page xxxii shows the syntax for the CBLAS calling sequences, which will appear for BLAS subroutines in addition to the Fortran, C, and C++ calling sequences.
- Table 5 on page xxxii shows the syntax for the LAPACKE calling sequences, which will appear for LAPACK subroutines in addition to the Fortran, C, and C++ calling sequences.

Note:

- For information about the CBLAS calling sequence, see [“11” on page 1253].

- For a list of BLAS, see Appendix A, “Basic Linear Algebra Subprograms (BLAS) and C BLAS (CBLAS),” on page 1225.
- For information about the LAPACKE calling sequence see the following URL:

```
http://www.netlib.org/lapack/LAPACKE.html
```

Table 3: Syntax for all subroutines.

Language	**Syntax**
Fortran	CALL NAME-1 | NAME-2 | ... | NAME-n (*arg-1*, *arg-2*, ..., *arg-m*)
C and C++	name-1 | name-2 | ... | name-n (*arg-1*, *arg-2*, ..., *arg-m*);

Table 4: Syntax for CBLAS subroutines.

Language	**Syntax**
CBLAS	cblas_name-1 | cblas_name-2 | ... | cblas_name-n (*arg-1*, *arg-2*, ..., *arg-m*);

Table 5: Syntax for LAPACKE subroutines.

Language	**Syntax**
LAPACKE	LAPACKE_name-1 | LAPACKE_name-2 | ... | LAPACKE_name-n (*arg-1*,*arg-2*, ..., *arg-m*);

The syntax indicates:

- Each possible subroutine or subprogram name that you can code in the calling sequence. Each name is separated by the | (or) symbol. You specify only one of these names in your calling sequence. (You do not code the | in the calling sequence.)
- The arguments, listed in the order in which you code them in the calling sequence. You must code them all in your calling sequence.

  You can distinguish between input arguments and output arguments by looking at "On Entry" and "On Return", respectively. An argument used for both input and output is described in both "On Entry" and "On Return". In this case, the input value for the argument is overlaid with the output value.

  The names of the arguments give an indication of the type of data that you should specify for the argument; for example:

  – Names beginning with the letters *i* through *n*, such as *m, incx, iopt*, and *isign*, indicate that you specify integer data.

  – Names beginning with the letters *a* through *h* and *o* through *z*, such as *b*, *t*, *alpha*, *sigma*, and *omega*, indicate that you specify real or complex data.

  – Names beginning with *cblas_* indicate that you specify enumerated types. These are used only for CBLAS.

  Note:

  – If you code a CBLAS calling sequence, there are times when an argument description references a character argument. In that situation, you should translate the character argument to its equivalent CBLAS enumerated type, as shown in Table 6 on page xxxiii.

  – The CBLAS_ORDER argument has been renamed to CBLAS_LAYOUT. See “Migrating Programs from ESSL for Linux on Power Version 5 Release 4 to Version 5 Release 5” on page 179 for the required changes to your existing programs.

– Although argument CBLAS_ORDER is still supported, IBM recommends using equivalent argument CBLAS_LAYOUT instead.

Table 6: Translating character argument values to CBLAS enumerated types

Character argument	Character Argument Value	CBLAS Argument	Enumerated Type Value
trans, transa, transb	'N' 'T' 'C'	*cblas_trans, cblas_transa, cblas_transb*	CblasNoTrans CblasTrans CblasConjTrans
side	'L' 'R'	*cblas_side*	CblasLeft CblasRight
uplo	'U' 'L'	*cblas_uplo*	CblasUpper CblasLower
diag	'U' 'N'	*cblas_diag*	CblasUnit CblasNonUnit

On Entry

This lists the input arguments, which are the arguments you pass to the ESSL subroutine. Each argument description first gives the meaning of the argument, and then gives the form of data required for the argument.

The calling sequences for the Level 2 CBLAS and the Level 3 CBLAS include input arguments that are enumerated types defined in essl.h. Argument *cblas_layout* indicates whether the input and output matrices are stored in column-major order or row-major order. All other enumerated type arguments replace the character arguments found in the Fortran, C and C++ calling sequences (see Table 6 on page xxxiii). Unlike the C and C++ interfaces to ESSL, complex scalar arguments are passed by reference instead of being passed by value.

On Return

This lists the output arguments, which are the arguments passed back to your program from the ESSL subroutine. Each argument description first gives the meaning of the argument, and then gives the form of data passed back to your program for the argument.

Notes

The notes describe any programming considerations and restrictions that apply to the arguments or the data for the arguments.

Function

This is a functional, or mathematical, description of the function performed by this subroutine. **It explains what computation is performed, not the implementation.** It explains the variations in the computation depending on the input arguments. References are made, where appropriate, to mathematical background books listed in the bibliography. References appear as a number enclosed in square brackets, where the number refers to the item listed under that number in the bibliography. For example, reference [1] cites the first item listed.

Special Usage

These are unique ways you can use the subroutine in your application. In most cases, this does not address applications of the ESSL subroutines; however, in special situations where the functional capability of the subroutine can be extended by following certain rules for its use, these rules are described.

Error Conditions

These are all the ESSL run-time errors that can occur in the subroutine. They are organized under three headings; Computational Errors, Input-Argument Errors, and Resource Errors. The return code values resulting from these errors are also explained.

Examples

The examples show how you would call the subroutine from a Fortran program using 32-bit integers. If you are using 64-bit integers, you may need to use a larger workspace and therefore you may need to increase the size of *naux* and *lwork*. (See "Setting Up Auxiliary Storage When Dynamic Allocation Is Not Used" on page 51.)

The examples provided for each subroutine show a variety of uses of the subroutine. Except where it is important to show differences in use between the various versions of the subroutine, the simplest version of the subroutine is used in the examples. In most cases, this is the short-precision real version of the subroutine. Each example provides a description of the important features of the example, followed by the Fortran calling sequence, the input data, and the resulting output data.

What's new in ESSL Guide and Reference

Read about new or significantly changed information for Engineering and Scientific Subroutine Library (ESSL) Version 6.1.

How to see what's new or changed

To help you see where technical changes occur, look for the following images:

- The >| image to mark where new or changed information begins.
- The |< image to mark where new or changed information ends.

August 2018

The following information is a summary of the updates that are made to the ESSL Version 6.1 Guide and Reference documentation:

- ESSL Version 6.1 supports the following IBM Power Systems™:
  - IBM POWER9™ servers

    Notes:

    - The ESSL SMP CUDA Library is supported only on the IBM Power Systems AC922 (8335-GTC and 8335-GTW) servers that include Tesla V100 with NVLink GPUs.
    - Red Hat Enterprise Linux 7.5 for Power® Little Endian (LE) (POWER9)
  - IBM POWER8™ servers

    Notes:

    - The ESSL SMP CUDA Library is supported only on IBM Power System S822LC (8335-GTB) servers with NVIDIA P100 GPUs.
    - Red Hat Enterprise Linux 7.5 (Little Endian (LE) mode)
- Added information about the following error messages:

  - 2538-2401 and 2538-2402 error messages in the “Resource Error Messages(2400-2499)” on page 196 topic.
  - 2538-2701 error message in the “Miscellaneous Error Messages(2700-2799)” on page 196 topic.
- BLAS and CBLAS subroutines:
  - SAXPBY and DAXPBY (Scales Vector X and Vector Y by two Scalars, Adds them to one another, and Stores in the Vector Y)
- LAPACK and LAPACKE subroutines:
  - CLANSY and ZLANSY (Complex Symmetric Matrix Norm)
  - SSYEVR, DSYEVR, CHEEVR, and ZHEEVR (Eigenvalues, and Optionally, the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix)
  - SSYGVD, DSYGVD, CHEGVD, and ZHEGVD (Eigenvalues, and Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem Using a Divide-and-Conquer Algorithm)
  - SSYGV, DSYGV, CHEGV, and ZHEGV (Eigenvalues, and Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem)
  - “SLAMCH and DLAMCH (Determine machine parameters for floating-point arithmetic)” on page 1210
- LAPACK GPU support:
  - SGESV, DGESV, CGESV, and ZGESV (General Matrix Factorization and Multiple Right-Hand Side Solve)
  - SGETRF, DGETRF, CGETRF, and ZGETRF (General Matrix Factorization)
  - SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)
  - SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)
  - SPOTRF, DPOTRF, CPOTRF, and ZPOTRF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)
  - SPOTRS, DPOTRS, CPOTRS, and ZPOTRS (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)
  - SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)
  - SPPTRF, DPPTRF, CPPTRF, and ZPPTRF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)
  - SPPTRS, DPPTRS, CPPTRS, and ZPPTRS (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)
  - SGEQRF, DGEQRF, CGEQRF, and ZGEQRF (General Matrix QR Factorization)
  - SGELS, DGELS, CGELS, and ZGELS (Linear Least Squares Solution for a General Matrix)
- Fourier Transforms GPU support:
  - SCFTD and DCFTD (Multidimensional Complex Fourier Transform)
  - SRCFTD and DRCFTD (Multidimensional Real-to-Complex Fourier Transform)
  - SCRFTD and DCRFTD (Multidimensional Complex-to-Real Fourier Transform)

Chapter 1. ESSL concepts

IBM Engineering and Scientific Subroutine Library (ESSL) is a state-of-the-art collection of high-performance subroutines providing a wide range of mathematical functions for many different scientific and engineering applications. Its primary characteristics are performance, functional capability, and usability.

Overview of ESSL

ESSL is provided as run-time libraries that run on the servers and processors listed in "Hardware Products Supported by ESSL" on page 5.

ESSL can be used with Fortran, C, and C++ programs operating under the Linux operating system.

To order ESSL, specify the following program number:

ESSL for Linux
> 5765-L61 <

Performance and Functional Capability

The mathematical subroutines, in nine computational areas, are tuned for performance. The computational areas are:

- Linear Algebra Subprograms
- Matrix Operations
- Linear Algebraic Equations
- Eigensystem Analysis
- Fourier Transforms, Convolutions and Correlations, and Related Computations
- Sorting and Searching
- Interpolation
- Numerical Quadrature
- Random Number Generation

ESSL runs under the Linux operating system.

ESSL provides the following run-time libraries (described in detail in "What ESSL Library Do You Want to Use?" on page 33):

- ESSL Serial Libraries and ESSL SMP Libraries, which run in the following environments:
  - 32-bit integer, 64-bit pointer environment
  - 64-bit integer, 64-bit pointer environment
- ESSL SMP CUDA Library which runs in the following environment:
  - 32-bit integer, 64-bit pointer environment

Notes:

- For the 32-bit integer, 64-bit pointer environment, in accordance with the LP64 data model, all ESSL integer arguments remain 32 bits except for the *iusadr* argument for ERRSET.
- To avoid 32-bit integer overflow problems (for example, matrices of order n where N > 46340), use the ESSL 64-bit integer, 64-bit pointer environment libraries.

These libraries contain a VSX (SIMD) version of selected subroutines for use on VSX enabled processor-based servers. These ESSL libraries are described in detail in "What ESSL Library Do You Want to Use?" on page 33.

All these libraries are designed to provide high levels of performance for numerically intensive computing jobs. All versions provide mathematically equivalent results.

The ESSL subroutines can be called from application programs written in Fortran, C, and C++.

Usability

ESSL is designed for usability:

- It has an easy-to-use call interface.
- If your existing application programs use the Serial Libraries, you only need to re-link your program to take advantage of the increased performance of the SMP Libraries.
- It has informative error-handling capabilities, enabling you to calculate auxiliary storage sizes and transform lengths.
- Online documentation that can be displayed using a Hypertext Markup Language (HTML) document browser is available for use with ESSL.

The Variety of Mathematical Functions

ESSL includes several different types of mathematical functions.

Areas of Application

ESSL provides a variety of mathematical functions for many different types of scientific and engineering applications. Some of the industries using these applications are: Aerospace, Automotive, Electronics, Petroleum, Finance, Utilities, and Research. Examples of applications in these industries are:

- Structural Analysis
- Time Series Analysis
- Computational Chemistry
- Computational Techniques
- Fluid Dynamics Analysis
- Mathematical Analysis
- Seismic Analysis Dynamic
- Systems Simulation Reservoir Modeling
- Nuclear Engineering Quantitative Analysis
- Electronic Circuit Design

What ESSL Provides

ESSL provides run-time libraries that are designed to provide high levels of performance for numerically intensive computing jobs.

The subroutines provided in ESSL, summarized in Table 7 on page 3, fall into the following groups:

- Nine major areas of mathematical computation, providing the computations commonly used by the industry applications listed
- Utilities, performing general-purpose functions

To help you select the ESSL subroutines that fulfill your needs for performance, accuracy, storage, and so forth, see "Selecting an ESSL Subroutine" on page 33.

Table 7: Summary of ESSL Subroutines.

ESSL Area of Computation	**Integer Subroutines**	**Short-Precision Subroutines**	**Long-Precision Subroutines**
Linear Algebra Subprograms:			
Vector-scalar	0	>|44|<	>|44|<
Sparse vector-scalar	0	11	11
Matrix-vector	0	38	38
Sparse matrix-vector	0	0	3
Matrix Operations:			
Addition, subtraction, multiplications, triangular solves, rank-k updates, rank-2k updates, and matrix transposes	0	29	30
Linear Algebraic Equations:			
Dense linear algebraic equations	0	>|83|<	>|83|<
Banded linear algebraic equations	0	40	40
Sparse linear algebraic equations	0	0	11
Linear least squares	0	13	13
Eigensystem Analysis:			
Solutions to the algebraic eigensystem analysis problem and the generalized eigensystem analysis problem	0	>|26|<	>| 26|<
Signal Processing Computations:			
Fourier transforms	0	18	14
Convolutions and correlations	0	10	2
Related computations	0	6	6
Sorting and Searching:			
Sorting, sorting with index, and binary and sequential searching	5	5	5
Interpolation:			
Polynomial and cubic spline interpolation	0	4	4
Numerical Quadrature:			
Numerical quadrature on a set of points or on a function	0	6	6
Random Number Generation:			
Generating vectors of uniformly distributed and normally distributed random numbers	1	5	5
Utilities:			
General service operations	9	>|1|<	>|4|<
Total ESSL Subroutines	15	>|339|<	>|350|<

Accuracy of the Computations

ESSL provides accuracy comparable to libraries using equivalent algorithms with identical precision formats. Both short- and long-precision real versions of the subroutines are provided in most areas of ESSL. In some areas, short- and long-precision complex versions are also provided, and, occasionally, an integer version is provided. The data types operated on by the short-precision and long-precision versions of the subroutines are ANSI/IEEE 32-bit and 64-bit binary floating-point format. See the *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754–1985*, for more detail. (There are ESSL-specific rules that apply to the results of computations on workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 61.)

For more information on accuracy, see "Getting the Best Accuracy" on page 60.

High Performance of ESSL

The ESSL subroutines have been designed to provide high performance. (See references ["40" on page 1255], ["51" on page 1256], and ["52" on page 1256].)

Algorithms

To achieve high performance, the subroutines use state-of-the-art algorithms tailored to specific operational characteristics of the hardware, such as cache size, Translation Lookaside Buffer (TLB) size, and page size.

Most subroutines use the following techniques to optimize performance:

- Managing the cache and TLB efficiently so the hit ratios are maximized; that is, data is blocked so it stays in the cache or TLB for its computation.
- Accessing data stored contiguously—that is, using stride-1 computations.
- Exploiting the large number of available floating-point registers.
- Using algorithms that minimize paging.
- Structuring the ESSL subroutines so, where applicable, the compiled code fully utilizes the dual floating-point execution units. Because two Multiply-Add instructions can be executed each cycle, neglecting overhead, this allows four floating-point operations per cycle to be performed.
- Structuring the ESSL subroutines so, where applicable, the compiled code takes full advantage of the hardware data prefetching.

Obtaining High Performance

Obtaining high performance depends on the type of processor you are using.

Obtaining High Performance on SMP processors with NVIDIA GPUs

The ESSL SMP CUDA Library is designed to exploit the processing power of the NVIDIA GPUs and of the Power CPUs for a subset of the ESSL subroutines. For a list of the multithreaded GPU-enabled subroutines in the ESSL SMP CUDA Library, see Table 34 on page 39.

Obtaining High Performance on SMP Processors

The ESSL SMP Libraries and the ESSL SMP CUDA Library are designed to exploit the processing power and shared memory of the SMP processor. In addition, a subset of the ESSL SMP subroutines have been coded to take advantage of increased performance from multithreaded (parallel) programming techniques. For a list of the multithreaded subroutines in the ESSL SMP Libraries, see Table 34 on page 39.

Choosing the number of threads depends on the problem size, the specific subroutine being called, and the number of physical processors you are running on. To achieve optimal performance, experimentation is necessary.

>|For best performance, you must use the `OMP_NUM_THREADS` environment variable to specify the number of threads that you want to create and bind the threads by using the `OMP_PROC_BIND` or

OMP_PLACES environment variable. Your performance might be improved by setting the OMP_PLACES environment variable to active.|<

Obtaining High Performance on VSX-Enabled Processors

The ESSL Serial Libraries, the ESSL SMP Libraries, and the ESSL SMP CUDA Library are designed to exploit the processing power of VSX-enabled processors. For details about how to use it to achieve optimal performance, see "SIMD Algorithms on VSX-Enabled Processors" on page 34.

SMT Mode

Simultaneous multithreading (SMT) is a processor technology that allows multiple instruction streams (threads) to run concurrently on the same physical processor, improving overall throughput. To the operating system, each hardware thread is treated as an independent logical processor. Single-threaded (ST) execution mode is also supported.

To obtain best performance for the IBM POWER9 servers, most HPC applications that use matrix multiplication extensively must be run in SMT-4 mode with four threads per core on POWER9 SMT-4 servers and in SMT-8 mode with eight threads per core on POWER9 SMT-8 servers. For example, in most cases, the performance of double precision general matrix multiplication (DGEMM) obtained in SMT-4 mode that uses the ESSL Symmetric Multi-Processing (SMP) Library with four threads that are bound to each core is better than the performance obtained by using one thread that is bound to each core or by running the applications in ST mode or SMT-2 mode.

>|To obtain the best performance for the IBM POWER8 servers, most HPC applications that use matrix multiplication extensively should use one thread bound to each core.|<

Mathematical Techniques

All areas of ESSL use state-of-the-art mathematical techniques to achieve high performance. For example, the matrix-vector linear algebra subprograms operate on a higher-level data structure, matrix-vector rather than vector-scalar. As a result, they optimize performance directly for your program and indirectly through those ESSL subroutines using them.

The Fortran Language Interface to the Subroutines

The ESSL subroutines follow standard Fortran calling conventions and must run in the Fortran run-time environment. When ESSL subroutines are called from a program in a language other than Fortran, such as C or C++, the Fortran conventions must be used. This applies to all aspects of the interface, such as the linkage conventions and the data conventions. For example, array ordering must be consistent with Fortran array ordering techniques. Data and linkage conventions for each language are given in Chapter 4, "Coding programs," on page 125.

Software and hardware

Review hardware and software products you can use with ESSL and products that you can use for installing ESSL.

Hardware Products Supported by ESSL

ESSL for Linux on POWER is supported on IBM POWER8 and POWER9 processor-based servers.

Notes: >|

- The ESSL SMP CUDA Library is supported on the IBM Power System AC922 (8335-GTC and 8335-GTW) servers that include Tesla V100 with NVLink GPUs.
- The ESSL SMP CUDA Library is supported on IBM Power System S822LC (8335-GTB) servers with NVIDIA P100 GPUs.

|<

Operating Systems Supported by ESSL

>|ESSL is supported in the Linux operating system on the following little endian mode environments:

- Red Hat Enterprise Linux 7.5 for Power Little Endian (POWER9).
- Red Hat Enterprise Linux 7.5 (Little Endian (LE) mode)

|<

Software Products Required by ESSL

ESSL requires the software products listed in Software requirements for compiling and running.

To assist C and C++ users, an ESSL header file is provided. Use of this file is described in "C Programs" on page 142 and "C++ Programs" on page 155.

Software requirements

The following table lists the required software products for ESSL for Linux on POWER:

Table 8: Required Software Products for ESSL for Linux on POWER.

Required software		Supported levels
For Compiling	IBM XL Fortran for Linux	15.1.6 or 16.1.0 with the latest service
	IBM XL C/C++ for Linux	13.1.6 or 16.1.0 with the latest service
	gcc and g++	See Note 3
For Linking, Loading, or Running (See Note 1)	IBM XL Fortran Runtime Environment for Linux (See Note 2)	15.1.6 or 16.1.0 with the latest service (See Note 2)
	gcc and g++ 64-bit libraries	See Note 3
	CUDA Toolkit (See Note 4)	>|9.2|<

Notes:

[1] Additional software packages may be required for building applications. For details, consult the Linux and compiler documentation.

[2] The correct version of IBM XL Fortran Runtime Environment and Addons Library for Linux is automatically shipped with the compiler. You can download the latest version of IBM XL Fortran Runtime Environment and Addons Library for Linux from the Latest updates for supported IBM XL Fortran compilers website.

[3] Use the compilers and libraries provided with your Linux distribution. The ESSL SMP libraries require the XL OpenMP runtime. The gcc OpenMP runtime is not compatible with the XL OpenMP runtime. Therefore, the ESSL SMP libraries can only be used with other compilers if the program calling ESSL is a serial program (does not use OpenMP) because in this case only the XL OpenMP runtime is used.

>|[4] This product is required only to use the ESSL SMP CUDA library.|<

Software Products for Installing and Customizing ESSL

The ESSL licensed program is distributed on a CD. Different software products are required for installing and customizing ESSL.

The *ESSL for Linux Installation Guide* provides the detailed information you need to install ESSL on the Linux operating system.

Man pages

To display a specific man page, run the man command as follows: `man subroutine-name`.

The ESSL man pages are installed in the `/usr/share/man/man3` directory. The BLAS and LAPACK man pages that are provided by the NETLIB open source packages are installed in the `/usr/share/man/manl` directory. By default, the ESSL man pages are displayed rather than the BLAS or LAPACK man pages with the same names. If you want to access the BLAS or LAPACK man pages, you must set the `MANPATH` environment variable.

To display the man pages properly on the Linux operating system, the `LANG` environment variable must be set to either of the following values: `C` or `en_US.iso8859-15`.

List of ESSL subroutines

ESSL provides several different types of subroutines.

Appendix A, "Basic Linear Algebra Subprograms (BLAS) and C BLAS (CBLAS)," on page 1225 contains a list of Level 1, 2, and 3 Basic Linear Algebra Subprograms (BLAS) included in ESSL.

Appendix B, "LAPACK and LAPACKE," on page 1231 contains a list of Linear Algebra Package (LAPACK) subroutines included in ESSL.

Linear algebra subprograms

There are several types of linear algebra subprograms.

- Vector-scalar linear algebra subprograms ("Vector-Scalar Linear Algebra Subprograms" on page 7)
- Sparse vector-scalar linear algebra subprograms ("Sparse Vector-Scalar Linear Algebra Subprograms" on page 10)
- Matrix-vector linear algebra subprograms ("Matrix-Vector Linear Algebra Subprograms" on page 11)
- Sparse matrix-vector linear algebra subprograms ("Sparse Matrix-Vector Linear Algebra Subprograms" on page 13)

Note:

1. The term **subprograms** is used to be consistent with the Basic Linear Algebra Subprograms (BLAS), because many of these subprograms correspond to the BLAS.
2. Some of the linear algebra subprograms were designed in accordance with the Level 1 and Level 2 BLAS de facto standard. If these subprograms do not comply with the standard as approved, IBM will consider updating them to do so. If IBM updates these subprograms, the updates could require modifications of the calling application program.

Vector-Scalar Linear Algebra Subprograms

The vector-scalar linear algebra subprograms include a subset of the standard set of Level 1 BLAS. For details on the BLAS, see reference ["95" on page 1258]. The remainder of the vector-scalar linear algebra subprograms are commonly used computations provided for your applications. Both real and complex versions of the subprograms are provided.

Table 9: List of Vector-Scalar Linear Algebra Subprograms.

Short-Precision Subprogram	Long-Precision Subprogram	Descriptive Name and Location
ISAMAX†• ICAMAX†• cblas_isamax• cblas_icamax•	IDAMAX†• IZAMAX†• cblas_idamax• cblas_izamax•	"ISAMAX, IDAMAX, ICAMAX, and IZAMAX (Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude)" on page 204

Table 9: List of Vector-Scalar Linear Algebra Subprograms. (continued)

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
ISAMIN† cblas_isamin	IDAMIN† cblas_idamin	“ISAMIN and IDAMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Absolute Value)” on page 206
ISMAX† cblas_ismax	IDMAX† cblas_idmax	“ISMAX and IDMAX (Position of the First or Last Occurrence of the Vector Element Having the Maximum Value)” on page 209
ISMIN† cblas_ismin	IDMIN† cblas_idmin	“ISMIN and IDMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Value)” on page 211
SASUM†• SCASUM†• cblas_sasum• cblas_scasum•	DASUM†• DZASUM†• cblas_dasum• cblas_dzasum•	“SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector)” on page 214
>| SAXPBY• cblas_saxpby•	DAXPBY• cblas_daxpby•	“SAXPBY and DAXPBY (Scales Vector X and Vector Y by two Scalars, Adds them to one another, and Stores in the Vector Y)” on page 219|<
SAXPY• CAXPY• cblas_saxpy• cblas_caxpy•	DAXPY• ZAXPY• cblas_daxpy• cblas_zaxpy•	“SAXPY, DAXPY, CAXPY, and ZAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y)” on page 216
SCOPY• CCOPY• cblas_scopy• cblas_ccopy•	DCOPY• ZCOPY• cblas_dcopy• cblas_zcopy•	“SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)” on page 223
SDOT†• CDOTU†• CDOTC†• cblas_sdot• cblas_cdotu_sub• cblas_cdotc_sub•	DDOT†• ZDOTU†• ZDOTC†• cblas_ddot• cblas_zdotu_sub• cblas_zdotc_sub•	“SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDOTC (Dot Product of Two Vectors)” on page 225
SNAXPY	DNAXPY	“SNAXPY and DNAXPY (Compute SAXPY or DAXPY N Times)” on page 229
SNDOT	DNDOT	“SNDOT and DNDOT (Compute Special Dot Products N Times)” on page 233
SNRM2†• SCNRM2†• cblas_snrm2• cblas_scnrm2•	DNRM2†• DZNRM2†• cblas_dnrm2• cblas_dznrm2•	“SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow)” on page 238

Table 9: List of Vector-Scalar Linear Algebra Subprograms. (continued)

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
SNORM2† CNORM2†	DNORM2† ZNORM2†	"SNORM2, DNORM2, CNORM2, and ZNORM2 (Euclidean Length of a Vector with No Scaling of Input)" on page 241
SROTG• CROTG• cblas_srotg• cblas_crotg•	DROTG• ZROTG• cblas_drotg• cblas_zrotg•	"SROTG, DROTG, CROTG, and ZROTG (Construct a Givens Plane Rotation)" on page 244
SROT• CROT• CSROT• cblas_srot• cblas_crot• cblas_csrot•	DROT• ZROT• ZDROT• cblas_drot• cblas_zrot• cblas_zdrot•	"SROT, DROT, CROT, ZROT, CSROT, and ZDROT (Apply a Plane Rotation)" on page 250
SROTMG• cblas_srotmg•	DROTMG• cblas_drotmg•	"SROTMG and DROTMG (Construct a modified Givens Transformation)" on page 254
SROTM• cblas_srotm•	DROTM• cblas_drotm•	"SROTM and DROTM (Apply a modified Givens Transformation)" on page 256
SSCAL• CSCAL• CSSCAL• cblas_sscal• cblas_cscal• cblas_csscal•	DSCAL• ZSCAL• ZDSCAL• cblas_dscal• cblas_zscal• cblas_zdscal•	"SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, and ZDSCAL (Multiply a Vector X by a Scalar and Store in the Vector X)" on page 259
SSWAP• CSWAP• cblas_sswap• cblas_cswap•	DSWAP• ZSWAP• cblas_dswap• cblas_zswap•	"SSWAP, DSWAP, CSWAP, and ZSWAP (Interchange the Elements of Two Vectors)" on page 262
SVEA CVEA cblas_svea cblas_cvea	DVEA ZVEA cblas_dvea cblas_zvea	"SVEA, DVEA, CVEA, and ZVEA (Add a Vector X to a Vector Y and Store in a Vector Z)" on page 265
SVES CVES cblas_sves cblas_cves	DVES ZVES cblas_dves cblas_zves	"SVES, DVES, CVES, and ZVES (Subtract a Vector Y from a Vector X and Store in a Vector Z)" on page 269

Table 9: List of Vector-Scalar Linear Algebra Subprograms. (continued)

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
SVEM CVEM cblas_svem cblas_cvem	DVEM ZVEM cblas_dvem cblas_zvem	"SVEM, DVEM, CVEM, and ZVEM (Multiply a Vector X by a Vector Y and Store in a Vector Z)" on page 273
SYAX CYAX CSYAX cblas_syax cblas_cyax cblas_csyax	DYAX ZYAX ZDYAX cblas_dyax cblas_zyax cblas_zdyax	"SYAX, DYAX, CYAX, ZYAX, CSYAX, and ZDYAX (Multiply a Vector X by a Scalar and Store in a Vector Y)" on page 276
SZAXPY CZAXPY cblas_szaxpy cblas_czaxpy	DZAXPY ZZAXPY cblas_dzaxpy cblas_zzaxpy	"SZAXPY, DZAXPY, CZAXPY, and ZZAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in a Vector Z)" on page 280

† This subprogram is invoked as a function in a Fortran program.

• Level 1 BLAS

Sparse Vector-Scalar Linear Algebra Subprograms

The sparse vector-scalar linear algebra subprograms operate on sparse vectors; that is, only the nonzero elements of the vector are stored. These subprograms provide similar functions to the vector-scalar subprograms. These subprograms represent a subset of the sparse extensions to the Level 1 BLAS described in reference ["39" on page 1255]. Both real and complex versions of the subprograms are provided.

Table 10: List of Sparse Vector-Scalar Linear Algebra Subprograms.

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
SSCTR CSCTR	DSCTR ZSCTR	"SSCTR, DSCTR, CSCTR, ZSCTR (Scatter the Elements of a Sparse Vector X in Compressed-Vector Storage Mode into Specified Elements of a Sparse Vector Y in Full-Vector Storage Mode)" on page 283
SGTHR CGTHR	DGTHR ZGTHR	"SGTHR, DGTHR, CGTHR, and ZGTHR (Gather Specified Elements of a Sparse Vector Y in Full-Vector Storage Mode into a Sparse Vector X in Compressed-Vector Storage Mode)" on page 285
SGTHRZ CGTHRZ	DGTHRZ ZGTHRZ	"SGTHRZ, DGTHRZ, CGTHRZ, and ZGTHRZ (Gather Specified Elements of a Sparse Vector Y in Full-Vector Mode into a Sparse Vector X in Compressed-Vector Mode, and Zero the Same Specified Elements of Y)" on page 288
SAXPYI CAXPYI	DAXPYI ZAXPYI	"SAXPYI, DAXPYI, CAXPYI, and ZAXPYI (Multiply a Sparse Vector X in Compressed-Vector Storage Mode by a Scalar, Add to a Sparse Vector Y in Full-Vector Storage Mode, and Store in the Vector Y)" on page 290

Table 10: List of Sparse Vector-Scalar Linear Algebra Subprograms. (continued)

Short-Precision Subprogram	Long-Precision Subprogram	Descriptive Name and Location
SDOTI† CDOTCI† CDOTUI†	DDOTI† ZDOTCI† ZDOTUI†	"SDOTI, DDOTI, CDOTUI, ZDOTUI, CDOTCI, and ZDOTCI (Dot Product of a Sparse Vector X in Compressed-Vector Storage Mode and a Sparse Vector Y in Full-Vector Storage Mode)" on page 292
† This subprogram is invoked as a function in a Fortran program.		

Matrix-Vector Linear Algebra Subprograms

The matrix-vector linear algebra subprograms operate on a higher-level data structure - matrix-vector rather than vector-scalar - using optimized algorithms to improve performance. These subprograms include a subset of the standard set of Level 2 BLAS. For details on the Level 2 BLAS, see ["44" on page 1255] and ["45" on page 1255]. Both real and complex versions of the subprograms are provided.

Table 11: List of Matrix-Vector Linear Algebra Subprograms.

Short-Precision Subprogram	Long-Precision Subprogram	Descriptive Name and Location
SGEMV◄ CGEMV◄ SGEMX§ SGEMTX§ cblas_sgemv◄ cblas_cgemv◄	DGEMV◄ ZGEMV◄ DGEMX§ DGEMTX§ cblas_dgemv◄ cblas_zgemv◄	"SGEMV, DGEMV, CGEMV, ZGEMV, SGEMX, DGEMX, SGEMTX, and DGEMTX (Matrix-Vector Product for a General Matrix, Its Transpose, or Its Conjugate Transpose)" on page 295
SGER◄ CGERU◄ CGERC◄ cblas_sger◄ cblas_cgeru◄ cblas_cgerc◄	DGER◄ ZGERU◄ ZGERC◄ cblas_dger◄ cblas_zgeru◄ cblas_zgerc◄	"SGER, DGER, CGERU, ZGERU, CGERC, and ZGERC (Rank-One Update of a General Matrix)" on page 306
SSPMV◄ CHPMV◄ SSYMV◄ CHEMV◄ SSLMX§ cblas_sspmv◄ cblas_chpmv◄ cblas_ssymv◄ cblas_chemv◄	DSPMV◄ ZHPMV◄ DSYMV◄ ZHEMV◄ DSLMX§ cblas_dspmv◄ cblas_zhpmv◄ cblas_dsymv◄ cblas_zhemv◄	"SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)" on page 313

Table 11: List of Matrix-Vector Linear Algebra Subprograms. (continued)

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
SSPR◄ CHPR◄ SSYR◄ CHER◄ SSLR1§ cblas_sspr◄ cblas_chpr◄ cblas_ssyr◄ cblas_cher◄	DSPR◄ ZHPR◄ DSYR◄ ZHER◄ DSLR1§ cblas_dspr◄ cblas_zhpr◄ cblas_dsyr◄ cblas_zher◄	“SSPR, DSPR, CHPR, ZHPR, SSYR, DSYR, CHER, ZHER, SSLR1, and DSLR1 (Rank-One Update of a Real Symmetric or Complex Hermitian Matrix)” on page 321
SSPR2◄ CHPR2◄ SSYR2◄ CHER2◄ SSLR2§ cblas_sspr2◄ cblas_chpr2◄ cblas_ssyr2◄ cblas_cher2◄	DSPR2◄ ZHPR2◄ DSYR2◄ ZHER2◄ DSLR2§ cblas_dspr2◄ cblas_zhpr2◄ cblas_dsyr2◄ cblas_zher2◄	“SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, ZHER2, SSLR2, and DSLR2 (Rank-Two Update of a Real Symmetric or Complex Hermitian Matrix) ” on page 328
SGBMV◄ CGBMV◄ cblas_sgbmv◄ cblas_cgbmv◄	DGBMV◄ ZGBMV◄ cblas_dgbmv◄ cblas_zgbmv◄	“SGBMV, DGBMV, CGBMV, and ZGBMV (Matrix-Vector Product for a General Band Matrix, Its Transpose, or Its Conjugate Transpose)” on page 336
SSBMV◄ CHBMV◄ cblas_ssbmv◄ cblas_chbmv◄	DSBMV◄ ZHBMV◄ cblas_dsbmv◄ cblas_zhbmv◄	“SSBMV, DSBMV, CHBMV, and ZHBMV (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Band Matrix)” on page 343
STRMV◄ CTRMV◄ STPMV◄ CTPMV◄ cblas_strmv◄ cblas_ctrmv◄ cblas_stpmv◄ cblas_ctpmv◄	DTRMV◄ ZTRMV◄ DTPMV◄ ZTPMV◄ cblas_dtrmv◄ cblas_ztrmv◄ cblas_dtpmv◄ cblas_ztpmv◄	“STRMV, DTRMV, CTRMV, ZTRMV, STPMV, DTPMV, CTPMV, and ZTPMV (Matrix-Vector Product for a Triangular Matrix, Its Transpose, or Its Conjugate Transpose)” on page 347
STRSV◄ CTRSV◄ STPSV◄ CTPSV◄ cblas_strsv◄ cblas_ctrsv◄ cblas_stpsv◄ cblas_ctps◄	DTRSV◄ ZTRSV◄ DTPSV◄ ZTPSV◄ cblas_dtrsv◄ cblas_ztrsv◄ cblas_dtpsv◄ cblas_ztps◄	“STRSV, DTRSV, CTRSV, ZTRSV, STPSV, DTPSV, CTPSV, and ZTPSV (Solution of a Triangular System of Equations with a Single Right-Hand Side)” on page 354

Table 11: List of Matrix-Vector Linear Algebra Subprograms. (continued)

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
STBMV◂ CTBMV◂ cblas_stbmv◂ cblas_ctbmv◂	DTBMV◂ ZTBMV◂ cblas_dtbmv◂ cblas_ztbmv◂	"STBMV, DTBMV, CTBMV, and ZTBMV (Matrix-Vector Product for a Triangular Band Matrix, Its Transpose, or Its Conjugate Transpose)" on page 360
STBSV◂ CTBSV◂ cblas_stbsv◂ cblas_ctbsv◂	DTBSV◂ ZTBSV◂ cblas_dtbsv◂ cblas_ztbsv◂	"STBSV, DTBSV, CTBSV, and ZTBSV (Triangular Band Equation Solve)" on page 365
◂ Level 2 BLAS § This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs.		

Sparse Matrix-Vector Linear Algebra Subprograms

The sparse matrix-vector linear algebra subprograms operate on sparse matrices; that is, only the nonzero elements of the matrix are stored. These subprograms provide similar functions to the matrix-vector subprograms.

Table 12: List of Sparse Matrix-Vector Linear Algebra Subprograms.

Long-Precision Subprogram	**Descriptive Name and Location**
DSMMX	"DSMMX (Matrix-Vector Product for a Sparse Matrix in Compressed-Matrix Storage Mode)" on page 371
DSMTM	"DSMTM (Transpose a Sparse Matrix in Compressed-Matrix Storage Mode)" on page 378
DSDMX	"DSDMX (Matrix-Vector Product for a Sparse Matrix or Its Transpose in Compressed-Diagonal Storage Mode)" on page 374

Matrix operations subroutines

Some of the matrix operation subroutines were designed in accordance with the Level 3 BLAS de facto standard. If these subroutines do not comply with the standard as approved, IBM will consider updating them to do so. If IBM updates these subroutines, the updates could require modifications of the calling application program. For details on the Level 3 BLAS, see reference ["42" on page 1255]. The matrix operation subroutines also include the commonly used matrix operations: addition, subtraction, multiplication, and transposition.

Table 13: List of Matrix Operation Subroutines.

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SGEADD CGEADD	DGEADD ZGEADD	"SGEADD, DGEADD, CGEADD, and ZGEADD (Matrix Addition for General Matrices or Their Transposes)" on page 386

Table 13: List of Matrix Operation Subroutines. (continued)

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SGESUB CGESUB	DGESUB ZGESUB	"SGESUB, DGESUB, CGESUB, and ZGESUB (Matrix Subtraction for General Matrices or Their Transposes)" on page 392
SGEMUL CGEMUL	DGEMUL ZGEMUL DGEMLP§	"SGEMUL, DGEMUL, CGEMUL, and ZGEMUL (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes)" on page 398
SGEMMS CGEMMS	DGEMMS ZGEMMS	"SGEMMS, DGEMMS, CGEMMS, and ZGEMMS (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm)" on page 408
SGEMM♦GPU CGEMM♦GPU cblas_sgemm♦ cblas_cgemm♦	DGEMM♦GPU ZGEMM♦GPU cblas_dgemm♦ cblas_zgemm♦	"SGEMM, DGEMM, CGEMM, and ZGEMM (Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes)" on page 413
SSYMM♦GPU CSYMM♦GPU CHEMM♦GPU cblas_ssymm♦ cblas_csymm♦ cblas_chemm♦	DSYMM♦GPU ZSYMM♦GPU ZHEMM♦GPU cblas_dsymm♦ cblas_zsymm♦ cblas_zhemm♦	"SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM (Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian)" on page 422
STRMM♦GPU CTRMM♦GPU cblas_strmm♦ cblas_ctrmm♦	DTRMM♦GPU ZTRMM♦GPU cblas_dtrmm♦ cblas_ztrmm♦	"STRMM, DTRMM, CTRMM, and ZTRMM (Triangular Matrix-Matrix Product)" on page 430
STRSM♦GPU CTRSM♦GPU cblas_strsm♦ cblas_ctrsm♦	DTRSM♦GPU ZTRSM♦GPU cblas_dtrsm♦ cblas_ztrsm♦	"STRSM, DTRSM, CTRSM, and ZTRSM (Solution of Triangular Systems of Equations with Multiple Right-Hand Sides)" on page 437
SSYRK♦GPU CSYRK♦GPU CHERK♦GPU cblas_ssyrk♦ cblas_csyrk♦ cblas_cherk♦	DSYRK♦GPU ZSYRK♦GPU ZHERK♦GPU cblas_dsyrk♦ cblas_zsyrk♦ cblas_zherk♦	"SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK (Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)" on page 445
SSYR2K♦GPU CSYR2K♦GPU CHER2K♦GPU cblas_ssyr2k♦ cblas_csyr2k♦ cblas_cher2k♦	DSYR2K♦GPU ZSYR2K♦GPU ZHER2K♦GPU cblas_dsyr2k♦ cblas_zsyr2k♦ cblas_zher2k♦	"SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K (Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)" on page 451

Table 13: List of Matrix Operation Subroutines. (continued)

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SGETMI CGETMI CGECMI	DGETMI ZGETMI ZGECMI	"SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI and ZGECMI (General Matrix Transpose or Conjugate Transpose [In-Place])" on page 459
SGETMO CGETMO CGECMO	DGETMO ZGETMO ZGECMO	"SGETMO, DGETMO, CGETMO, ZGETMO, CGECMO, and ZGECMO (General Matrix Transpose or Conjugate Transpose [Out-of-Place])" on page 462

♦ Level 3 BLAS

§ This subroutine is provided only for migration from earlier release of ESSL and is not intended for use in new programs.

> |GPU This subroutine can offload computations to the GPUs when the ESSL SMP CUDA Library is used.| <

Linear algebraic equations subroutines

The linear algebraic equations consist of:

- "Dense Linear Algebraic Equation Subroutines" on page 15
- "Banded Linear Algebraic Equations" on page 20
- "Sparse Linear Algebraic Equations" on page 22
- "Linear Least Squares" on page 23

Note: Some of the linear algebraic equations were designed in accordance with the LAPACK de facto standard. If these subprograms do not comply with the standard as approved, IBM will consider updating them to do so. If IBM updates these subprograms, the updates could require modifications of the calling application program. For details on LAPACK, see ["9" on page 1253].

Dense Linear Algebraic Equation Subroutines

The dense linear algebraic equation subroutines provide solutions to linear systems of equations for both real and complex general matrices and their transposes, positive definite real symmetric and complex Hermitian matrices, indefinite real or complex symmetric or complex Hermitian matrices, and triangular matrices. Some of these subroutines correspond to the LAPACK routines described in reference ["9" on page 1253].

Table 14: List of LAPACK Dense Linear Algebraic Equation Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SGESV∆GPU CGESV∆GPU LAPACKE_sgesv∆ LAPACKE_cgesv∆	DGESV∆GPU ZGESV∆GPU LAPACKE_dgesv∆ LAPACKE_zgesv∆	"SGESV, DGESV, CGESV, ZGESV (General Matrix Factorization and Multiple Right-Hand Side Solve)" on page 480

Table 14: List of LAPACK Dense Linear Algebraic Equation Subroutines. (continued)

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SGETRF$^{\Delta GPU}$ CGETRF$^{\Delta GPU}$ LAPACKE_sgetrf$^{\Delta}$ LAPACKE_cgetrf$^{\Delta}$	DGETRF$^{\Delta GPU}$ ZGETRF$^{\Delta GPU}$ LAPACKE_dgetrf$^{\Delta}$ LAPACKE_zgetrf$^{\Delta}$	"SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)" on page 484
SGETRS$^{\Delta GPU}$ CGETRS$^{\Delta GPU}$ LAPACKE_sgetrs$^{\Delta}$ LAPACKE_cgetrs$^{\Delta}$	DGETRS$^{\Delta GPU}$ ZGETRS$^{\Delta GPU}$ LAPACKE_dgetrf$^{\Delta}$ LAPACKE_zgetrf$^{\Delta}$	"SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)" on page 488
SGECON$^{\Delta}$ CGECON$^{\Delta}$ LAPACKE_sgecon $^{\Delta}$ LAPACKE_cgecon $^{\Delta}$	DGECON$^{\Delta}$ ZGECON$^{\Delta}$ LAPACKE_dgecon $^{\Delta}$ LAPACKE_zgecon $^{\Delta}$	"SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)" on page 504
SGETRI$^{\Delta}$ CGETRI$^{\Delta}$ LAPACKE_sgetri$^{\Delta}$ LAPACKE_cgetri$^{\Delta}$	DGETRI$^{\Delta}$ ZGETRI$^{\Delta}$ LAPACKE_dgetri$^{\Delta}$ LAPACKE_zgetri$^{\Delta}$	"SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)" on page 511
SLANGE$^{\Delta}$ CLANGE$^{\Delta}$ LAPACKE_slange$^{\Delta}$ LAPACKE_clange$^{\Delta}$	DLANGE$^{\Delta}$ ZLANGE$^{\Delta}$ LAPACKE_dlange $^{\Delta}$ LAPACKE_zlange $^{\Delta}$	"SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)" on page 518
SPPSV$^{\Delta GPU}$ CPPSV$^{\Delta GPU}$ LAPACKE_sppsv$^{\Delta}$ LAPACKE_cppsv$^{\Delta}$	DPPSV$^{\Delta GPU}$ ZPPSV$^{\Delta GPU}$ LAPACKE_dppsv$^{\Delta}$ LAPACKE_zppsv$^{\Delta}$	"SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)" on page 521
SPOSV$^{\Delta GPU}$ CPOSV$^{\Delta GPU}$ LAPACKE_sposv$^{\Delta}$ LAPACKE_cposv$^{\Delta}$	DPOSV$^{\Delta GPU}$ ZPOSV$^{\Delta GPU}$ LAPACKE_dposv$^{\Delta}$ LAPACKE_zposv$^{\Delta}$	"SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)" on page 527
SPOTRF$^{\Delta GPU}$ CPOTRF$^{\Delta GPU}$ SPPTRF$^{\Delta GPU}$ CPPTRF$^{\Delta GPU}$ LAPACKE_spotrf$^{\Delta}$ LAPACKE_cpotrf$^{\Delta}$ LAPACKE_spptrf$^{\Delta}$ LAPACKE_cpptrf$^{\Delta}$	DPOTRF$^{\Delta GPU}$ ZPOTRF$^{\Delta GPU}$ DPPTRF$^{\Delta GPU}$ ZPPTRF$^{\Delta GPU}$ LAPACKE_dpotrf$^{\Delta}$ LAPACKE_zpotrf$^{\Delta}$ LAPACKE_dpptrf$^{\Delta}$ LAPACKE_zpptrf$^{\Delta}$	"SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)" on page 533

Table 14: List of LAPACK Dense Linear Algebraic Equation Subroutines. (continued)

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SPOTRS$^{\Delta GPU}$ CPOTRS$^{\Delta GPU}$ SPPTRS$^{\Delta GPU}$ CPPTRS$^{\Delta GPU}$ LAPACKE_spotrs$^{\Delta}$ LAPACKE_cpotrs$^{\Delta}$ LAPACKE_spptrs$^{\Delta}$ LAPACKE_cpptrs$^{\Delta}$	DPOTRS$^{\Delta GPU}$ ZPOTRS$^{\Delta GPU}$ DPPTRS$^{\Delta GPU}$ ZPPTRS$^{\Delta GPU}$ LAPACKE_dpotrs$^{\Delta}$ LAPACKE_zpotrs$^{\Delta}$ LAPACKE_dpptrs$^{\Delta}$ LAPACKE_zpptrs$^{\Delta}$	"SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS, CPPTRS, and ZPPTRS (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)" on page 545
SPOCON$^{\Delta}$ CPOCON$^{\Delta}$ SPPCON$^{\Delta}$ CPPCON$^{\Delta}$ LAPACKE_spocon $^{\Delta}$ LAPACKE_cpocon $^{\Delta}$ LAPACKE_sppcon $^{\Delta}$ LAPACKE_cppcon $^{\Delta}$	DPOCON$^{\Delta}$ ZPOCON$^{\Delta}$ DPPCON$^{\Delta}$ ZPPCON$^{\Delta}$ LAPACKE_dpocon $^{\Delta}$ LAPACKE_zpocon $^{\Delta}$ LAPACKE_dppcon $^{\Delta}$ LAPACKE_zppcon $^{\Delta}$	"SPOCON, DPOCON, CPOCON, ZPOCON, SPPCON, DPPCON, CPPCON, and ZPPCON (Estimate the Reciprocal of the Condition Number of a Positive Definite Real Symmetric or Complex Hermitian Matrix)" on page 555
SPOTRI$^{\Delta}$ CPOTRI$^{\Delta}$ SPPTRI$^{\Delta}$ CPPTRI$^{\Delta}$ LAPACKE_spotri$^{\Delta}$ LAPACKE_cpotri$^{\Delta}$ LAPACKE_spptri$^{\Delta}$ LAPACKE_cpptri$^{\Delta}$	DPOTRI$^{\Delta}$ ZPOTRI$^{\Delta}$ DPPTRI$^{\Delta}$ ZPPTRI$^{\Delta}$ LAPACKE_dpotri$^{\Delta}$ LAPACKE_zpotri$^{\Delta}$ LAPACKE_dpptri$^{\Delta}$ LAPACKE_zpptri$^{\Delta}$	"SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPPTRI, DPPTRI, CPPTRI, ZPPTRI, SPPICD, and DPPICD (Positive Definite Real Symmetric or Complex Hermitian Matrix Inverse, Condition Number Reciprocal, and Determinant)" on page 569
SLANSY$^{\Delta}$ >|CLANSY$^{\Delta}$|< CLANHE$^{\Delta}$ SLANSP$^{\Delta}$ CLANHP$^{\Delta}$ LAPACKE_slansy$^{\Delta}$ >| LAPACKE_clansy$^{\Delta}$ |< LAPACKE_clanhe $^{\Delta}$ LAPACKE_slansp$^{\Delta}$ LAPACKE_clanhp $^{\Delta}$	DLANSY$^{\Delta}$ >|ZLANSY$^{\Delta}$|< ZLANHE$^{\Delta}$ DLANSP$^{\Delta}$ ZLANHP$^{\Delta}$ LAPACKE_dlansy$^{\Delta}$ >| LAPACKE_zlansy$^{\Delta}$ |< LAPACKE_zlanhe $^{\Delta}$ LAPACKE_dlansp $^{\Delta}$ LAPACKE_zlanhp $^{\Delta}$	"SLANSY, DLANSY, CLANSY, ZLANSY, CLANHE, ZLANHE, SLANSP, DLANSP, CLANHP, and ZLANHP (Real Symmetric or Complex Hermitian Matrix Norm)" on page 579

Table 14: List of LAPACK Dense Linear Algebraic Equation Subroutines. (continued)

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SSYSV$^{\Delta}$ CSYSV$^{\Delta}$ CHESV$^{\Delta}$ SSPSV$^{\Delta}$ CSPSV$^{\Delta}$ CHPSV$^{\Delta}$ LAPACKE_ssysv$^{\Delta}$ LAPACKE_csysv$^{\Delta}$ LAPACKE_chesv$^{\Delta}$ LAPACKE_sspsv$^{\Delta}$ LAPACKE_cspsv$^{\Delta}$ LAPACKE_chpsv$^{\Delta}$	DSYSV$^{\Delta}$ ZSYSV$^{\Delta}$ ZHESV$^{\Delta}$ DSPSV$^{\Delta}$ ZSPSV$^{\Delta}$ ZHPSV$^{\Delta}$ LAPACKE_dsysv$^{\Delta}$ LAPACKE_zsysv$^{\Delta}$ LAPACKE_zhesv$^{\Delta}$ LAPACKE_dspsv$^{\Delta}$ LAPACKE_zspsv$^{\Delta}$ LAPACKE_zhpsv$^{\Delta}$	"SSYSV, DSYSV, CSYSV, ZSYSV, CHESV, ZHESV, SSPSV, DSPSV, CSPSV, ZSPSV, CHPSV, and ZHPSV (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)" on page 584
SSYTRF$^{\Delta}$ CSYTRF$^{\Delta}$ CHETRF$^{\Delta}$ SSPTRF$^{\Delta}$ CSPTRF$^{\Delta}$ CHPTRF$^{\Delta}$ LAPACKE_ssytrf$^{\Delta}$ LAPACKE_csytrf$^{\Delta}$ LAPACKE_chetrf$^{\Delta}$ LAPACKE_ssptrf$^{\Delta}$ LAPACKE_csptrf$^{\Delta}$ LAPACKE_chptrf$^{\Delta}$	DSYTRF$^{\Delta}$ ZSYTRF$^{\Delta}$ ZHETRF$^{\Delta}$ DSPTRF$^{\Delta}$ ZSPTRF$^{\Delta}$ ZHPTRF$^{\Delta}$ LAPACKE_dsytrf$^{\Delta}$ LAPACKE_zsytrf$^{\Delta}$ LAPACKE_zhetrf$^{\Delta}$ LAPACKE_dsptrf$^{\Delta}$ LAPACKE_zsptrf$^{\Delta}$ LAPACKE_zhptrf$^{\Delta}$	"SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, and ZHPTRF (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization)" on page 593
SSYTRS$^{\Delta}$ CSYTRS$^{\Delta}$ CHETRS$^{\Delta}$ SSPTRS$^{\Delta}$ CSPTRS$^{\Delta}$ CHPTRS$^{\Delta}$ LAPACKE_ssytrs$^{\Delta}$ LAPACKE_csytrs$^{\Delta}$ LAPACKE_chetrs$^{\Delta}$ LAPACKE_ssptrs$^{\Delta}$ LAPACKE_csptrs$^{\Delta}$ LAPACKE_chptrs$^{\Delta}$	DSYTRS$^{\Delta}$ ZSYTRS$^{\Delta}$ ZHETRS$^{\Delta}$ DSPTRS$^{\Delta}$ ZSPTRS$^{\Delta}$ ZHPTRS$^{\Delta}$ LAPACKE_dsytrs$^{\Delta}$ LAPACKE_zsytrs$^{\Delta}$ LAPACKE_zhetrs$^{\Delta}$ LAPACKE_dsptrs$^{\Delta}$ LAPACKE_zsptrs$^{\Delta}$ LAPACKE_zhptrs$^{\Delta}$	"SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, ZHETRS, SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, and ZHPTRS (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)" on page 601
STRTRI$^{\Delta}$ CTRTRI$^{\Delta}$ STPTRI$^{\Delta}$ CTPTRI$^{\Delta}$ LAPACKE_strtri$^{\Delta}$ LAPACKE_stptri$^{\Delta}$ LAPACKE_ctrtri$^{\Delta}$ LAPACKE_ctptri$^{\Delta}$	DTRTRI$^{\Delta}$ ZTRTRI$^{\Delta}$ DTPTRI$^{\Delta}$ ZTPTRI$^{\Delta}$ LAPACKE_dtrtri$^{\Delta}$ LAPACKE_dtptri$^{\Delta}$ LAPACKE_ztrtri$^{\Delta}$ LAPACKE_ztptri$^{\Delta}$	"STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)" on page 621

Table 14: *List of LAPACK Dense Linear Algebraic Equation Subroutines. (continued)*

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SLANTR^Δ CLANTR^Δ SLANTP^Δ CLANTP^Δ LAPACKE_slantr^Δ LAPACKE_clantr^Δ LAPACKE_slantp^Δ LAPACKE_clantp^Δ	DLANTR^Δ ZLANTR^Δ DLANTP^Δ ZLANTP^Δ LAPACKE_dlantr^Δ LAPACKE_zlantr^Δ LAPACKE_dlantp^Δ LAPACKE_zlantp^Δ	"SLANTR, DLANTR, CLANTR, ZLANTR, SLANTP, DLANTP, CLANTP, and ZLANTP (Trapezoidal or Triangular Matrix Norm)" on page 628

^Δ LAPACK

> |^GPU This subroutine can offload computations to the GPUs when the ESSL SMP CUDA Library is used.| <

Table 15: *List of Dense Linear Algebraic Equation Subroutines.*

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SGEF CGEF	DGEF ZGEF DGEFP§	"SGEF, DGEF, CGEF, and ZGEF (General Matrix Factorization)" on page 493
SGESM CGESM	DGESM ZGESM	"SGESM, DGESM, CGESM, and ZGESM (General Matrix, Its Transpose, or Its Conjugate Transpose Multiple Right-Hand Side Solve)" on page 499
SGES CGES	DGES ZGES	"SGES, DGES, CGES, and ZGES (General Matrix, Its Transpose, or Its Conjugate Transpose Solve)" on page 496
SGEFCD	DGEFCD	"SGEFCD and DGEFCD (General Matrix Factorization, Condition Number Reciprocal, and Determinant)" on page 508
SGEICD	DGEICD	"SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)" on page 511
SPOF CPOF SPPF	DPOF ZPOF DPPF DPPFP§	"SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)" on page 533
SPOSM CPOSM	DPOSM ZPOSM	"SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS, CPPTRS, and ZPPTRS (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)" on page 545
SPPS	DPPS	"SPPS and DPPS (Positive Definite Real Symmetric Matrix Solve)" on page 552
SPPFCD SPOFCD	DPPFCD DPOFCD	"SPPFCD, DPPFCD, SPOFCD, and DPOFCD (Positive Definite Real Symmetric Matrix Factorization, Condition Number Reciprocal, and Determinant)" on page 563

Table 15: List of Dense Linear Algebraic Equation Subroutines. (continued)

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SPPICD SPOICD	DPPICD DPOICD	“SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPPTRI, DPPTRI, CPPTRI, ZPPTRI, SPPICD, and DPPICD (Positive Definite Real Symmetric or Complex Hermitian Matrix Inverse, Condition Number Reciprocal, and Determinant)” on page 569
	DBSSV	“DBSSV (Symmetric Indefinite Matrix Factorization and Multiple Right-Hand Side Solve)” on page 607
	DBSTRF	“DBSTRF (Symmetric Indefinite Matrix Factorization)” on page 613
	DBSTRS	“DBSTRS (Symmetric Indefinite Matrix Multiple Right-Hand Side Solve)” on page 617
STRI§ STPI§	DTRI§ DTPI§	“STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)” on page 621
§ This subroutine is provided for migration from earlier releases of ESSL and is not intended for use in new programs. Documentation for this subroutine is no longer provided.		

Banded Linear Algebraic Equations

The banded linear algebraic equation subroutines provide solutions to linear systems of equations for:

- Real or complex general band matrices
- Positive definite real symmetric or complex Hermitian band matrices
- Real or complex general tridiagonal matrices
- Positive definite real symmetric or complex Hermitian tridiagonal matrices

Table 16: List of LAPACK Banded Linear Algebraic Equation Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SGBSV∆ CGBSV∆ LAPACKE_sgbsv∆ LAPACKE_cgbsv∆	DGBSV∆ ZGBSV∆ LAPACKE_dgbsv∆ LAPACKE_zgbsv∆	“SGBSV, DGBSV, CGBSV, and ZGBSV (General Band Matrix Factorization and Multiple Right-Hand Side Solve)” on page 635
SGBTRF∆ CGBTRF∆ LAPACKE_sgbtrf∆ LAPACKE_cgbtrf∆	DGBTRF∆ ZGBTRF∆ LAPACKE_dgbtrf∆ LAPACKE_zgbtrf∆	“SGBTRF, DGBTRF, CGBTRF and ZGBTRF (General Band Matrix Factorization)” on page 639
SGBTRS∆ CGBTRS∆ LAPACKE_sgbtrs∆ LAPACKE_cgbtrs∆	DGBTRS∆ ZGBTRS∆ LAPACKE_dgbtrs∆ LAPACKE_zgbtrs∆	“SGBTRS, DGBTRS, CGBTRS, and ZGBTRS (General Band Matrix Multiple Right-Hand Side Solve)” on page 643

Table 16: List of LAPACK Banded Linear Algebraic Equation Subroutines. (continued)

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SPBSV$^{\Delta}$ CPBSV$^{\Delta}$ LAPACKE_spbsv$^{\Delta}$ LAPACKE_cpbsv$^{\Delta}$	DPBSV$^{\Delta}$ ZPBSV$^{\Delta}$ LAPACKE_dpbsv$^{\Delta}$ LAPACKE_zpbsv$^{\Delta}$	"SPBSV, DPBSV, CPBSV, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization and Multiple Right-Hand Side Solve)" on page 651
SPBTRF$^{\Delta}$ CPBTRF$^{\Delta}$ LAPACKE_spbtrf$^{\Delta}$ LAPACKE_cpbtrf$^{\Delta}$	DPBTRF$^{\Delta}$ ZPBTRF$^{\Delta}$ LAPACKE_dpbtrf$^{\Delta}$ LAPACKE_zpbtrf$^{\Delta}$	"SPBTRF, DPBTRF, CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)" on page 656
SPBTRS$^{\Delta}$ CPBTRS$^{\Delta}$ LAPACKE_spbtrs$^{\Delta}$ LAPACKE_cpbtrs$^{\Delta}$	DPBTRS$^{\Delta}$ ZPBTRS$^{\Delta}$ LAPACKE_dpbtrf$^{\Delta}$ LAPACKE_zpbtrf$^{\Delta}$	"SPBTRS, DPBTRS, CPBTRS, and ZPBTRS (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Multiple Right-Hand Side Solve)" on page 661
SGTSV$^{\Delta}$ CGTSV$^{\Delta}$ LAPACKE_sgtsv$^{\Delta}$ LAPACKE_cgtsv$^{\Delta}$	DGTSV$^{\Delta}$ ZGTSV$^{\Delta}$ LAPACKE_sgtsv$^{\Delta}$ LAPACKE_cgtsv$^{\Delta}$	"SGTSV, DGTSV, CGTSV, and ZGTSV (General Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)" on page 666
SGTTRF$^{\Delta}$ CGTTRF$^{\Delta}$ LAPACKE_sgttrf$^{\Delta}$ LAPACKE_cgttrf$^{\Delta}$	DGTTRF$^{\Delta}$ ZGTTRF$^{\Delta}$ LAPACKE_dgttrf$^{\Delta}$ LAPACKE_zgttrf$^{\Delta}$	"SGTTRF, DGTTRF, CGTTRF, and ZGTTRF (General Tridiagonal Matrix Factorization)" on page 670
SGTTRS$^{\Delta}$ CGTTRS$^{\Delta}$ LAPACKE_sgttrs$^{\Delta}$ LAPACKE_cgttrs$^{\Delta}$	DGTTRS$^{\Delta}$ ZGTTRS$^{\Delta}$ LAPACKE_dgttrs$^{\Delta}$ LAPACKE_zgttrs$^{\Delta}$	"SGTTRS, DGTTRS, CGTTRS, and ZGTTRS (General Tridiagonal Matrix Multiple Right-Hand Side Solve)" on page 674
SPTSV$^{\Delta}$ CPTSV$^{\Delta}$ LAPACKE_sptsv$^{\Delta}$ LAPACKE_cptsv$^{\Delta}$	DPTSV$^{\Delta}$ ZPTSV$^{\Delta}$ LAPACKE_dptsv$^{\Delta}$ LAPACKE_zptsv$^{\Delta}$	"SPTSV, DPTSV, CPTSV, and ZPTSV (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)" on page 679
SPTTRF$^{\Delta}$ CPTTRF$^{\Delta}$ LAPACKE_spttrf$^{\Delta}$ LAPACKE_cpttrf$^{\Delta}$	DPTTRF$^{\Delta}$ ZPTTRF$^{\Delta}$ LAPACKE_dpttrf$^{\Delta}$ LAPACKE_zpttrf$^{\Delta}$	"SPTTRF, DPTTRF, CPTTRF, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)" on page 684

Table 16: List of LAPACK Banded Linear Algebraic Equation Subroutines. (continued)

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SPTTRS$^{\Delta}$ CPTTRS$^{\Delta}$ LAPACKE_spttrs$^{\Delta}$ LAPACKE_cpttrs$^{\Delta}$	DPTTRS$^{\Delta}$ ZPTTRS$^{\Delta}$ LAPACKE_dpttrs$^{\Delta}$ LAPACKE_zpttrs$^{\Delta}$	"SPTTRS, DPTTRS, CPTTRS, and ZPTTRS (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Multiple Right-Hand Solve)" on page 687
$^{\Delta}$ LAPACK		

Table 17: List of non-LAPACK Banded Linear Algebraic Equation Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SGBF§	DGBF§	"SGBF and DGBF (General Band Matrix Factorization)" on page 693
SGBS§	DGBS§	"SGBS and DGBS (General Band Matrix Solve)" on page 649
SPBF§ SPBCHF§	DPBF§ DPBCHF§	"SPBF, DPBF, SPBCHF, and DPBCHF (Positive Definite Symmetric Band Matrix Factorization)" on page 698
SPBS§ SPBCHS§	DPBS§ DPBCHS§	"SPBS, DPBS, SPBCHS, and DPBCHS (Positive Definite Symmetric Band Matrix Solve)" on page 701
SGTF§	DGTF§	"SGTF and DGTF (General Tridiagonal Matrix Factorization)" on page 704
SGTS§	DGTS§	"SGTS and DGTS (General Tridiagonal Matrix Solve)" on page 706
SGTNP CGTNP	DGTNP ZGTNP	"SGTNP, DGTNP, CGTNP, and ZGTNP (General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting)" on page 708
SGTNPF CGTNPF	DGTNPF ZGTNPF	"SGTNPF, DGTNPF, CGTNPF, and ZGTNPF (General Tridiagonal Matrix Factorization with No Pivoting)" on page 711
SGTNPS CGTNPS	DGTNPS ZGTNPS	"SGTNPS, DGTNPS, CGTNPS, and ZGTNPS (General Tridiagonal Matrix Solve with No Pivoting)" on page 714
SPTF§	DPTF§	"SPTF and DPTF (Positive Definite Symmetric Tridiagonal Matrix Factorization)" on page 716
SPTS§	DPTS§	"SPTS and DPTS (Positive Definite Symmetric Tridiagonal Matrix Solve)" on page 718
§ This subroutine is provided for migration from earlier releases of ESSL and is not intended for use in new programs.		

Sparse Linear Algebraic Equations

The sparse linear algebraic equation subroutines provide direct and iterative solutions to linear systems of equations both for general sparse matrices and their transposes and for sparse symmetric matrices.

Table 18: List of Sparse Linear Algebraic Equation Subroutines.

Long-Precision Subroutine	**Descriptive Name and Location**
DGSF	"DGSF (General Sparse Matrix Factorization Using Storage by Indices, Rows, or Columns)" on page 720
DGSS	"DGSS (General Sparse Matrix or Its Transpose Solve Using Storage by Indices, Rows, or Columns)" on page 725
DGKFS DGKFSP§	"DGKFS (General Sparse Matrix or Its Transpose Factorization, Determinant, and Solve Using Skyline Storage Mode)" on page 729
DSKFS DSKFSP§	"DSKFS (Symmetric Sparse Matrix Factorization, Determinant, and Solve Using Skyline Storage Mode)" on page 744
DSRIS	"DSRIS (Iterative Linear System Solver for a General or Symmetric Sparse Matrix Stored by Rows)" on page 760
DSMCG‡	"DSMCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Matrix Storage Mode)" on page 769
DSDCG	"DSDCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)" on page 776
DSMGCG‡	"DSMGCG (General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode)" on page 782
DSDGCG	"DSDGCG (General Sparse Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)" on page 788

§ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs. Documentation for this subroutine is no longer provided.

‡ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs. Use DSRIS instead.

Linear Least Squares

The linear least squares subroutines provide least squares solutions to linear systems of equations for general matrices using a QR factorization or a singular value decomposition. Some of these subroutines correspond to the LAPACK routines described in reference ["9" on page 1253].

Table 19: List of LAPACK Linear Least Squares Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SGESVD$^{\Delta}$ CGESVD$^{\Delta}$ SGESDD$^{\Delta}$ CGESDD$^{\Delta}$ LAPACKE_sgesvd $^{\Delta}$ LAPACKE_cgesvd $^{\Delta}$ LAPACKE_sgesdd $^{\Delta}$ LAPACKE_cgesdd $^{\Delta}$	DGESVD$^{\Delta}$ ZGESVD$^{\Delta}$ DGESDD$^{\Delta}$ ZGESDD$^{\Delta}$ LAPACKE_dgesvd $^{\Delta}$ LAPACKE_zgesvd $^{\Delta}$ LAPACKE_dgesdd $^{\Delta}$ LAPACKE_zgesdd $^{\Delta}$	"SGESVD, DGESVD, CGESVD, ZGESVD, SGESDD, DGESDD, CGESDD, and ZGESDD (Singular Value Decomposition for a General Matrix)" on page 794
SGEQRF$^{\Delta GPU}$ CGEQRF$^{\Delta GPU}$ LAPACKE_sgeqrf$^{\Delta}$ LAPACKE_cgeqrf$^{\Delta}$	DGEQRF$^{\Delta GPU}$ ZGEQRF$^{\Delta GPU}$ LAPACKE_dgeqrf$^{\Delta}$ LAPACKE_zgeqrf$^{\Delta}$	"SGEQRF, DGEQRF, CGEQRF, and ZGEQRF (General Matrix QR Factorization)" on page 807
SGELS$^{\Delta\ GPU}$ CGELS$^{\Delta\ GPU}$ LAPACKE_sgels$^{\Delta}$ LAPACKE_cgels$^{\Delta}$	DGELS$^{\Delta\ GPU}$ ZGELS$^{\Delta\ GPU}$ LAPACKE_dgels$^{\Delta}$ LAPACKE_zgels$^{\Delta}$	"SGELS, DGELS, CGELS, and ZGELS (Linear Least Squares Solution for a General Matrix)" on page 812
SGELSD$^{\Delta}$ CGELSD$^{\Delta}$ LAPACKE_sgelsd$^{\Delta}$ LAPACKE_cgelsd$^{\Delta}$	DGELSD$^{\Delta}$ ZGELSD$^{\Delta}$ LAPACKE_dgelsd $^{\Delta}$ LAPACKE_zgelsd$^{\Delta}$	"SGELSD, DGELSD, CGELSD, and ZGELSD (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)" on page 822

$^{\Delta}$ LAPACK

> |^{GPU} This subroutine can offload computations to the GPUs when the ESSL SMP CUDA Library is used.| <

Table 20: List of Non–LAPACK Linear Least Squares Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SGESVF§	DGESVF§	"SGESVF and DGESVF (Singular Value Decomposition for a General Matrix)" on page 830
SGESVS§	DGESVS§	"SGESVS and DGESVS (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)" on page 837
SGELLS§	DGELLS§	"SGELLS and DGELLS (Linear Least Squares Solution for a General Matrix with Column Pivoting)" on page 841

Table 20: List of Non–LAPACK Linear Least Squares Subroutines. (continued)

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
§ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs.		

Eigensystem analysis subroutines

The eigensystem analysis subroutines provide solutions to the algebraic eigensystem analysis problem and the generalized eigensystem analysis problem. These subroutines correspond to the LAPACK routines described in reference [“9” on page 1253].

Table 21: List of LAPACK Eigensystem Analysis Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SGEEVΔ CGEEVΔ SGEEVXΔ CGEEVXΔ LAPACKE_sgeevΔ LAPACKE_cgeevΔ LAPACKE_sgeevxΔ LAPACKE_cgeevxΔ	DGEEVΔ ZGEEVΔ DGEEVXΔ ZGEEVXΔ LAPACKE_dgeevΔ LAPACKE_zgeevΔ LAPACKE_dgeevxΔ LAPACKE_zgeevxΔ	“SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)” on page 850

Table 21: List of LAPACK Eigensystem Analysis Subroutines. (continued)

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SSYEV$^{\Delta}$ CHEEV$^{\Delta}$ SSPEVX$^{\Delta}$ CHPEVX$^{\Delta}$ SSYEVX$^{\Delta}$ CHEEVX$^{\Delta}$ >|SSYEVR$^{\Delta}$|< >|CHEEVR$^{\Delta}$|< LAPACKE_ssyev$^{\Delta}$ LAPACKE_cheev$^{\Delta}$ LAPACKE_sspevx Δ LAPACKE_chpevx Δ LAPACKE_ssyevx Δ LAPACKE_cheevx Δ >| LAPACKE_ssyevr$^{\Delta}$ |< >| LAPACKE_cheevr Δ|<	DSYEV$^{\Delta}$ ZHEEV$^{\Delta}$ DSPEVX$^{\Delta}$ ZHPEVX$^{\Delta}$ DSYEVX$^{\Delta}$ ZHEEVX$^{\Delta}$ >|DSYEVR$^{\Delta}$|< >|ZHEEVR$^{\Delta}$|< LAPACKE_dsyev$^{\Delta}$ LAPACKE_zheev$^{\Delta}$ LAPACKE_dspevx Δ LAPACKE_zhpevx Δ LAPACKE_dsyevx Δ LAPACKE_zheevx Δ >| LAPACKE_dsyevr$^{\Delta}$ |< >| LAPACKE_zheevr Δ|<	"SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, ZHEEVX, SSYEVR, DSYEVR, CHEEVR, and ZHEEVR (Eigenvalues, and Optionally, the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix) " on page 867
SSPEVD$^{\Delta}$ CHPEVD$^{\Delta}$ SSYEVD$^{\Delta}$ CHEEVD$^{\Delta}$ LAPACKE_sspevd Δ LAPACKE_chpevd Δ LAPACKE_ssyevd Δ LAPACKE_cheevd Δ	DSPEVD$^{\Delta}$ ZHPEVD$^{\Delta}$ DSYEVD$^{\Delta}$ ZHEEVD$^{\Delta}$ LAPACKE_dspevd Δ LAPACKE_zhpevd Δ LAPACKE_dsyevd Δ LAPACKE_zheevd Δ	"SSPEVD, DSPEVD, CHPEVD, ZHPEVD, SSYEVD, DSYEVD, CHEEVD, and ZHEEVD (Eigenvalues and, Optionally the Eigenvectors, of a Real Symmetric or Complex Hermitian Matrix Using a Divide-and-Conquer Algorithm) " on page 893
SGGEV$^{\Delta}$ CGGEV$^{\Delta}$ SGGEVX$^{\Delta}$ CGGEVX$^{\Delta}$ LAPACKE_sggev$^{\Delta}$ LAPACKE_cggev$^{\Delta}$ LAPACKE_sggevx Δ LAPACKE_cggevx Δ	DGGEV$^{\Delta}$ ZGGEV$^{\Delta}$ DGGEVX$^{\Delta}$ ZGGEVX$^{\Delta}$ LAPACKE_dggev$^{\Delta}$ LAPACKE_zggev$^{\Delta}$ LAPACKE_dggevx Δ LAPACKE_zggevx Δ	"SGGEV, DGGEV, CGGEV, ZGGEV, SGGEVX, DGGEVX, CGGEVX, and ZGGEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix Generalized Eigenproblem)" on page 905

Table 21: List of LAPACK Eigensystem Analysis Subroutines. (continued)

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
>| SSYGVD$^{\Delta}$ CHEGVD$^{\Delta}$ LAPACKE_ssygvd$^{\Delta}$ LAPACKE_chegvd$^{\Delta}$	DSYGVD$^{\Delta}$ ZHEGVD$^{\Delta}$ LAPACKE_dsygvd$^{\Delta}$ LAPACKE_zhegvd$^{\Delta}$	"SSYGVD, DSYGVD, CHEGVD, ZHEGVD (Eigenvalues, and Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem Using a Divide-and-Conquer Algorithm)" on page 922|<
>|SSYGV$^{\Delta}$|< >|CHEGV$^{\Delta}$|< SSPGVX$^{\Delta}$ CHPGVX$^{\Delta}$ SSYGVX$^{\Delta}$ CHEGVX$^{\Delta}$ >| LAPACKE_ssygv$^{\Delta}$ |< >| LAPACKE_chegv$^{\Delta}$ |< LAPACKE_sspgvx$^{\Delta}$ LAPACKE_chpgvx$^{\Delta}$ LAPACKE_ssygvx$^{\Delta}$ LAPACKE_chegvx$^{\Delta}$	>|DSYGV$^{\Delta}$|< >|ZHEGV$^{\Delta}$|< DSPGVX$^{\Delta}$ ZHPGVX$^{\Delta}$ DSYGVX$^{\Delta}$ ZHEGVX$^{\Delta}$ >| LAPACKE_dsygv$^{\Delta}$ |< >| LAPACKE_zhegv$^{\Delta}$ |< LAPACKE_dspgvx$^{\Delta}$ LAPACKE_zhpgvx$^{\Delta}$ LAPACKE_dsygvx$^{\Delta}$ LAPACKE_zhegvx$^{\Delta}$	"SSYGV, DSYGV, CHEGV, ZHEGV, SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX (Eigenvalues, and Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem)" on page 931
$^{\Delta}$LAPACK		

Fourier Transforms, Convolutions and Correlations, and Related Computations

This signal processing area provides:

- Fourier transform subroutines
- Convolution and correlation subroutines
- Related-computation subroutines

Fourier Transforms

The Fourier transform subroutines perform mixed-radix transforms in one, two, and three dimensions.

Table 22: List of Fourier Transform Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SCFTD$^{\text{GPU}}$	DCFTD$^{\text{GPU}}$	"SCFTD and DCFTD (Multidimensional Complex Fourier Transform)" on page 962

Table 22: List of Fourier Transform Subroutines. (continued)

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SRCFTD^{GPU}	DRCFTD^{GPU}	"SRCFTD and DRCFTD (Multidimensional Real-to-Complex Fourier Transform)" on page 970
SCRFTD^{GPU}	DCRFTD^{GPU}	"SCRFTD and DCRFTD (Multidimensional Complex-to-Real Fourier Transform)" on page 978
SCFT§ SCFTP^{§,ND}	DCFT§	"SCFT and DCFT (Complex Fourier Transform)" on page 986
SRCFT§	DRCFT§	"SRCFT and DRCFT (Real-to-Complex Fourier Transform)" on page 993
SCRFT§	DCRFT§	"SCRFT and DCRFT (Complex-to-Real Fourier Transform)" on page 1000
SCOSF SCOSFT^{§,ND}	DCOSF	"SCOSF and DCOSF (Cosine Transform)" on page 1008
SSINF	DSINF	"SSINF and DSINF (Sine Transform)" on page 1015
SCFT2§ SCFT2P^{§,ND}	DCFT2§	"SCFT2 and DCFT2 (Complex Fourier Transform in Two Dimensions)" on page 1021
SRCFT2§	DRCFT2§	"SRCFT2 and DRCFT2 (Real-to-Complex Fourier Transform in Two Dimensions)" on page 1028
SCRFT2§	DCRFT2§	"SCRFT2 and DCRFT2 (Complex-to-Real Fourier Transform in Two Dimensions)" on page 1034
SCFT3§ SCFT3P^{§,ND}	DCFT3§	"SCFT3 and DCFT3 (Complex Fourier Transform in Three Dimensions)" on page 1041
SRCFT3§	DRCFT3§	"SRCFT3 and DRCFT3 (Real-to-Complex Fourier Transform in Three Dimensions)" on page 1047
SCRFT3§	DCRFT3§	"SCRFT3 and DCRFT3 (Complex-to-Real Fourier Transform in Three Dimensions)" on page 1053

§ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs.

ND Documentation for this subroutine is no longer provided.

> |GPU This subroutine can offload computations to the GPUs when the ESSL SMP CUDA Library is used.| <

Convolutions and Correlations

The convolution and correlation subroutines provide the choice of using Fourier methods or direct methods. The Fourier-method subroutines contain a high-performance mixed-radix capability. There are also several direct-method subroutines that provide decimated output.

Table 23: List of Convolution and Correlation Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SCON§ SCOR§		"SCON and SCOR (Convolution or Correlation of One Sequence with One or More Sequences)" on page 1060
SCOND SCORD		"SCOND and SCORD (Convolution or Correlation of One Sequence with Another Sequence Using a Direct Method)" on page 1065
SCONF SCORF		"SCONF and SCORF (Convolution or Correlation of One Sequence with One or More Sequences Using the Mixed-Radix Fourier Method)" on page 1070
SDCON SDCOR	DDCON DDCOR	"SDCON, DDCON, SDCOR, and DDCOR (Convolution or Correlation with Decimated Output Using a Direct Method)" on page 1079
SACOR§		"SACOR (Autocorrelation of One or More Sequences)" on page 1083
SACORF		"SACORF (Autocorrelation of One or More Sequences Using the Mixed-Radix Fourier Method)" on page 1086

§ These subroutines are provided only for migration from earlier releases of ESSL and are not intended for use in new programs.

Related Computations

The related-computation subroutines consist of a group of computations that can be used in general signal processing applications. They are similar to those provided on the IBM 3838 Array Processor; however, the ESSL subroutines generally solve a wider range of problems.

Table 24: List of Related-Computation Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SPOLY	DPOLY	"SPOLY and DPOLY (Polynomial Evaluation)" on page 1091
SIZC	DIZC	"SIZC and DIZC (I-th Zero Crossing)" on page 1094
STREC	DTREC	"STREC and DTREC (Time-Varying Recursive Filter)" on page 1096
SQINT	DQINT	"SQINT and DQINT (Quadratic Interpolation)" on page 1099
SWLEV CWLEV	DWLEV ZWLEV	"SWLEV, DWLEV, CWLEV, and ZWLEV (Wiener-Levinson Filter Coefficients)" on page 1102

Sorting and searching subroutines

The sorting and searching subroutines operate on three types of data: integer, short-precision real, and long-precision real. The sorting subroutines perform sorts with or without index designations. The searching subroutines perform either a binary or sequential search.

Table 25: List of Sorting and Searching Subroutines.

Integer Subroutine	Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
ISORT	SSORT	DSORT	"ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)" on page 1110
ISORTX	SSORTX	DSORTX	"ISORTX, SSORTX, and DSORTX (Sort the Elements of a Sequence and Note the Original Element Positions)" on page 1111
ISORTS	SSORTS	DSORTS	"ISORTS, SSORTS, and DSORTS (Sort the Elements of a Sequence Using a Stable Sort and Note the Original Element Positions)" on page 1114
IBSRCH	SBSRCH	DBSRCH	"IBSRCH, SBSRCH, and DBSRCH (Binary Search for Elements of a Sequence X in a Sorted Sequence Y)" on page 1117
ISSRCH	SSSRCH	DSSRCH	"ISSRCH, SSSRCH, and DSSRCH (Sequential Search for Elements of a Sequence X in the Sequence Y)" on page 1120

Interpolation subroutines

The interpolation subroutines provide the capabilities of doing polynomial interpolation, local polynomial interpolation, and both one- and two-dimensional cubic spline interpolation (Table 26 on page 30).

Table 26: List of Interpolation Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SPINT	DPINT	"SPINT and DPINT (Polynomial Interpolation)" on page 1126
STPINT	DTPINT	"STPINT and DTPINT (Local Polynomial Interpolation)" on page 1130
SCSINT	DCSINT	"SCSINT and DCSINT (Cubic Spline Interpolation)" on page 1133
SCSIN2	DCSIN2	"SCSIN2 and DCSIN2 (Two-Dimensional Cubic Spline Interpolation)" on page 1137

Numerical quadrature subroutines

The numerical quadrature subroutines provide Gaussian quadrature methods for integrating a tabulated function and a user-supplied function over a finite, semi-infinite, or infinite region of integration.

Table 27: List of Numerical Quadrature Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SPTNQ	DPTNQ	"SPTNQ and DPTNQ (Numerical Quadrature Performed on a Set of Points)" on page 1145
SGLNQ†	DGLNQ†	"SGLNQ and DGLNQ (Numerical Quadrature Performed on a Function Using Gauss-Legendre Quadrature)" on page 1147

Table 27: List of Numerical Quadrature Subroutines. (continued)

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SGLNQ2†	DGLNQ2†	"SGLNQ2 and DGLNQ2 (Numerical Quadrature Performed on a Function Over a Rectangle Using Two-Dimensional Gauss-Legendre Quadrature)" on page 1149
SGLGQ†	DGLGQ†	"SGLGQ and DGLGQ (Numerical Quadrature Performed on a Function Using Gauss-Laguerre Quadrature)" on page 1155
SGRAQ†	DGRAQ†	"SGRAQ and DGRAQ (Numerical Quadrature Performed on a Function Using Gauss-Rational Quadrature) " on page 1158
SGHMQ†	DGHMQ†	"SGHMQ and DGHMQ (Numerical Quadrature Performed on a Function Using Gauss-Hermite Quadrature)" on page 1161
† This subprogram is invoked as a function in a Fortran program.		

Random number generation subroutines

Random number generation subroutines generate uniformly distributed random numbers or normally distributed random numbers using one of the following algorithms:

- SIMD-oriented Mersenne Twister algorithm
- Multiplicative congruential methods
- Polar methods
- Tausworthe exclusive-or algorithm

Table 28: List of Random Number Generation Initialization Subroutines.

Subroutine	Descriptive Name and Location
INITRNG	"INITRNG (Initialize Random Number Generators)" on page 1166

Table 29: List of Random Number Generation Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SURNG	DURNG	"SURNG and DURNG (Generate a Vector of Uniformly Distributed Pseudo-Random Numbers)" on page 1170
SNRNG	DNRNG	"SNRNG and DNRNG (Generate a Vector of Normally Distributed Pseudo-Random numbers)" on page 1173
SURAND	DURAND	"SURAND and DURAND (Generate a Vector of Uniformly Distributed Random Numbers)" on page 1176
SNRAND	DNRAND	"SNRAND and DNRAND (Generate a Vector of Normally Distributed Random Numbers)" on page 1178
SURXOR§	DURXOR§	"SURXOR and DURXOR (Generate a Vector of Long Period Uniformly Distributed Random Numbers)" on page 1181
§ This subroutine is provided for migration from earlier releases of ESSL and is not intended for use in new programs.		

Utilities subroutines

The utility subroutines perform general service functions that support ESSL, rather than mathematical computations.

Table 30: List of Utility Subroutines.

Subroutine	Descriptive Name and Location
EINFO	"EINFO (ESSL Error Information-Handler Subroutine)" on page 1188
ERRSAV	"ERRSAV (ESSL ERRSAV Subroutine)" on page 1190
ERRSET	"ERRSET (ESSL ERRSET Subroutine)" on page 1191
ERRSTR	"ERRSTR (ESSL ERRSTR Subroutine)" on page 1192
IVSSET§	Set the Vector Section Size (VSS) for the ESSL/370 Scalar Library
IEVOPS§	Set the Extended Vector Operations Indicator for the ESSL/370 Scalar Library
IESSL	"IESSL (Determine the Level of ESSL Installed)" on page 1193
SETGPUS	"SETGPUS (Set the Number of GPUs and Identify Which GPUs ESSL Should Use)" on page 1194
STRIDE	"STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)" on page 1195
> SLAMCH and DLAMCH	"SLAMCH and DLAMCH (Determine machine parameters for floating-point arithmetic)" on page 1210 <
DSRSM	"DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)" on page 1212
DGKTRN	"DGKTRN (For a General Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode)" on page 1215
DSKTRN	"DSKTRN (For a Symmetric Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode)" on page 1219

§ This subroutine is provided for migration from earlier releases of ESSL and is not intended for use in new programs. Documentation for this subroutine is no longer provided.

Chapter 2. Planning for ESSL

Planning your ESSL program involves several tasks.

Selecting an ESSL Subroutine

Your choice of which ESSL subroutine to use is based mainly on the functional needs of your program. However, you have a choice of several variations of many of the subroutines. In addition, there are instances where certain subroutines cannot be used.

What ESSL Library Do You Want to Use?

ESSL provides serial and SMP libraries, as described here. For additional details about using these libraries, see Chapter 4, “Coding programs,” on page 125 and Chapter 5, “Processing Your Program,” on page 169.

Serial and SMP Libraries Provided by ESSL

ESSL provides the following serial library:

- ESSL Serial Libraries, which support the following environments:
  - 32-bit integer, 64-bit pointer environment
  - 64-bit integer, 64-bit pointer environment

These serial libraries provide thread-safe versions of the ESSL subroutines. You may choose to use these libraries if you decide to develop your own multithreaded programs that call the thread-safe ESSL subroutines.

ESSL also provides the following SMP libraries:

- ESSL SMP Libraries, which support the following environments:
  - 32-bit integer, 64-bit pointer environment
  - 64-bit integer, 64-bit pointer environment
- ESSL SMP CUDA Library, which supports the following environment:
  - >|32-bit integer, 64-bit pointer environment|<

  The ESSL SMP CUDA library provides the following options for a subset of ESSL subroutines:
  - Use one or more NVIDIA GPUs
  - Use one or more NVIDIA GPUs and Power CPUs

  The GPU enabled subroutines that the ESSL SMP CUDA Library contains are listed in “Using the ESSL SMP CUDA Library” on page 42.

These ESSL SMP libraries and ESSL SMP CUDA library provide thread-safe versions of the ESSL subroutines, and in addition, a subset of these subroutines are also multithreaded versions; that is, they support the shared memory parallel processing programming model.

The number of threads you choose to use depends on the problem size, the specific subroutine being called, and the number of physical processors you are running on. To achieve optimal performance, experimentation is necessary. The maximum number of threads supported by ESSL is 512.

You do not have to change your existing application programs that call ESSL to take advantage of the increased performance of using the SMP processors or the GPUs; you can simply re-link your existing application programs.

The multithreaded subroutines in the ESSL SMP Libraries are listed in “Multithreaded Subroutines Provided by ESSL” on page 38.

Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL

Some of the subroutines in the libraries provided by ESSL use SIMD algorithms, as explained in the following sections.

SIMD Algorithms on VSX-Enabled Processors

A subset of ESSL subroutines use SIMD algorithms that use the VSX unit on VSX enabled processors. These subroutines need to use the vector load and store instructions to effectively utilize the VSX unit. Alignment requirements for the SIMD algorithms are described in Table 31 on page 34 and Table 32 on page 35.

See Table 33 on page 37 for a list of the ESSL subroutines that automatically use SIMD algorithms when the appropriate alignment restrictions (as described in Table 31 on page 34 and Table 32 on page 35) are met.

Note: For Fourier Transform and Fourier Method Convolution and Correlation subroutines, if you choose to have ESSL calculate the size of auxiliary storage (see "Who Do You Want to Calculate the Size of Auxiliary Storage? You or ESSL?" on page 51), you must pass all array arguments with the same alignment as those passed during the initialization and computation calls. Because of this, it is recommended that you use the processor-independent formulas.

Table 31: VSX Alignment Requirements for SIMD Algorithms in Linear Algebra Subroutines.

Data Type	Vector and Matrix Alignment	Vector Stride	Leading Dimensions
Long-precision real	Quadword and doubleword	Varies depending on the type of subroutine: **1** For vector-scalar linear algebra subroutines **Any** For matrix-vector linear algebra subprograms	Any
Short-precision real	Doubleword and singleword	Varies depending on the type of subroutine: **1** For vector-scalar linear algebra subroutines **Any** For matrix-vector linear algebra subprograms	Any
Long-precision complex	Quadword	Any	Any
Short-precision complex	Doubleword	Varies depending on the type of subroutine: **1** For vector-scalar linear algebra subroutines **Any** For matrix-vector linear algebra subprograms	Any

Table 31: VSX Alignment Requirements for SIMD Algorithms in Linear Algebra Subroutines. (continued)

Data Type	Vector and Matrix Alignment	Vector Stride	Leading Dimensions

Note:

1. As long as the alignment requirements described in this table are met, you do not have to change your existing application programs that call ESSL to take advantage of the increased performance produced by the SIMD subroutines. However, you will obtain optimal performance for these subroutines when the following additional conditions are met:
   - Vectors and matrices are quadword aligned.
   - LDAs are multiples of 2 for real long-precision matrices.
   - LDAs are multiples of 4 for real short-precision matrices.
   - LDAs are multiples of 2 for complex short-precision matrices.
   - Stride is 1 for vectors.
2. If the alignment restrictions in the table are not met, in some cases attention message 2610 will be issued. The default behavior for message 2610 is for the message to be suppressed. To change the default behavior, see "ERRSET (ESSL ERRSET Subroutine)" on page 1191.

Table 32: VSX Alignment Requirements for SIMD Algorithms in Fourier Transform Subroutines and Convolution and Correlation Subroutines.

Data Type	Vector and Matrix Alignment	Stride Between Elements Within Sequence	Stride Between Sequences
Long-precision real	Quadword (see Notes "1" on page 36 and "2" on page 36)	1 (see Note "3" on page 36)	Multiple of 2 (see Note "3" on page 36)
Short-precision real	Doubleword	1 (see Note "3" on page 36)	Multiple of 4 (see Note "3" on page 36)
Long-precision complex	Quadword	Any	Any
Short-precision complex	Doubleword	1	Multiple of 2 (see Note "3" on page 36)

Table 32: VSX Alignment Requirements for SIMD Algorithms in Fourier Transform Subroutines and Convolution and Correlation Subroutines. (continued)

Data Type	Vector and Matrix Alignment	Stride Between Elements Within Sequence	Stride Between Sequences

Notes:

1. AUX1 must be aligned on a quadword boundary.
2. AUX and AUX2 must either be aligned on a quadword boundary or dynamically allocated.
3. For _COSF and _SINF, the stride between elements within a sequence and the stride between sequences can have any value.
4. As long as the alignment requirements described in this table are met, you do not have to change your existing application programs that call ESSL to take advantage of the increased performance produced by the SIMD subroutines. However, some subroutines require separate calls for initialization and computation, and it can occur that the alignment of an array meets the requirements during initialization but does not meet the requirements during computation. When this happens, in some cases one of the following happens:
   - Error 2152 will be issued and your program will terminate. If you want your program to continue processing, use ERRSET with an ESSL error exit routine, ENOTRM, to make error 2152 recoverable
   - Error 2211 will be issued and your program will terminate
5. If the alignment restrictions in this table are not met, in some cases one or more of the following attention messages will be issued:
   - 2610
   - 2611
   - 2612

   The default behavior for these messages is to be suppressed. To change the default behavior, see "ERRSET (ESSL ERRSET Subroutine)" on page 1191.

Table 33: ESSL Subroutines that Automatically Use SIMD Algorithms When Alignment Restrictions are Met on VSX-enabled Processors.

Subroutine Names
Vector-Scalar Linear Algebra Subprograms (See Note): ISAMAX, IDAMAX, ICAMAX, IZAMAX ISAMIN, IDAMIN ISMAX, IDMAX ISMIN, IDMIN SASUM, DASUM, SCASUM, DZASUM SAXPY, DAXPY, CAXPY, ZAXPY >|SAXPBY, DAXPBY|< SCOPY, DCOPY, CCOPY, ZCOPY SDOT, DDOT, CDOTU, ZDOTU, CDOTC, ZDOTC DNRM2, DZNRM2 DNORM2, ZNORM2 SROT, DROT, CROT, ZROT, CSROT, ZDROT SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, ZDSCAL SSWAP, DSWAP, CSWAP, ZSWAP SVEA, DVEA, CVEA, ZVEA SVES, DVES, CVES, ZVES SVEM, DVEM, CVEM SYAX, DYAX, CYAX, ZYAX, CSYAX, ZDYAX SZAXPY, DZAXPY, CZAXPY, ZZAXPY
Matrix-Vector Linear Algebra Subprograms (See Note): SGEMV, DGEMV, CGEMV, ZGEMV SGER, DGER, CGERU, ZGERU, CGERC, ZGERC SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV SSPR, DSPR, CHPR, ZHPR, SSYR, DSYR, CHER, ZHER SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, ZHER2 STRMV, DTRMV, CTRMV, ZTRMV STPMV, DTPMV, CTPMV, ZTPMV STRSV, DTRSV, CTRSV, ZTRSV, STPSV, DTPSV, CTPSV, ZTPSV
Matrix Operations (See Note): SGEMUL, DGEMUL, CGEMUL, ZGEMUL SGEMM, DGEMM, CGEMM, ZGEMM SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, ZHEMM STRMM, DTRMM, CTRMM, ZTRMM STRSM, DTRSM, CTRSM, ZTRSM SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, ZHERK SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, ZHER2K SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI, ZGECMI SGETMO, DGETMO, CGETMO, ZGETMO, CGECMO, ZGECMO

Table 33: ESSL Subroutines that Automatically Use SIMD Algorithms When Alignment Restrictions are Met on VSX-enabled Processors. (continued)

Subroutine Names
Fourier Transforms: SCFTD, DCFTD SRCFTD, DRCFTD SCRFTD, DCRFTD SCFT, DCFT SRCFT, DRCFT SCRFT, DCRFT SCOSF, DCOSF SSINF, DSINF SCFT2, DCFT2 SRCFT2, DRCFT2 SCRFT2, DCRFT2 SCFT3, DCFT3 SRCFT3, DRCFT3 SCRFT3, DCRFT3
Convolutions and Correlations: SCONF, SCORF, SACORF
Random Number Generation: SURNG, DURNG SNRNG, DNRNG
Note: Many of the dense and banded linear algebraic equations and eigensystem analysis subroutines make one or more calls to the vector-scalar, matrix-vector linear algebra, and matrix operation subroutines listed in this table, and therefore they indirectly use SIMD algorithms.

Multithreaded Subroutines Provided by ESSL

Table 34 on page 39 lists the multithreaded subroutines provided by ESSL and also indicates which of those subroutines use SIMD algorithms.

Table 34: Multithreaded Subroutines.

Subroutine Category	**Multithreaded Subroutine**	**Does this subroutine also use SIMD algorithms on VSX-enabled processors?** **See "SIMD Algorithms on VSX-Enabled Processors" on page 34.**
Vector-Scalar Linear Algebra Subprograms[1]	SASUM, DASUM, SCASUM, DZASUM	No
	SAXPY, DAXPY, CAXPY, ZAXPY	No
	>|SAXPBY, DAXPBY|<	No
	SCOPY, DCOPY, CCOPY, ZCOPY	No
	SDOT, DDOT, CDOTU, ZDOTU, CDOTC, ZDOTC	No
	SNDOT, DNDOT	No
	SNORM2, DNORM2, CNORM2, ZNORM2	No
	SROT, DROT, CROT, ZROT, CSROT, ZDROT	No
	SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, ZDSCAL	No
	SSWAP, DSWAP, CSWAP, ZSWAP	No
	SVEA, DVEA, CVEA, ZVEA	No
	SVES, DVES, CVES, ZVES	No
	SVEM, DVEM, CVEM, ZVEM	No
	SYAX, DYAX, CYAX, ZYAX, CSYAX, ZDYAX	No
	SZAXPY, DZAXPY, CZAXPY, ZZAXPY	No

Table 34: Multithreaded Subroutines. (continued)

Subroutine Category	**Multithreaded Subroutine**	**Does this subroutine also use SIMD algorithms on VSX-enabled processors?** **See "SIMD Algorithms on VSX-Enabled Processors" on page 34.**
Matrix-Vector Linear Algebra Subprograms[1]	SGEMV, DGEMV, CGEMV, ZGEMV	Yes
	SGER, DGER, CGERU, ZGERU, CGERC, ZGERC	Yes
	SSPMV, DSPMV, CHPMV, ZHPMV	Yes
	SSYMV, DSYMV, CHEMV, ZHEMV	Yes
	SSPR, DSPR, CHPR, ZHPR	Yes
	SSYR, DSYR, CHER, ZHER	Yes
	SSPR2, DSPR2, CHPR2, ZHPR2	Yes
	SSYR2, DSYR2, CHER2, ZHER2	Yes
	SGBMV[2], DGBMV[2]	No
	CGBMV[2], ZGBMV[2]	No
	SSBMV[2], DSBMV[2]	No
	CHBMV[2], ZHBMV[2]	No
	STRMV, DTRMV, CTRMV, ZTRMV	Yes except DTRMV and ZTRMV
	STPMV, DTPMV, CTPMV, ZTPMV	Yes except DTPMV and ZTPMV
	STRSV, DTRSV, CTRSV, ZTRSV	Yes except DTRSV and ZTRSV
	STPSV, DTPSV, CTPSV, ZTPSV	Yes except DTPSV, ZTPSV
	STBMV[2], DTBMV[2]	No
	CTBMV[2], ZTBMV[2]	No
Sparse Matrix Vector Linear Algebra Subprograms	DSMMX, DSDMX	No

Table 34: Multithreaded Subroutines. (continued)

Subroutine Category	**Multithreaded Subroutine**	**Does this subroutine also use SIMD algorithms on VSX-enabled processors?** **See "SIMD Algorithms on VSX-Enabled Processors" on page 34.**
Matrix Operations[1]	SGEADD, DGEADD, CGEADD, ZGEADD	No
	SGESUB, DGESUB, CGESUB, ZGESUB	No
	SGEMUL, DGEMUL, CGEMUL, ZGEMUL	Yes
	SGEMM, DGEMM, CGEMM, ZGEMM	Yes
	SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, ZHEMM	Yes
	STRMM, DTRMM, CTRMM, ZTRMM	Yes
	STRSM, DTRSM, CTRSM, ZTRSM	Yes
	SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, ZHERK	Yes
	SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, ZHER2K	Yes
	SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI, ZGECMI	Yes
	SGETMO, DGETMO, CGETMO, ZGETMO, CGECMO, ZGECMO	Yes
Dense Linear Algebraic Equations	SGESV, DGESV, CGESV, ZGESV	See Note "1" on page 42
	SGEF, DGEF, CGEF, ZGEF	See Note "1" on page 42
	SGES, DGES, CGES, ZGES	See Note "1" on page 42
	SGETRF, DGETRF, CGETRF, ZGETRF	See Note "1" on page 42
	SGETRS, DGETRS, CGETRS, ZGETRS	See Note "1" on page 42
	SPPSV, DPPSV, CPPSV, ZPPSV	See Note "1" on page 42
	SPPF, DPPF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, DPOF, DPOTRF	See Note "1" on page 42
	SPPTRS, DPPTRS, CPPTRS, ZPPTRS	See Note "1" on page 42
	SPOSV, DPOSV, CPOSV, ZPOSV	See Note "1" on page 42
	SPOSM, DPOSM, CPOSM, ZPOSM	See Note "1" on page 42
	SPPFCD[3], DPPFCD[3], DPOFCD[3]	See Note "1" on page 42
	SPPTRI, DPPTRI, CPPTRI, ZPPTRI, SPPICD[3], DPPICD[3], DPOICD[3]	See Note "1" on page 42
	STRI, DTRI, STRTRI, DTRTRI, CTRTRI, ZTRTRI	See Note "1" on page 42

Table 34: Multithreaded Subroutines. (continued)

Subroutine Category	**Multithreaded Subroutine**	**Does this subroutine also use SIMD algorithms on VSX-enabled processors?** **See "SIMD Algorithms on VSX-Enabled Processors" on page 34.**
Banded Linear Algebraic Equations	SGBSV, DGBSV, CGBSV, ZGBSV	See Note "1" on page 42
	SGBTRS, DGBTRS, CGBTRS, ZGBTRS	See Note "1" on page 42
	SPBSV, DPBSV, CPBSV, ZPBSV	See Note "1" on page 42
	SPBTRS, DPBTRS, CPBTRS, ZPBTRS	See Note "1" on page 42
Sparse Linear Algebraic Equations	DSRIS[4]	No
Linear Least Squares	SGEQRF, DGEQRF, CGEQRF, ZGEQRF	See Note "1" on page 42
Fourier Transforms	SCFTD, SRCFTD, SCRFTD, SCFT, SRCFT, SCRFT, SCFT2, SRCFT2, SCRFT2, SCFT3, SRCFT3, SCRFT3	Yes
	DCFTD, DRCFTD, DCRFTD, DCFT, DRCFT, DCRFT, DCFT2, DRCFT2, DCRFT2, DCFT3, DRCFT3, DCRFT3	Yes
Convolution and Correlation	SCOND, SCORD	No
	SDCON, SDCOR, DDCON, DDCOR	No
	SCONF, SCORF SACORF	Yes

Note:

1. Many of the dense and banded linear algebraic equations and eigensystem analysis subroutines make one or more calls to the vector-scalar, matrix-vector linear algebra, and matrix operation subroutines listed in this table, and therefore they indirectly use multiple threads and SIMD algorithms.
2. The Level 2 Banded BLAS use multiple threads only when the bandwidth is sufficiently large.
3. Multiple threads are used for the factor or inverse computation.
4. DSRIS only uses multiple threads when `IPARM(4)` = 1 or 2.

Using the ESSL SMP CUDA Library

You can use the ESSL SMP CUDA Library in two ways for the subset of ESSL Subroutines that are GPU-enabled:

- Using NVIDIA GPUs for the bulk of the computation
- Using a hybrid combination of Power CPUs and NVIDIA GPUs

The ESSL SMP CUDA Library Linear Algebra subroutines leverage ESSL BLAS, NVIDIA cuBLAS, and blocking techniques to handle problem sizes larger than the GPU memory size. The algorithms support multiple GPUs and are designed for use in both SMP and MPI applications. The ESSL SMP CUDA Library Fourier Transform subroutines leverage NVIDIA CUDA Fast Fourier Transform library (cuFFT) and use GPUs only when the GPU memory is large enough to accommodate the computation (data and working space) of a single transform.

The ESSL SMP CUDA Library contains GPU-enabled versions of the following subroutines:

Table 35: List of ESSL SMP CUDA subroutines that are GPU-enabled.

Type of subroutine	**Subroutine name**
Matrix Operations	• SGEMM, DGEMM, CGEMM, and ZGEMM • SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM • STRMM, DTRMM, CTRMM, and ZTRMM • STRSM, DTRSM, CTRSM, and ZTRSM • SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK • SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K
Dense Linear Algebraic Equations	• SGESV, DGESV, CGESV, and ZGESV • SGETRF, DGETRF, CGETRF, and ZGETRF • SGETRS, DGETRS, CGETRS, and ZGETRS • SPPSV, DPPSV, CPPSV, and ZPPSV • SPPTRF, DPPTRF, CPPTRF, and ZPPTRF • SPPTRS, DPPTRS, CPPTRS, and ZPPTRS • SPOSV, DPOSV, CPOSV, and ZPOSV • SPOTRF, DPOTRF, CPOTRF, and ZPOTRF • SPOTRS, DPOTRS, CPOTRS, and ZPOTRS
Linear Least Squares	• SGEQRF, DGEQRF, CGEQRF, and ZGEQRF • SGELS, DGELS, CGELS, and ZGELS
Fourier Transforms	• SCFTD and DCFTD • SRCFTD and DRCFTD • SCRFTD and DCRFTD

Note: In the descriptions that follow *host* refers to the Power server and *device* refers to the GPU

To use the ESSL SMP CUDA Library, you must specify only host arrays as arguments and link your applications using -lesslsmpcuda (see Chapter 5, “Processing Your Program,” on page 169). If desired, you can change the default behavior of the ESSL SMP CUDA Library using either environment variables or the SETGPUS subroutine, see “ESSL SMP CUDA Library Options” on page 44.

For information on the NVIDIA CUDA support, see the following:

```
http://developer.nvidia.com/cuda-toolkit
```

ESSL Support for NVIDIA GPU Compute Modes

NVIDIA allows you to use GPU compute modes to control how application threads run on the GPU.

Restriction: ESSL requires all visible GPUs to be set to the same compute mode, except for those in PROHIBITED mode, which ESSL ignores.

The NVIDIA compute modes are as follows:

0 DEFAULT
Multiple host threads can use the device at the same time.

ESSL can use one or more visible GPUs on the host. See "ESSL SMP CUDA Library Options" on page 44 for information on the CUDA_VISIBLE_DEVICES environment variable.

2 PROHIBITED

No host thread can use the device.

ESSL does not use any GPUs in PROHIBITED compute mode; it uses only the GPUs in other compute modes. If all GPUs are in PROHIBITED compute mode, ESSL issues attention message 2538-2614 and runs using CPUs only, ignoring the setting of the ESSL_CUDA_HYBRID environment variable. See "ESSL SMP CUDA Library Options" on page 44 for information on the ESSL_CUDA_HYBRID environment variable.

3 EXCLUSIVE_PROCESS

Only one context is allowed per device, usable from multiple threads at a time.

ESSL can use one or more visible GPUs on the host. If the CUDA MPS[1] is being used with more than 1 GPU, you can use the SETGPUS subroutine or the environmental variable CUDA_VISIBLE_DEVICES with the local rank of the MPI tasks to select the different GPUs for MPI tasks that you want ESSL to use. See "ESSL SMP CUDA Library Options" on page 44 for information on the CUDA_VISIBLE_DEVICES environment variable.

ESSL SMP CUDA Library Options

The ESSL SMP CUDA Library allows you to control these options:

Control how many and which GPUs ESSL uses

By default, ESSL uses all devices. Use the CUDA_VISIBLE_DEVICES environment variable or the SETGPUS subroutine to change this default. The CUDA applications will see only the devices whose index is specified in the CUDA_VISIBLE_DEVICES environmental variable, and the devices are enumerated in the order of the sequence specified. For example, if you have three GPUs defined, 0, 1, 2, you can specify that a CUDA application use only a subset of the GPUs, 1 and 2, using the environmental variable as follows:

```
export CUDA_VISIBLE_DEVICES=1,2
```

You can also specify a new order in which your three GPUs are enumerated:

```
export CUDA_VISIBLE_DEVICES=2,1,0
```

If you need different MPI tasks to use different GPUs, you can use the SETGPUS subroutine or the environmental variable CUDA_VISIBLE_DEVICES with the local rank of the MPI tasks to ensure each task uses unique GPUs. See "SETGPUS (Set the Number of GPUs and Identify Which GPUs ESSL Should Use)" on page 1194.

In some cases ESSL does not use GPUs:

- The GPU-enabled subroutine is called from within an OpenMP parallel construct (OMP_IN_PARALLEL is true).
- For pre- and post-scaling operations, for example, handling the alpha argument in _TRMM.
- When the problem size is too small to benefit from using GPUs.
- For the Fourier transform subroutines, when the transform length or data layout are not supported by the NVIDIA cuFFT library or when the problem size is too big to fit in the GPU memory.

Specifying Whether ESSL Runs in Hybrid Mode

By default, the ESSL SMP CUDA Library runs in hybrid mode. Use the `ESSL_CUDA_HYBRID` environment variable to change this default (valid values are `yes` or `no`).

[1] NVIDIA CUDA Multi Process Service (MPS) is a feature that allows multiple CUDA processes to share a single GPU context.

The default hybrid mode for the Linear Algebra subroutines (`ESSL_CUDA_HYBRID=yes`) means that the ESSL SMP CUDA Library subroutines can run on both Power CPUs and NVIDIA GPUs.

Subroutines SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K only use the Power CPUs for scaling operations.

For the Fourier Transform subroutines, `ESSL_CUDA_HYBRID=yes` means that the subroutines can run on either CPUs or NVIDIA GPUs (not both), depending on performance. `ESSL_CUDA_HYBRID=no` means that the subroutines must run on GPUs if the transform length and data layout are supported by NVIDIA cuFFT.

Specifying Whether ESSL Pins Host Memory Buffers

By default, ESSL does not pin host memory buffers (ESSL_CUDA_PIN=no). Use the ESSL_CUDA_PIN environment variable to change this default (valid values are yes, no, or pinned).

If you want ESSL to pin your host memory buffers on entry to gpu-enabled subroutines and unpin them before returning, specify ESSL_CUDA_PIN=yes.

Performance might be improved if you pin your host memory buffers used in the ESSL calling sequences once before any calls to ESSL subroutines. To pin your host memory buffers use the NVIDIA CUDA subroutine cudaHostRegister. If you pin your own buffers you should specify ESSL_CUDA_PIN=pinned.

Note: Host memory buffers that are only partially pinned may lead to NVIDIA Error 11 from cublasSetMatrixAsync or cublasSetMatrix.

How ESSL Assigns Threads

The ESSL SMP CUDA Library requires at least one OpenMP thread for each GPU used. If the number of OpenMP threads is less than the number of GPUs, ESSL issues attention message 2538-2615 and uses the same number of GPUs as there are OpenMP threads.

ESSL SMP CUDA Library uses the following priorities to assign threads:

- ESSL reserves 1 thread for each GPU used
- Some ESSL subroutines might reserve threads needed to support multiple streams
- The remaining threads are used for the CPU, but a subroutine might not run in hybrid mode if there are not enough threads left or if the problem size is too small.

MPI Applications

There are two ways to use the ESSL SMP CUDA Library with MPI Applications depending on how the GPUs are used by the local MPI tasks:

- GPUs are not shared, meaning that each MPI task on a node uses unique GPUs. You can use the SETGPUS subroutine or the environmental variable CUDA_VISIBLE_DEVICES with the local rank of the MPI tasks to ensure that each task uses unique GPUs. See the OMPI_COMM_WORLD_LOCAL_RANK description at the following Open MPI URL:

  ```
  https://www.open-mpi.org/
  ```

- >|GPUs are shared, meaning that the number of MPI tasks per node oversubscribe the GPUs. For this case, it is recommend that you run by using the NVIDIA MPS. The NVIDIA MPS is a runtime service that is designed to let multiple MPI processes by using CUDA run concurrently on a single GPU in a way that is transparent to the MPI program. NVIDIA MPS supports at most 48 MPI tasks per V100 GPU and 16 MPI tasks per P100 GPUs, but if you are using ESSL, it is recommended that you use Core Affinity and no more tasks than the number of cores that are being used.

  If you are sharing GPUs, it is possible that ESSL cannot allocate workspace on the GPU. To reduce the amount of context local storage per MPI task if you are using V100 GPUs, set the `CUDA_MPS_ACTIVE_THREADS_PERCENTAGE` environment variable to 200/n, where n is the number of MPI tasks per GPU. If possible, reduce the number of MPI tasks per node or increase the number of GPUs that are being used per node to potentially eliminate the allocation failures.

If error `cudaStreamCreate failed with CUDA message: all CUDA-capable devices are busy or unavailable` occurs when you are using ESSL with MPI applications and NVIDIA MPS, confirm that the NVIDIA MPS Daemons are running on all nodes that the MPI job is using.

|<

You can use SETGPUS (see "SETGPUS (Set the Number of GPUs and Identify Which GPUs ESSL Should Use)" on page 1194) or the environmental variable CUDA_VISIBLE_DEVICES with the local rank of the MPI tasks to inform ESSL which GPUs your MPI Tasks can use.

For best performance, consider increasing the block size that you are using to distribute your data across the MPI tasks. Consider block sizes in the range 1024 - 4096 elements.

NVIDIA GPU Power Capping

The ESSL and NVIDIA library subroutines are highly optimized and for some problem sizes your application may exceed the SW Power cap for one or more of the GPUs. If this happens your performance will be degraded because the frequency of the corresponding GPU clock will be reduced because the GPU is consuming too much power.

You can confirm that this is happening by using nvidia-smi to monitor the GPUs while your application is running.

```
nvidia-smi dmon
```

If you wish to adjust the Power Cap Limit follow these steps:

1. Determine the current, default and maximum power limit as follows:

   ```
   nvidia-smi -q | grep 'Power Limit'
   ```

2. Ensure that persistence mode is being used.
3. Increase the SW Power Cap limit for all GPUs as follows, where *xxx* is the desired value in watts:

   ```
   nvidia-smi -pl xxx
   ```

Note: You must increase the power limit and set persistence each time the server is booted.

For additional information, see the following URL:

```
http://international.download.nvidia.com/tesla/pdf/gpu-boost-tesla-k40-app-note.pdf
```

What Type of Data Are You Processing in Your Program?

The version of the ESSL subroutine you select should agree with the data you are using. ESSL provides a short- and long-precision version of most of its subroutines processing short- and long-precision data, respectively. In a few cases, it also provides an integer version processing integer data or returning just integer data. The subroutine names are distinguished by a one- or two-letter prefix based on the following letters:

S for short-precision real
D for long-precision real
C for short-precision complex
Z for long-precision complex
I for integer

The precision of your data affects the accuracy of your results. This is discussed in "Getting the Best Accuracy" on page 60. For a description of these data types, see "How Do You Set Up Your Scalar Data?" on page 47.

How Is Your Data Structured? And What Storage Technique Are You Using?

Some subroutines process specific data structures, such as sparse vectors and matrices or dense and banded matrices. In addition, these data structures can be stored using various storage techniques. You should select the proper subroutine on the basis of the type of data structure you have and the storage technique you want to use. If possible, you should use a storage technique that conserves storage and potentially improves performance. For more about storage techniques, see "Setting Up Your Data" on page 47.

What about Performance and Accuracy?

ESSL provides variations among some of its subroutines. You should consider performance and accuracy when deciding which subroutine is the best to use. Study "Function" in each subroutine description. It helps you understand exactly what each subroutine does, and helps you determine which subroutine is best for you. For example, some subroutines perform multiple computations of a certain type. This might give you better performance than a subroutine that does each computation individually. In other cases, one subroutine may do scaling while another does not. If scaling is not necessary for your data, you get better performance by using the subroutine without scaling.

Avoiding Conflicts with Internal ESSL Routine Names That are Exported

Do not use names for your own subroutines, functions, and global variables that are the same as the ESSL exported names. Internal ESSL routine names that are exported all begin with the **ESV** prefix. Therefore, it is sufficient for you to avoid using this prefix for your own names.

Setting Up Your Data

There are various items to consider when setting up your scalar and array data.

How Do You Set Up Your Scalar Data?

A scalar item is a single item of data, whether it is a constant, a variable, or an element of an array. ESSL assumes that your scalar data conforms to the appropriate standards. The scalar data types and how you should code them for each programming language are listed in "Coding Your Scalar Data" specific to each language in Chapter 4, "Coding programs," on page 125.

Scalar data passed to ESSL from all types of programs, including Fortran, C, and C++, should conform to the ANSI/IEEE 32-bit and 64-bit binary floating-point format, as described in the *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754–1985*.

How Do You Set Up Your Arrays?

An array represents an area of storage in your program, containing data stored in a series of locations. An array has a single name. It is made up of one or more pieces of scalar data, all the same type. These are the elements of the array. It can be passed to the ESSL subroutine as input, returned to your program as output, or used for both input and output, in which case the original contents are overwritten.

Arrays can contain conceptual (mathematical) data structures, such as vectors, matrices, or sequences. There are many different types of data structures. Each type of data structure requires a unique arrangement of data in an array and does not necessarily have to include all the elements of the array. In addition, the elements of these data structures are not always contiguous in storage within an array. Stride and leading dimension arguments passed to ESSL subroutines define the separations in array storage for the elements of the vector, matrix, and sequence. All these aspects of data structures are described in Chapter 3, "Setting up data structures," on page 71. You must first understand array storage techniques to fully understand the concepts of data structures, stride, and leading dimension, especially if you are using them in unconventional ways.

ESSL subroutines assume that all arrays passed to them are stored using the Fortran array storage techniques (in column-major order), and they process your data accordingly. For details, see "Setting Up Arrays in Fortran" on page 126. On the other hand, C, and C++ programs store arrays in row-major order. For details on what you can do, see:

- For C, see "Setting Up Arrays in C" on page 146.
- For C++, see "Setting Up Arrays in C++" on page 159.

How Should Your Array Data Be Aligned?

All arrays, regardless of the type of data, should be aligned on a doubleword boundary to ensure optimal performance.

For all subroutines running on VSX enabled processors, see "SIMD Algorithms on VSX-Enabled Processors" on page 34.

For information about how your programming language aligns data, see your programming language manuals.

What Storage Mode Should You Use for Your Data?

The amount of storage used by arrays and the storage arrangement of data in the arrays can affect overall program performance. As a result, ESSL provides subroutines that operate on different types of data structures, stored using various storage modes. You should chose a storage mode that conserves storage and potentially improves performance. For definitions of the various data structures and their corresponding storage modes, see Chapter 3, "Setting up data structures," on page 71. You can also find special storage considerations, where applicable, in "Notes" in each subroutine description.

How Do You Convert from One Storage Mode to Another?

ESSL provides conversion subroutines and sample programs to help you convert from one storage mode to another.

Conversion Subroutines

ESSL provides several subroutines that help you convert from one storage mode to another:

- DSRSM is used to migrate your existing program from sparse matrices stored by rows to sparse matrices stored in compressed-matrix storage mode. This converts the matrices into a storage format that is compatible with the input requirements for some ESSL sparse matrix subroutines, such as DSMMX.
- DGKTRN and DSKTRN are used to convert your sparse matrix from one skyline storage mode to another, if necessary, before calling the subroutines DGKFS/DGKFSP or DSKFS/DSKFSP, respectively.

Sample Programs

In addition, sample programs are provided with many of the storage mode descriptions in Chapter 3, "Setting up data structures," on page 71. You can use these sample programs to convert your data to the desired storage mode by adapting them to your application program.

Setting Up Your ESSL Calling Sequences

This gives the general rules for setting up the ESSL calling sequences. The information given here applies to all types of programs, running in all environments. For a description and examples of how to code the ESSL calling sequences in your particular programming language, see the following:

- "Fortran Programs" on page 125
- "C Programs" on page 142
- "C++ Programs" on page 155

What Is an Input-Output Argument?

Some arguments are used for both input and output. The contents of the input argument are overlaid with the output value(s) on return to your program. Be careful that you save any data you need to preserve before calling the ESSL subroutine.

What Are the General Rules to Follow when Specifying Data for the Arguments?

You should follow the syntax rules given for each argument in “On Entry” in the subroutine description. Input-argument error messages may be issued, and your program may terminate when you make an error specifying the input arguments. For example:

- Data passed to ESSL must be of the correct type: 32-bit or 64-bit integer, 32-bit or 64-bit logical, character, real, complex, short-precision, or long-precision. There is no conversion of data. Assuming you are using the ESSL header file with your C and C++ programs, you first need to define the following:
  - Complex and logical data in C programs, using the guidelines in “Setting Up Complex Data Types in C” on page 145 and “Using Logical Data in C” on page 145.
  - Short-precision complex and logical data in C++ programs, using the guidelines in “Using Logical Data in C++” on page 159.
- Character values must be one of the specified values. For example, it may have to be 'N', 'T', or 'C'.
- Numeric values must fall within the correct range for that argument. For example, a numeric value may need to be greater than or equal to 0, or it may have to be a nonzero value.
- Arrays must be defined correctly; that is, they must have the correct dimensions, or the dimensions must fall within the correct range. For example, input and output matrices may need to be conformable, or the number of rows in the matrix must be less than or equal to the leading dimension specified. (ESSL assumes all arrays are stored in column-major order.)

What Happens When a Value of 0 Is Specified for N?

For most ESSL subroutines, if you specify 0 for the number of elements to be processed in a vector or the order of a matrix (usually argument *n*), no computation is performed. After checking for input-argument errors, the subroutine returns immediately and no result is returned. In the other subroutines, an error message may be issued.

How Do You Specify the Beginning of the Data Structure in the ESSL Calling Sequence?

When you specify a vector, matrix, or sequence in your calling sequence, it does not necessarily have to start at the beginning of the array. It can begin at any point in the array. For example, if you want vector ***x*** to start at element 3 in array A, which is declared `A(1:12)`, specify `A(3)` in your calling sequence for argument *x*, such as in the following SASUM calling sequence in your Fortran program:

```
             N    X   INCX
             |    |     |
X = SASUM( 4 , A(3) , 2 )
```

Also, for example, if you want matrix ***A*** to start at the second row and third column of array A, which is declared `A(0:10,2:8)`, specify `A(1,4)` in your calling sequence for argument *a*, such as in the following SGEADD calling sequence in your Fortran program:

```
                  A     LDA  TRANSA  B  LDB TRANSB  C  LDC  M   N
                  |      |     |     |   |    |     |   |   |   |
CALL  SGEADD( A(1,4) , 11 , 'N'  , B , 4 , 'N'  , C , 4 , 4 , 3 )
```

For more examples of specifying vectors and matrices, see Chapter 3, “Setting up data structures,” on page 71.

Using Auxiliary Storage in ESSL

For the ESSL subroutines listed in Table 36 on page 50, you need to provide extra working storage to perform the computation. It is necessary to understand the use of dynamic allocation for providing auxiliary storage in ESSL and, if dynamic allocation is not an option, how to calculate the amount of auxiliary storage you need by use of formulas or error-handling capabilities provided in ESSL.

Auxiliary storage, or working storage, is supplied through one or more arguments, such as *aux*, in the calling sequence for the ESSL subroutine. **If the working storage does not need to persist after the subroutine call, it is suggested you use dynamic allocation.** For example, in the Fourier Transforms subroutines, you may allocate *aux2* dynamically, but not *aux1*. See the subroutine descriptions for details and variations.

Table 36: ESSL Subroutines Requiring Auxiliary Working Storage.

Subroutine Names
Linear Algebra Subprograms: DSMTM
Matrix Operations: _GEMMS
Dense Linear Algebraic Equations: _GEFCD _PPFCD _GEICD _PPICD _POFCD _POICD DGEFP$^{\Delta}$ DPPFP$^{\Delta}$
Sparse Linear Algebraic Equations: DGSF DGSS DGKFS DGKFSP$^{\Delta}$ DSKFS DSKFSP$^{\Delta}$ DSRIS DSMCG DSDCG DSMGCG DSDGCG
Linear Least Squares: _GESVF _GELLS
Fourier Transforms: _CFTD _RCFTD _CRFTD _CFT _RCFT _CRFT _COSF _SINF SCOSFT$^{\Delta}$ _CFT2 _RCFT2 _CRFT2 _CFT3 _RCFT3 _CRFT3 SCFTP$^{\Delta}$ SCFT2P$^{\Delta}$ SCFT3P$^{\Delta}$
Convolutions and Correlations: SCONF SCORF SACORF
Related Computations: _WLEV
Interpolation: _TPINT _CSIN2
Random Number Generation: _NRAND
Utilities: DGKTRN DSKTRN

Table 36: ESSL Subroutines Requiring Auxiliary Working Storage. (continued)
Subroutine Names
Δ Documentation for this subroutine is no longer provided. The *aux* and *naux* arguments for the subroutine are specified the same as for the corresponding serial ESSL subroutine.

Dynamic Allocation of Auxiliary Storage

Dynamic allocation for the auxiliary storage is performed when error 2015 is unrecoverable and *naux* = 0. For details on which *aux* arguments allow dynamic allocation, see the subroutine descriptions.

Setting Up Auxiliary Storage When Dynamic Allocation Is Not Used

You set up the storage area in your program and pass it to ESSL through arguments, specifying the size of the *aux* work area in the *naux* argument.

Who Do You Want to Calculate the Size of Auxiliary Storage? You or ESSL?

You have a choice of two methods for determining how much auxiliary storage you should specify:

- Use the formulas provided in the subroutine description to derive **sufficient values** for your current and future needs. Use them if **ease of migration** to future machines and future releases of ESSL is your primary concern. For details, see "How Do You Calculate the Size of Auxiliary Storage Using the Formulas?" on page 51.
- Use the ESSL error-handling facilities to return to you a **minimum value** for the particular processor you are currently running on. (Values vary by platform.) Use this approach if **conserving storage** is your primary concern. For details, see "How Do You Get ESSL to Calculate the Size of Auxiliary Storage Using ESSL Error Handling?" on page 51.

How Do You Calculate the Size of Auxiliary Storage Using the Formulas?

The formulas provided for calculating *naux* indicate a **sufficient** amount of auxiliary storage required, which, in most cases, is larger than the minimum amount, returned by ESSL error handling. There are two types of formulas:

- **Simple formulas**

  These are given in the *naux* argument syntax descriptions. In general, these formulas result in the minimum required value, but, in a few cases, they provide overestimates.
- **Processor-independent formulas**

  These are given separately in each subroutine description. In general, these provide overestimates.

Both types of formulas provide values that are sufficient for all processors. As a result, you can migrate to any other processor and to future releases of ESSL without being concerned about having to increase the amount of storage for *aux*. You do, of course, need to weigh your storage requirements against the convenience of using this larger value.

To calculate the amount of storage using the formulas, you must substitute values for specific variables, such as *n*, *m*, *n1*, or *n2*. These variables are arguments specified in the ESSL calling sequence or derived from the arguments in the calling sequence.

How Do You Get ESSL to Calculate the Size of Auxiliary Storage Using ESSL Error Handling?

When getting ESSL to calculate auxiliary storage, ask yourself which of the following ways you prefer to obtain the information from ESSL:

- **By leaving error 2015 unrecoverable,** you can obtain the minimum required value of *naux* from the input-argument error message, but your program terminates.
- **By making error 2015 recoverable,** you can obtain the minimum required value of *naux* from the input-argument error message and have the updated *naux* argument returned to your program.

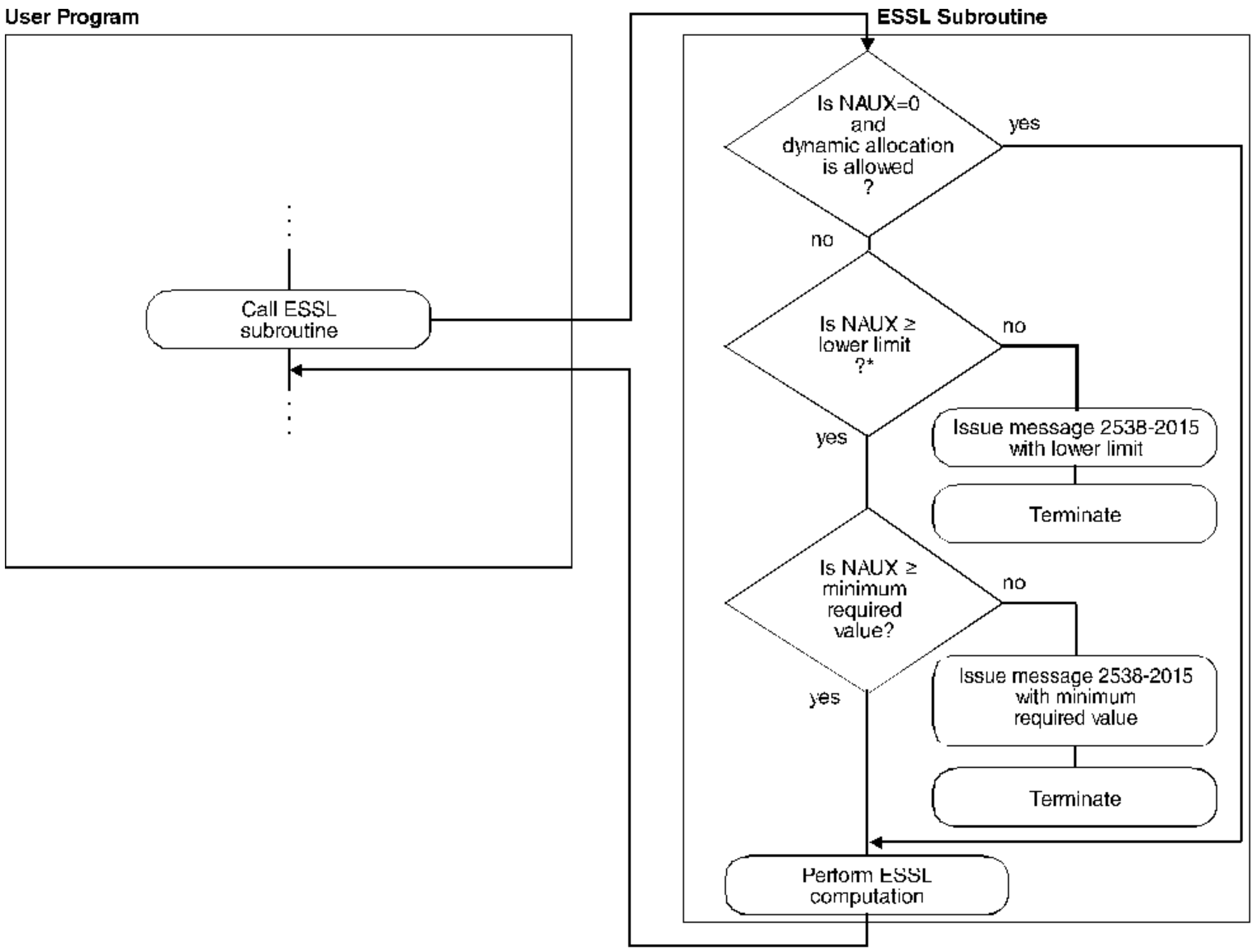
For both techniques, the amount returned by the ESSL error-handling facility is the **minimum** amount of auxiliary storage required to run your program successfully **on the particular processor you are currently running on.** The ESSL error-handling capability usually returns a smaller value than you derive by using the formulas listed for the subroutine. This is because the formulas provide a good estimate, but ESSL can calculate exactly what is needed on the basis of your data.

The values returned by ESSL error handling **may not apply to future processors.** You should not use them if you plan to run your program on a future processor. You should use them only if you are concerned with minimizing the amount of auxiliary storage used by your program.

Having ESSL Calculate Auxiliary Storage Size with Unrecoverable Error 2015

In this case, you obtain the minimum required value of *naux* from the error message, but your program terminates. The following description assumes that dynamic allocation is not selected as an option.

Leave error 2015 as unrecoverable, without calls to EINFO and ERRSET. Run your program with the *naux* values smaller than required by the subroutine for the particular processor you are running on. As a general guideline, specify values smaller than those listed in the formulas. However, if a lower limit is specified in the syntax (only for several *naux1* arguments in the Fourier transform, convolution, and correlation subroutines), you should not go below that limit. The ESSL error monitor returns the necessary sizes of the *aux* storage areas in the input-argument error message. This does, however, terminate your program when the error is encountered. (If you accidentally specify a sufficient amount of storage for the ESSL subroutine to perform the computation, error handling does not issue an error message and processing continues normally.) Figure 1 on page 53 illustrates what happens when error 2015 is unrecoverable.


User Program
ESSL Subroutine
Call ESSL subroutine
Is NAUX=0 and dynamic allocation is allowed ?
yes
no
Is NAUX ≥ lower limit ?*
no
yes
Issue message 2538-2015 with lower limit
Terminate
Is NAUX ≥ minimum required value?
no
yes
Issue message 2538-2015 with minimum required value
Terminate
Perform ESSL computation

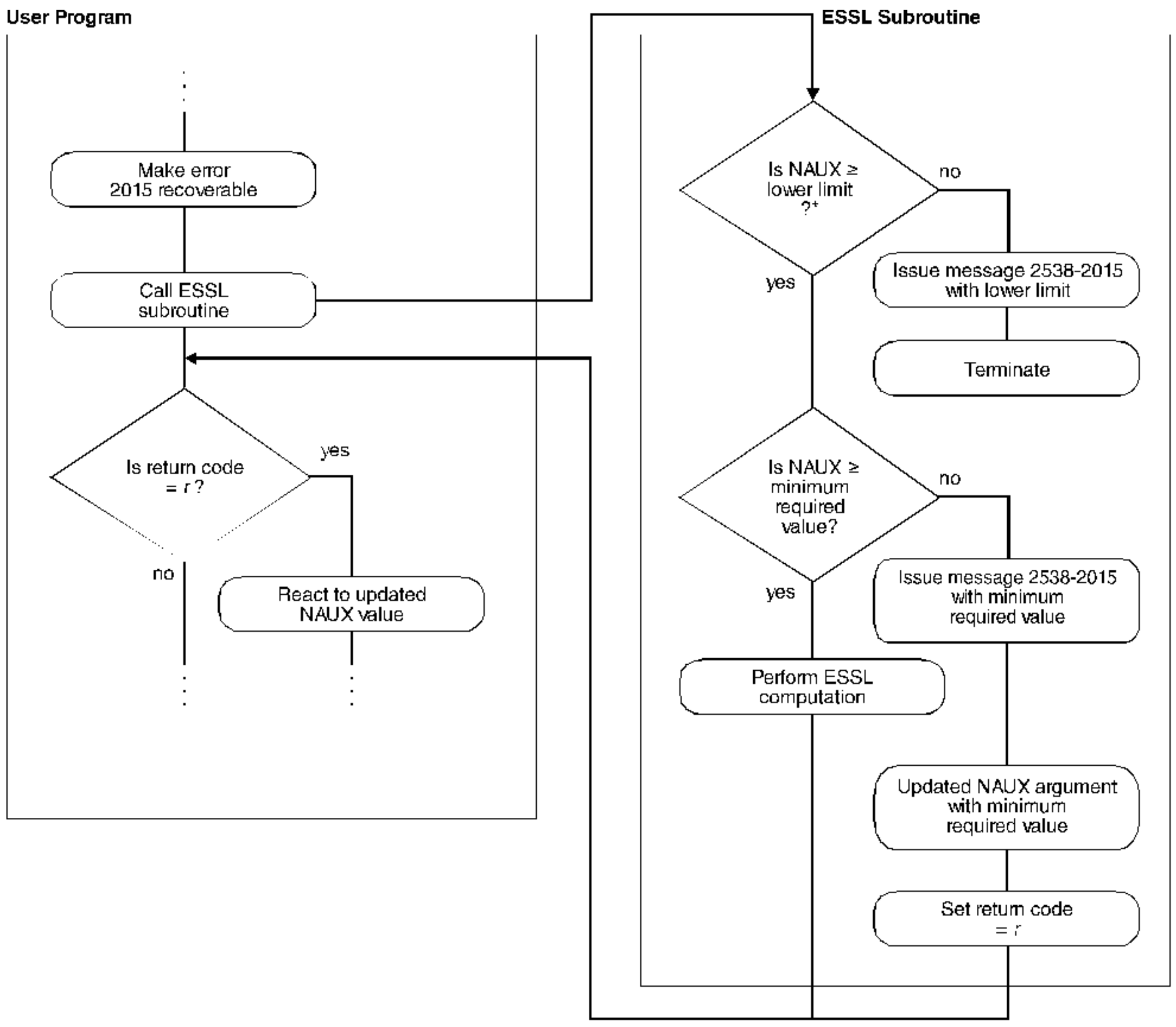
* This check applies only to several NAUX1 arguments in the Fourier transform, convolution, and correlation
subroutines.

Figure 1: How to Obtain an NAUX Value from an Error Message, but Terminate

Having ESSL Calculate Auxiliary Storage Size with Recoverable Error 2015

In this case, you obtain the minimum required value of *naux* from the error message and from the updated *naux* argument returned to your program.

Use EINFO and ERRSET with an ESSL error exit routine, ENOTRM, to make error 2015 recoverable. This allows you to dynamically determine in your program the minimum sizes required for the auxiliary working storage areas, specified in the *naux* arguments. Run your program with the *naux* values smaller than required by the subroutine for the particular processor you are running on. As a general guideline, specify values smaller than those listed in the formulas. However, if a lower limit is specified in the syntax (only for several *naux1* arguments in the Fourier transform, convolution, and correlation subroutines), you should not go below that limit. The ESSL error monitor returns the necessary sizes of the *aux* storage areas in the input-argument error message and a return code is passed back to your program, indicating that updated values are also returned in the *naux* arguments. You can then react to these updated values during run time in your program. ESSL does not perform any computation when this error occurs. For details on how to do this, see Chapter 4, “Coding programs,” on page 125. (If you accidentally specify a sufficient amount of storage for the ESSL subroutine to perform the computation, error handling does not issue an error message and processing continues normally.) Figure 2 on page 54 illustrates what happens when error 2015 is recoverable.


User Program
ESSL Subroutine
Make error 2015 recoverable
Call ESSL subroutine
Is return code = r?
yes
no
React to updated NAUX value
Is NAUX ≥ lower limit ?*
no
yes
Issue message 2538-2015 with lower limit
Terminate
Is NAUX ≥ minimum required value?
no
yes
Issue message 2538-2015 with minimum required value
Perform ESSL computation
Updated NAUX argument with minimum required value
Set return code = r

* This check applies only to several NAUX1 arguments in the Fourier transform, convolution, and correlation
subroutines.

Figure 2: How to Obtain an NAUX Value from an Error Message and in Your Program

Example of Input-Argument Error Recovery for Auxiliary Storage Sizes

The following example illustrates all the actions taken by the ESSL error-handling facility for each possible value of a recoverable input argument, *naux*. A key point here is that if you want to have the updated argument value returned to your program, you must make error 2015 recoverable and then specify an *naux* value greater than or equal to 20 and less than 300. For values out of that range, the error recovery facility is not in effect. (These values of *naux*, 20 and 300, are used only for the purposes of this example and do not relate to any of the ESSL subroutines.)

NAUX	Meaning of the NAUX Value
20	Lower limit of *naux* required for using recoverable input-argument error-handling facilities in ESSL. (This applies only to several *naux1* arguments in the Fourier transform, convolution, and correlation subroutines. You can find the lower limit in the syntax description for the *naux1* argument. For a list of subroutines, see "Using Auxiliary Storage in ESSL" on page 49.)

300
Minimum value of *naux*, required for successful running (on the processor the program is being run on).

Table 37 on page 55 describes the actions taken by ESSL in every possible situation for the values given in this example.

Table 37: Example of Input-Argument Error Recovery for Auxiliary Storage Sizes.

NAUX Value	Action When 2015 Is an Unrecoverable Input-Argument Error	Action When 2015 Is a Recoverable Input-Argument Error
naux < 20	An input-argument error message is issued. The value in the error message is the lower limit, 20. The application program stops.	An input-argument error message is issued. The value in the error message is the lower limit, 20. The application program stops.
20 ≤ *naux* < 300	An input-argument error message is issued. The value in the error message is the minimum required value, 300. The application program stops.	ESSL returns the value of *naux* as 300 to the application program, and an input-argument error message is issued. The value in the error message is the minimum required value, 300. ESSL does no computation, and control is returned to the application program.
naux ≥ 300	Your application program runs successfully.	Your application program runs successfully.

Coding Your Program to Obtain Auxiliary Storage Sizes

If you leave error 2015 unrecoverable, you **do not code anything** in your program. You just look at the error messages to get the sizes of auxiliary storage. On the other hand, if you want to make error 2015 recoverable to obtain the auxiliary storage sizes dynamically in your program, you need to **add some coding statements** to your program. For details on coding these statements in each programming language, see the following examples:

- For Fortran, see "Input-Argument Errors in Fortran Example" on page 136
- For C, see "Input-Argument Errors in C Example" on page 151
- For C++, see "Input-Argument Errors in C++ Example" on page 164

You may want to provide a separate subroutine to calculate the auxiliary storage size whenever you need it. Figure 3 on page 56 shows how you might code a separate Fortran subroutine. Before calling SCFT in your program, call this subroutine, SCFTQ, which calculates the minimum size and stores it in the *naux* arguments. Upon return, your program checks the return code. If it is nonzero, the *naux* arguments were updated, as planned. You should then make sure adequate storage is available and call SCFT. On the other hand, if the return code is zero, error handling was not invoked, the *naux* arguments were not updated, and the initialization step was performed for SCFT.

```
      SUBROUTINE SCFTQ (INIT, X, INC1X, INC2X, Y, INC1Y, INC2Y,
     *                  N, M, ISIGN, SCALE, AUX1, NAUX1,AUX2,NAUX2)
      REAL*4 X(0:*),Y(0:*),SCALE
      REAL*8 AUX1(7),AUX2(0:*)
      INTEGER*4 INIT,INC1X,INC2X,INC1Y,INC2Y,N,M,ISIGN,NAUX1,NAUX2
      EXTERNAL ENOTRM
      CHARACTER*8   S2015
          CALL EINFO(0)
          CALL ERRSAV(2015,S2015)
          CALL ERRSET(2015,0,-1,1,ENOTRM,0)
C     SETS NAUX1 AND NAUX2 TO THE MINIMUM VALUES REQUIRED TO USE
C     THE RECOVERABLE INPUT-ARGUMENT ERROR-HANDLING FACILITY
          NAUX1 = 7
          NAUX2 = 0
          CALL SCFT(INIT,X,INC1X,INC2X,Y,INC1Y,INC2Y,
     *              N,M,ISIGN,SCALE,AUX1,NAUX1,AUX2,NAUX2,*10)
          CALL ERRSTR(2015,S2015)
          RETURN
10        CONTINUE
          CALL ERRSTR(2015,S2015)
          RETURN 1
          END
```

Figure 3: Sample Fortran Subroutine to Calculate Auxiliary Storage Sizes in a 32-bit Integer, 64-bit Pointer Environment

Providing a Correct Transform Length to ESSL

This describes how to calculate the length of your transform by use of formulas or error-handling capabilities provided in ESSL.

For the ESSL subroutines listed in Table 38 on page 56, you need to provide one or more transform lengths for the computation of a Fourier transform. These transform lengths are supplied through one or more arguments, such as *n*, *n1*, *n2*, and *n3*, in the calling sequence for the ESSL subroutine. Only certain lengths of transforms are permitted in the computation.

Table 38: ESSL Subroutines Requiring Transform Lengths.

Subroutine Names
Fourier Transforms: _CFT _RCFT _CRFT _COSF _SINF SCOSFT _CFT2 _RCFT2 _CRFT2 _CFT3 _RCFT3 _CRFT3 SCFTP SCFT2P SCFT3P

Who Do You Want to Calculate the Transform Length? You or ESSL?

You have a choice of two methods for determining an acceptable length for your transform to be processed by ESSL:

- Use the formula or large table in "Acceptable Lengths for the Transforms" on page 956 to determine an acceptable length. For details, see "How Do You Calculate the Transform Length Using the Table or Formula?" on page 57.
- Use the ESSL error-handling facilities to return to you an acceptable length. For details, see "How Do You Get ESSL to Calculate the Transform Length Using ESSL Error Handling?" on page 57.

How Do You Calculate the Transform Length Using the Table or Formula?

The lengths ESSL accepts for transforms in the Fourier transform subroutines are listed in "Acceptable Lengths for the Transforms" on page 956. You should use the information in that table to find the two values your length falls between. You then specify the **larger** length for your transform. If you find a perfect match, you can use that value without having to change it. The formula provided expresses how to calculate the acceptable values listed in the table. If necessary, you can use the formula to dynamically check lengths in your program.

How Do You Get ESSL to Calculate the Transform Length Using ESSL Error Handling?

This describes how to get ESSL to calculate transform lengths. Ask yourself which of the following ways you prefer to obtain the information from ESSL:

- **By leaving error 2030 unrecoverable,** you can obtain an acceptable value for *n* from the input-argument error message, but your program terminates.
- **By making error 2030 recoverable,** you obtain an acceptable value for *n* from the input-argument error message and have the updated *n* argument returned to your program.

Because the Fourier transform subroutines allow only certain lengths for transforms, ESSL provides this error-handling capability to return acceptable lengths to your program. It returns them in the transform length arguments. The value ESSL returns is the **next larger acceptable length** for a transform, based on the length you specify in the *n* argument.

Having ESSL Calculate the Transform Length with Unrecoverable Error 2030

In this case, you obtain an acceptable value of *n* from the error message, but your program terminates.

Leave error 2030 as unrecoverable, without calls to EINFO and ERRSET. Run your program with a close approximation of the transform length you want to use. If this happens not to be an acceptable length, the ESSL error monitor returns an acceptable length of the transform in input-argument error message. This does, however, terminates your program when the error is encountered. (If you do happen to specify an acceptable length for the transform, error handling does not issue an error message and processing continues normally.) Figure 4 on page 57 illustrates what happens when error 2030 is unrecoverable.

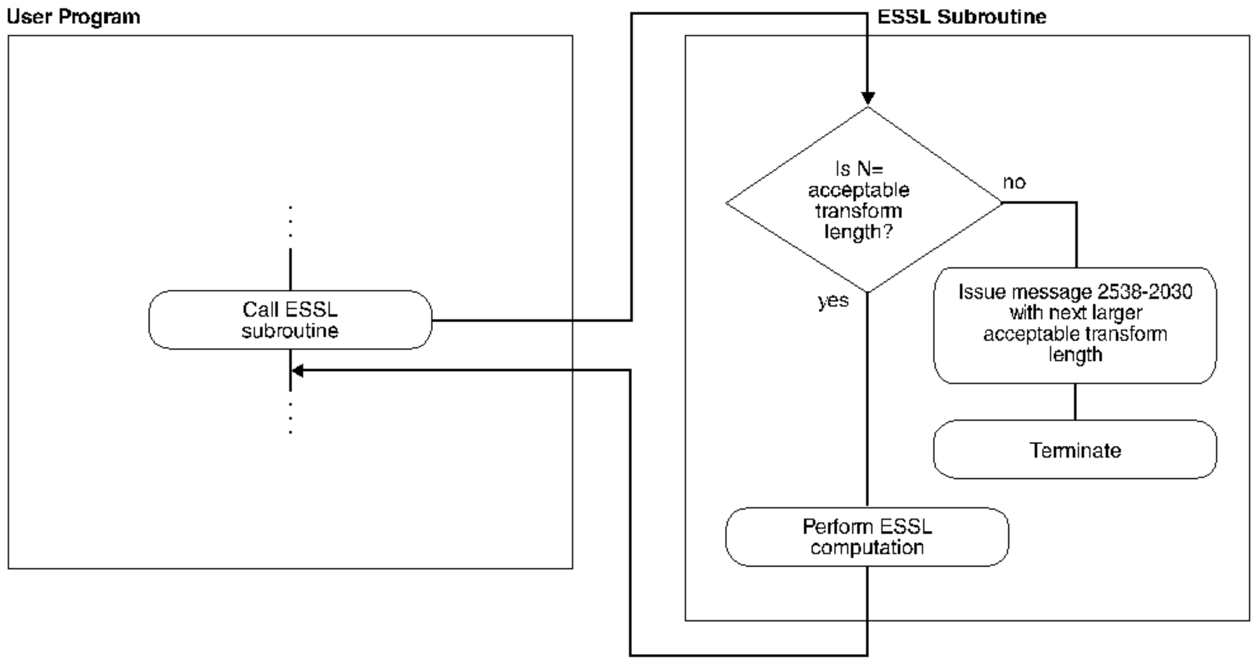

User Program
Call ESSL subroutine
ESSL Subroutine
Is N= acceptable transform length?
no
yes
Issue message 2538-2030 with next larger acceptable transform length
Terminate
Perform ESSL computation

Figure 4: How to Obtain an N Value from an Error Message, but Terminate

Having ESSL Calculate the Transform Length with Recoverable Error 2030

In this case, you obtain an acceptable value of *n* from the error message and from the updated *n* argument returned to your program.

Use EINFO and ERRSET with an ESSL error exit routine, ENOTRM, to make error 2030 recoverable. This allows you to dynamically determine in your program an acceptable length for your transform, specified in the *n* argument(s). Run your program with a close approximation of the transform length you want to use. If this happens not to be an acceptable length, the ESSL error monitor returns an acceptable length of the transform in the input-argument error message and a return code is passed back to your program, indicating that updated values are also returned in the *n* argument(s). You can then react to these updated values during run time in your program. ESSL does not perform any computation when this error occurs. For details on how to do this, see Chapter 4, "Coding programs," on page 125. (If you do happen to specify an acceptable length for the transform, error handling does not issue an error message and processing continues normally.) Figure 5 on page 58 illustrates what happens when error 2030 is recoverable.

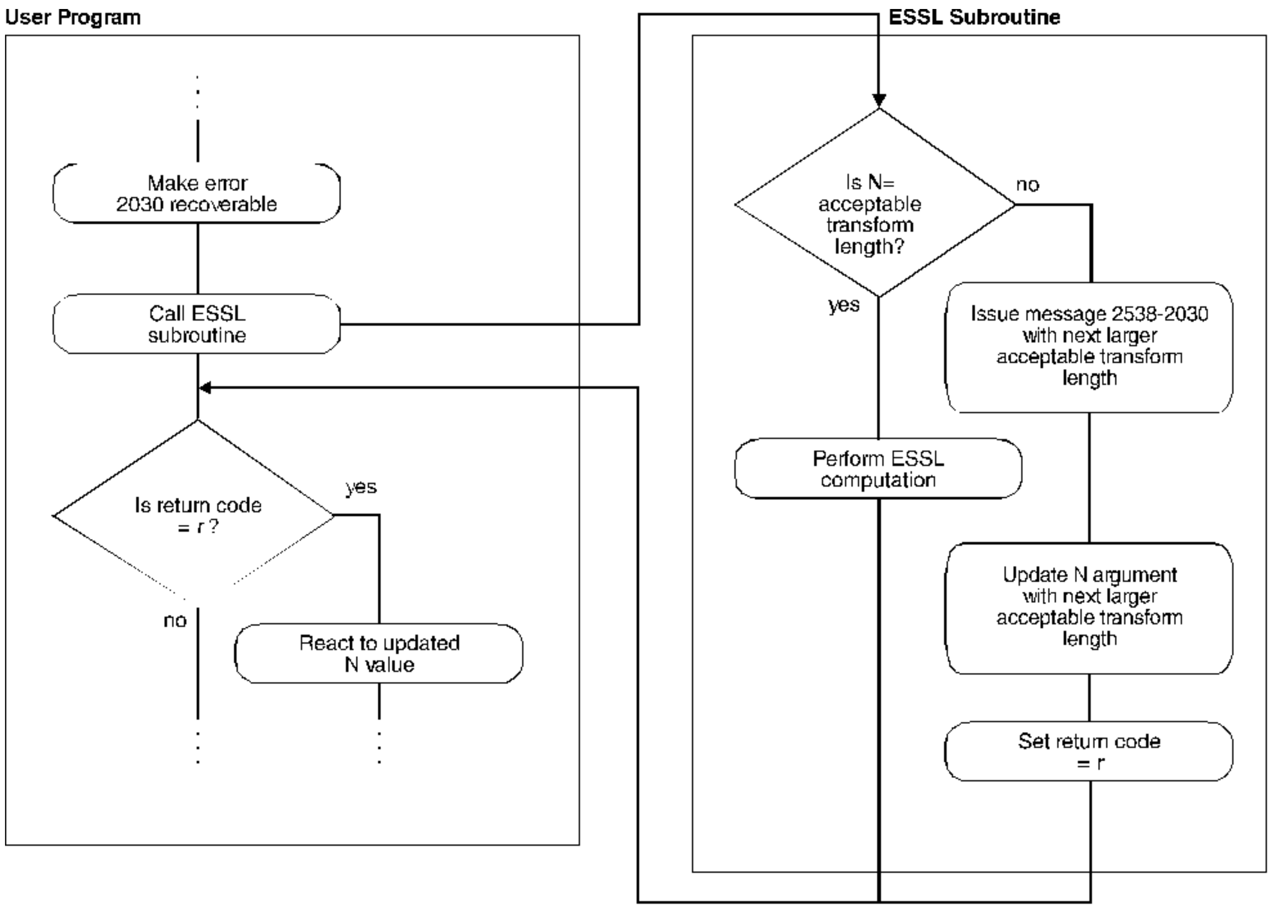

User Program
ESSL Subroutine
Make error 2030 recoverable
Call ESSL subroutine
Is return code = r ?
yes
no
React to updated N value
Is N= acceptable transform length?
no
yes
Issue message 2538-2030 with next larger acceptable transform length
Perform ESSL computation
Update N argument with next larger acceptable transform length
Set return code = r

Figure 5: How to Obtain an N Value from an Error Message and in Your Program

Example of Input-Argument Error Recovery for Transform Lengths

The following example illustrates all the actions taken by the ESSL error-handling facility for each possible value of a recoverable input argument, *n*. The values of *n* used in the example are as follows:

N	Meaning of the N Value
7208960	An acceptable transform length, required for successful computing of a Fourier transform
7340032	The next larger acceptable transform length, required for successful computing of a Fourier transform

Table 39 on page 59 describes the actions taken by ESSL in every possible situation for the values given in this example.

Table 39: Example of Input-Argument Error Recovery for Transform Lengths.

N Value	Action When 2030 Is an Unrecoverable Input-Argument Error	Action When 2030 Is a Recoverable Input-Argument Error
n = 7208960 **–or–** *n* = 7340032	Your application program runs successfully.	Your application program runs successfully.
7208960 < *n* < 7340032	An input-argument error message is issued. The value in the error message is 7340032. The application program stops.	ESSL returns the value of *n* as 7340032 to the application program, and an input-argument error message is issued. The value in the error message is 7340032. ESSL does no computation, and control is returned to the application program.

Coding Your Program to Obtain Transform Lengths

If you leave error 2030 unrecoverable, you **do not code anything** in your program. You just look at the error messages to get the transform lengths. On the other hand, if you want to make error 2030 recoverable to obtain the transform lengths dynamically in your program, you need to **add some coding statements** to your program. For details on coding these statements in each programming language, see the following examples:

- For Fortran, see "Input-Argument Errors in Fortran Example" on page 136.
- For C, see "Input-Argument Errors in C Example" on page 151.
- For C++, see "Input-Argument Errors in C++ Example" on page 164.

You may want to provide a separate subroutine to calculate the transform length whenever you need it. Figure 6 on page 60 shows how you might code a separate Fortran subroutine. Before calling SCFT in your program, you call this subroutine, SCFTQ, which calculates the correct length and stores it in *n*. Upon return, your program checks the return code. If it is nonzero, the *n* argument was updated, as planned. You then do any necessary data setup and call SCFT. On the other hand, if the return code is zero, error handling was not invoked, the *n* argument was not updated, and the initialization step was performed for SCFT.

```
      SUBROUTINE SCFTQ (INIT, X, INC1X, INC2X, Y, INC1Y, INC2Y,
     *                  N, M, ISIGN, SCALE, AUX1, NAUX1,AUX2,NAUX2)
      REAL*4 X(0:*),Y(0:*),SCALE
      REAL*8 AUX1(7),AUX2(0:*)
      INTEGER*4 INIT,INC1X,INC2X,INC1Y,INC2Y,N,M,ISIGN,NAUX1,NAUX2
      EXTERNAL ENOTRM
      CHARACTER*8   S2030
          CALL EINFO(0)
          CALL ERRSAV(2030,S2030)
          CALL ERRSET(2030,0,-1,1,ENOTRM,0)
          CALL SCFT(INIT,X,INC1X,INC2X,Y,INC1Y,INC2Y,
     *              N,M,ISIGN,SCALE,AUX1,NAUX1,AUX2,NAUX2,*10)
          CALL ERRSTR(2030,S2030)
          RETURN
10        CONTINUE
          CALL ERRSTR(2030,S2030)
          RETURN 1
          END
```

Figure 6: Sample Fortran Subroutine to Calculate Transform Length in a 32-bit Integer, 64-bit Pointer Environment

You might want to combine the request for auxiliary storage sizes along with your request for transform lengths. Figure 7 on page 60 shows how you might code a separate Fortran subroutine combining both requests. It combines the functions performed by the subroutine shown above and that shown in “Coding Your Program to Obtain Auxiliary Storage Sizes” on page 55.

```
      SUBROUTINE SCFTQ (INIT, X, INC1X, INC2X, Y, INC1Y, INC2Y,
     *                  N, M, ISIGN, SCALE, AUX1, NAUX1, AUX2, NAUX2)
      REAL*4 X(0:*),Y(0:*),SCALE
      REAL*8 AUX1(7),AUX2(0:*)
      INTEGER*4 INIT,INC1X,INC2X,INC1Y,INC2Y,N,M,ISIGN,NAUX1,NAUX2
      EXTERNAL ENOTRM
      CHARACTER*8   S2015,S2030
          CALL EINFO(0)
          CALL ERRSAV(2015,S2015)
          CALL ERRSAV(2030,S2030)
          CALL ERRSET(2015,0,-1,1,ENOTRM,0)
          CALL ERRSET(2030,0,-1,1,ENOTRM,0)
C     SETS NAUX1 AND NAUX2 TO THE MINIMUM VALUES REQUIRED TO USE
C     THE RECOVERABLE INPUT-ARGUMENT ERROR-HANDLING FACILITY
          NAUX1 = 7
          NAUX2 = 0
          CALL SCFT(INIT,X,INC1X,INC2X,Y,INC1Y,INC2Y,
     *              N,M,ISIGN,SCALE,AUX1,NAUX1,AUX2,NAUX2,*10)
          CALL ERRSTR(2015,S2015)
          CALL ERRSTR(2030,S2030)
          RETURN
10        CONTINUE
          CALL ERRSTR(2015,S2015)
          CALL ERRSTR(2030,S2030)
          RETURN 1
          END
```

Figure 7: Sample Fortran Subroutine to Calculate Auxiliary Storage Sizes and Transform Length in a 32-bit Integer, 64-bit Pointer Environment

Getting the Best Accuracy

This explains how accuracy of your results can be affected in various situations and what you can do to achieve the best possible accuracy.

What Precisions Do ESSL Subroutines Operate On?

Both short- and long-precision real versions of the subroutines are provided in most areas of ESSL. In some areas, short- and long-precision complex versions are also provided, and, occasionally, a 32-bit or 64-bit integer version is provided. The subroutine names are distinguished by a one- or two-letter prefix based on the following letters:

S for short-precision real
D for long-precision real
C for short-precision complex
Z for long-precision complex
I for integer

For a description of these data types, see "How Do You Set Up Your Scalar Data?" on page 47. The scalar data types and how you should code them for each programming language are listed under "Coding Your Scalar Data" specific to each programming language in Chapter 4, "Coding programs," on page 125.

How does the Nature of the ESSL Computation Affect Accuracy?

In subroutines performing operations such as copy and swap, the accuracy of data is not affected. In subroutines performing computations involving mathematical operations on array data, the accuracy of the result may be affected by the following:

- The algorithm, which can vary depending on values or array sizes within the computation or the number of threads used, or whether CPUs, GPUs, or both are used.
- The matrix and vector sizes

For this reason, the ESSL subroutines do **not** have a closed formula for the error of computation. In other words, there is no formula with which you can calculate the error of computation in each subroutine.

Many of the short-precision subprograms provide increased accuracy by accumulating results in long precision. However, when short-precision subroutines use the VSX unit to improve performance, they do not accumulate intermediate results in long precision. This is noted in the functional description of each subprogram.

Where applicable, the ESSL subroutines use the Multiply-Add instructions, which combine a Multiply and Add operation without an intermediate rounding operation.

The ESSL Serial Libraries and ESSL SMP Libraries allow you to run applications in any of the following environments, and results obtained in any of these environments using the same ESSL library are mathematically equivalent but may not be bitwise-identical:

- 32-bit integer, 64-bit pointer environment
- 64-bit integer, 64-bit pointer environment

What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?

The data types operated on by the short-precision, long-precision, and integer versions of the subroutines are ANSI/IEEE 32-bit and 64-bit binary floating-point format, and 32-bit and 64-bit integer. See the *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754–1985* for more detail.

There are ESSL-specific rules that apply to the results of computations using the ANSI/IEEE standards. When running your program, the result of a multiplication of NaN ("Not-a-Number") by a scalar zero, under certain circumstances, may differ in the ESSL subroutines from the result you expect.

Usually, when NaN is multiplied by a scalar zero, the result is NaN; however, in some ESSL subroutines where scaling is performed, the result may be zero. For example, in computing $\alpha\boldsymbol{A}$, where α is a scalar and $\boldsymbol{A}$ is a matrix, if α is zero and one (or more) of the elements of $\boldsymbol{A}$ is NaN, the scaled result, using that element, may be a zero, rather than NaN. To avoid problems, you should consider this when designing your program.

How is Underflow Handled?

ESSL does not mask underflow. If your program incurs a number of unmasked underflows, its overall performance decreases. Floating-point exception trapping is disabled by default. Therefore, you do not have to mask underflow unless you have changed the default.

Where Can You Find More Information on Accuracy?

Information about accuracy can be found in the following places:

- Migration considerations concerning accuracy of results between releases, platforms, and so forth are described in Chapter 6, “Migrating programs to ESSL,” on page 179.
- Specific information on accuracy for each area of ESSL is given in “Performance and Accuracy Considerations” associated with the subroutine descriptions for that area.
- The functional description under “Function” for each subroutine explains what you need to know about the accuracy of the computation. Varying implementation techniques are sometimes used to improve performance. To let you know how accuracy is affected, the functional description may explain in general terms the different techniques used in the computation.
- For details on accuracy considerations when using GPUs see:

```
https://developer.nvidia.com/content/precision-performance-floating-point-and-ieee-754-compliance-nvidia-gpus
```

What about Bitwise-Identical Results?

There are several circumstances where you may not get bitwise-identical results, although the results are mathematically equivalent:

- Results obtained on different hardware platforms
- Results obtained using different ESSL releases
- Results obtained using different ESSL Libraries
- Results obtained using a different number of threads
- Results obtained using arrays that are aligned differently. For example, the Power VSX unit require specific data alignments. If a subroutine uses one of these units and the input and/or output arrays are not aligned as required, some data may be processed using the floating point unit before or after the main SIMD loop.
- Results obtained using the ESSL SMP CUDA Library with environment variables ESSL_CUDA_HYBRID=yes and ESSL_CUDA_HYBRID=no. See “Using the ESSL SMP CUDA Library” on page 42.

Getting the Best Performance

This describes how you can achieve the best possible performance from the ESSL subroutines.

What General Coding Techniques Can You Use to Improve Performance?

There are many ways in which you can improve the performance of your program. Here are some of them:

- Use the basic linear algebra subprograms and matrix operations in the order of optimum performance: matrix-matrix computations, matrix-vector computations, and vector-scalar computations. When data is presented in matrices or vectors, rather than vectors or scalars, multiple operations can be performed by a single ESSL subroutine.
- Where possible, use subroutines that do multiple computations, such as SNDOT and SNAXPY, rather than individual computations, such as SDOT and SAXPY.
- Use a stride of 1 for the data in your computations. Not having vector elements consecutively accessed in storage can degrade your performance. The closer the vector elements are to each other in storage,

the better your performance. For an explanation of stride, see "How Stride Is Used for Vectors" on page 73.

- Do **not** specify the size of the leading dimension of an array (*lda*) or stride of a vector (*inc*) equal to or near a multiple of:
  - 128 for a long-precision array
  - 256 for a short-precision array
- On VSX enabled processors, specify the size of the leading dimension of a long or short-precision array as follows:
  - Long-precision real arrays - multiple of 2
  - Short-precision real arrays - multiple of 4
  - Short-precision complex arrays - multiple of 2

  Vectors and matrices are quadword aligned.
- Do **not** specify the individual sizes of your one-dimensional arrays as multiples of 128. This is especially important when you are passing several one-dimensional arrays to an ESSL subroutine. (The multiplicity can cause a performance problem that otherwise might not occur.)
- For small problems, avoid using a large leading dimension (*lda*) for your matrix.
- In general, align your arrays on doubleword boundaries, regardless of the type of data. For VSX enabled processors, see "SIMD Algorithms on VSX-Enabled Processors" on page 34. For information on how your programming language aligns data, see your programming language manuals.
- One subroutine may do scaling while another does not. If scaling is not necessary for your data, you get better performance by using the subroutine without scaling. SNORM2 and DNORM2 are examples of subroutines that do not do scaling, versus SNRM2 and DNRM2, which do scaling.
- Use the STRIDE subroutine to calculate the optimal stride values for your input or output data when using any of the Fourier transform subroutines, except _RCFT and _CRFT. Using these stride values for your data allows the Fourier transform subroutines to achieve maximum performance. You first obtain the optimal stride values from STRIDE, calling it once for each stride value desired. You then arrange your data using these stride values. After the data is set up, you call the Fourier transform subroutine. For details on the STRIDE subroutine and how to use it for each Fourier transform subroutine, see "STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)" on page 1195. For additional information, see "Setting Up Your Data" on page 959.
- >|When you use the ESSL SMP CUDA library subroutines SCFTD, DCFTD, SRCFTD, DRCFTD, SCRFTD, and DCRFTD, stride values from the STRIDE subroutine might negatively impact performance or might also prevent the computations that are running on the GPUs. |<
- If you are using the ESSL SMP CUDA library, performance might improve if you pin your host memory buffers. See "Using the ESSL SMP CUDA Library" on page 42.

Where Can You Find More Information on Performance?

Information about performance can be found in the following places:

- Many of the techniques ESSL uses to achieve the best possible performance are described in the "High Performance of ESSL" on page 4.
- Migration considerations concerning performance are described in Chapter 6, "Migrating programs to ESSL," on page 179.
- Specific information on performance for each area of ESSL is given in "Performance and Accuracy Considerations" for each grouping of subroutine descriptions.
- Detailed performance information for selected subroutines can be found in reference ["40" on page 1255], ["51" on page 1256], ["52" on page 1256].

Dealing with Errors when Using ESSL

At run time, you can encounter different types of errors or messages that are related to the use of the ESSL subroutines:

- Program exceptions
- ESSL input-argument errors
- ESSL computational errors
- ESSL resource errors
- ESSL attention messages

There are specific ways to handle all these situations.

What Can You Do about Program Exceptions?

The program exceptions you can encounter in ESSL are described in the *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754–1985*.

What Can You Do about ESSL Input-Argument Errors?

This gives an overview on how you can handle input-argument errors.

All Input-Argument Errors

ESSL checks the validity of most input arguments. If it finds that any are invalid, it issues the appropriate error messages. Also, except for the three recoverable errors described below, it terminates your program. You should use standard programming techniques to diagnose and fix unrecoverable input-argument errors, as described in Chapter 7, "Troubleshooting ESSL," on page 183.

You can determine the input-argument errors that can occur in a subroutine by looking under "Error Conditions" in each subroutine description. Error messages for all input-argument errors are listed in "Input-Argument Error Messages(2001-2099)" on page 187.

Recoverable Errors 2015, 2030 and 2200 Can Return Updated Values in the NAUX, N and NSINFO Arguments

For three input-argument errors, 2015, 2030, and 2200 in Fortran, C, and C++ programs, you have the option to continue running and have an updated value of the input argument returned to your program for subsequent use. These are called recoverable errors. This recoverable error-handling capability gives you flexibility in determining the correct values for the arguments. You can:

- Determine the correct size of an auxiliary work area by using error 2015. For help in deciding whether you want to use this capability and details on how to use it, see "Using Auxiliary Storage in ESSL" on page 49.
- Determine the correct length of a transform by using error 2030. For help in deciding whether you want to use this capability and details on how to use it, see "Providing a Correct Transform Length to ESSL" on page 56.
- Determine the minimal size of the array AP for DBSTRF and DBSSV by using error 2200. For help deciding whether you want to use this capability, see "DBSTRF (Symmetric Indefinite Matrix Factorization)" on page 613 and "DBSSV (Symmetric Indefinite Matrix Factorization and Multiple Right-Hand Side Solve)" on page 607

If you chose to leave errors 2015, 2030 and 2200 unrecoverable, you do not need to make any coding changes to your program. The input-argument error message is issued upon termination, containing the updated values you could have specified for the program to run successfully. You then make the necessary corrections in your program and rerun it.

If you choose to make errors 2015, 2030 and 2200 recoverable, you call the ERRSET subroutine to set up the ESSL error exit routine, ENOTRM, and then call the ESSL subroutine. When one or more of these errors occurs, the input-argument error message is issued with the updated values. In addition, the updated values are returned to your program in the input arguments named in the error message, along with a nonzero return code and processing continues. Return code values associated with these recoverable errors are described under "Error Conditions" for each ESSL subroutine in Part 2.

For details on how to code the necessary statements in your program to make 2015, 2030 and 2200 recoverable, see the following:

- "Input-Argument Errors in Fortran" on page 134
- "Input-Argument Errors in C" on page 147
- "Input-Argument Errors in C++" on page 161

What Can You Do about ESSL Computational Errors?

This gives an overview on how you can handle computational errors.

All Computational Errors

ESSL computational errors are errors occurring in the computational data, such as in your vectors and matrices. You can determine the computational errors that can occur in a subroutine by looking under "Error Conditions" in each subroutine description. These errors cause your program to terminate abnormally unless you take preventive action. A message is also provided in your output, containing information about the error. Messages are listed in "Computational Error Messages(2100-2199)" on page 191.

When a computational error occurs, you should assume that the results are unpredictable. The result of the computation is valid only if no errors have occurred. In this case, a zero return code is returned.

Figure 8 on page 65 shows what happens when a computational error occurs.

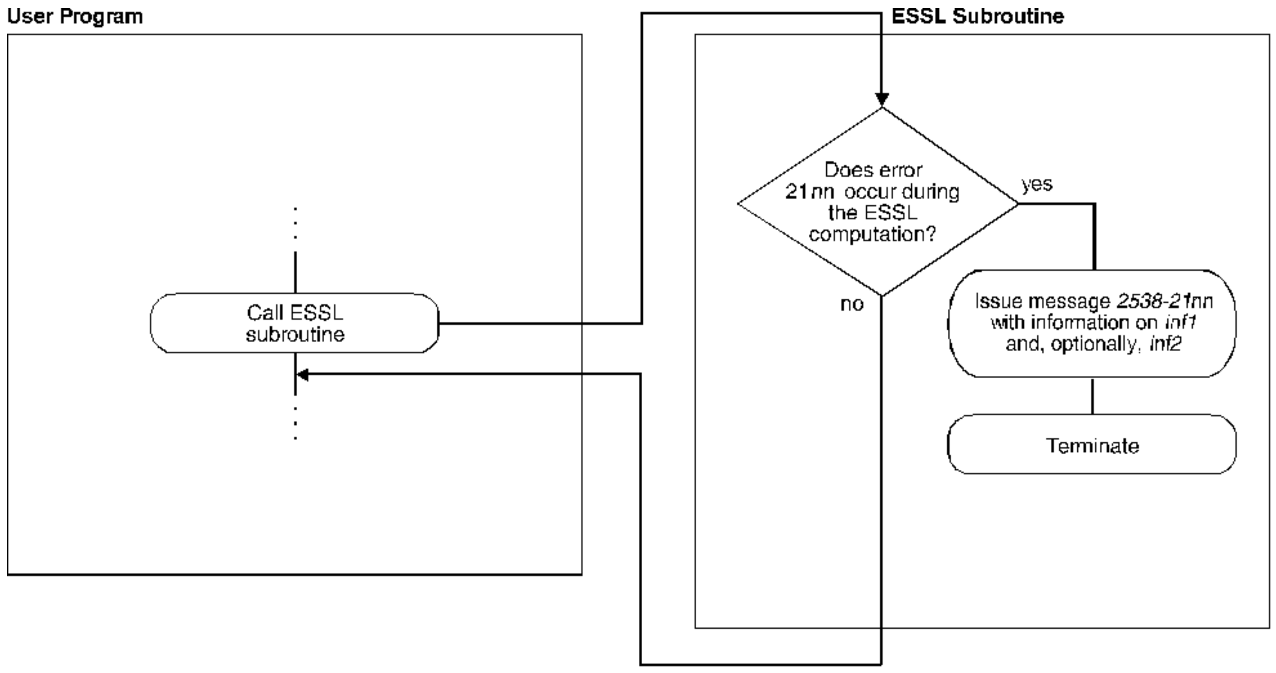

User Program
ESSL Subroutine
Call ESSL subroutine
Does error 21*nn* occur during the ESSL computation?
yes
no
Issue message *2538-21nn* with information on *inf1* and, optionally, *inf2*
Terminate

Figure 8: How to Obtain Computational Error Information from an Error Message, but Terminate

Recoverable Computational Errors Can Return Values Through EINFO

In Fortran, C, and C++ programs, you have the capability to make certain computational errors recoverable and have information returned to your program about the errors. Recoverable computational errors are listed in "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188. First, you call EINFO in the beginning of your program to initialize the ESSL error option table. You then call ERRSET to reset the number of allowable errors for the computational error codes in which you are interested. When a computational error occurs, a nonzero return code is returned for each computational error. Return code values associated with these errors are described under "Error Conditions" in each subroutine description. Based on the return code, your program can branch to an appropriate statement to call the ESSL error information-handler subroutine, EINFO, to obtain specific information about the data involved in the error. This information is returned in the EINFO output arguments, *inf1* and, optionally, *inf2*. You can then check the information returned and continue processing, if you choose. The syntax for EINFO is described under "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188. You also get a message in your output for each computational error encountered, containing information about the error. The EINFO subroutine provides the same information in the messages as it provides to your program.

For details on how to code the necessary statements in your program to obtain specific information on computational errors, see the following:

- "Computational Errors in Fortran" on page 136
- "Computational Errors in C" on page 151
- "Computational Errors in C++" on page 165

Figure 9 on page 66 shows what happens if you make a computational error recoverable.

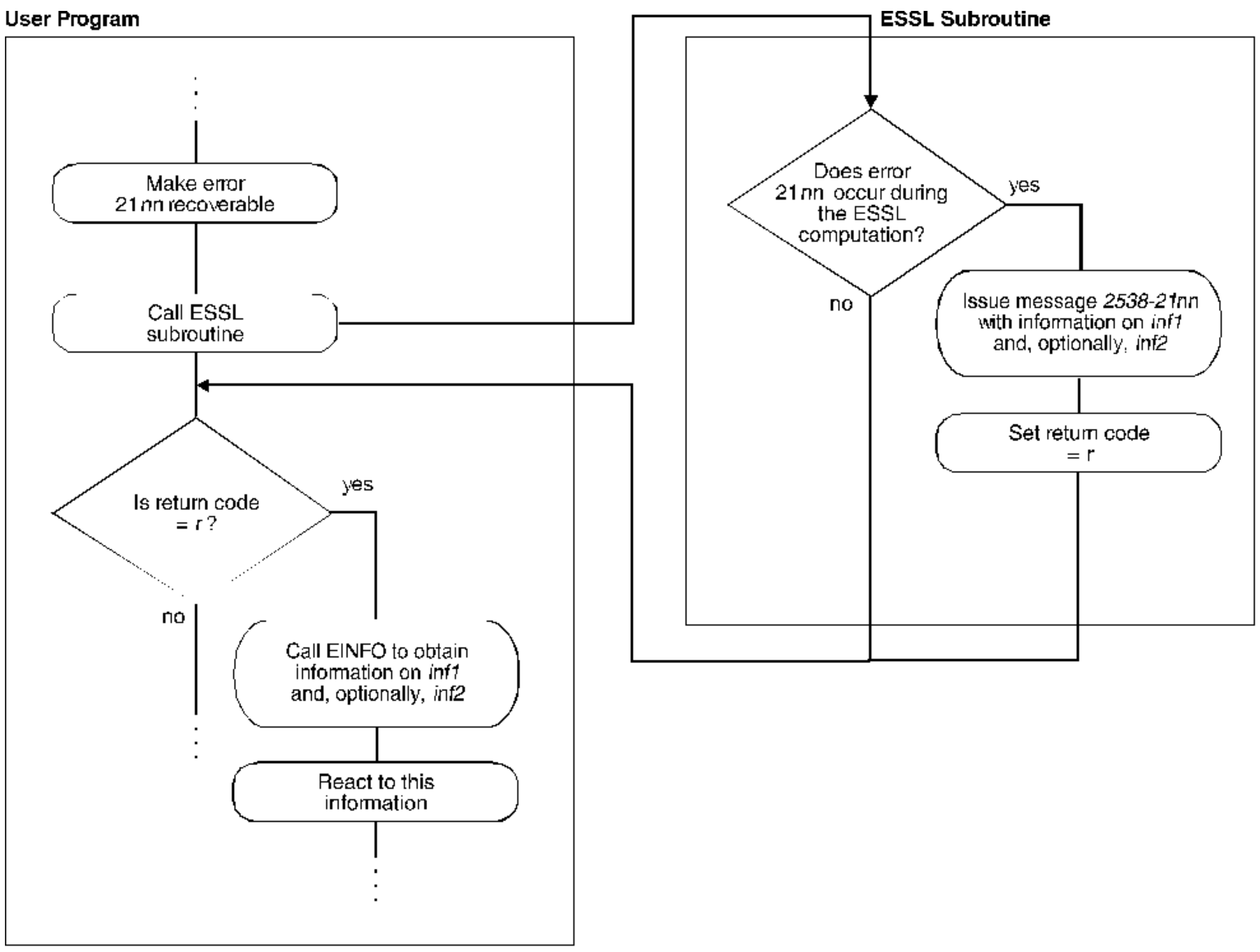

User Program
ESSL Subroutine
Make error 21nn recoverable
Call ESSL subroutine
Is return code = r ?
yes
no
Call EINFO to obtain information on inf1 and, optionally, inf2
React to this information
Does error 21nn occur during the ESSL computation?
yes
no
Issue message 2538-21nn with information on inf1 and, optionally, inf2
Set return code = r

Figure 9: How to Obtain Computational Error Information in an Error Message and in Your Program

What Can You Do about ESSL Resource Errors?

This gives an overview on how you can handle resource errors.

All Resource Errors

ESSL returns a resource error and terminates your program when an attempt to allocate work area fails. Some ESSL subroutines attempt to allocate work area for their internal use. Other ESSL subroutines attempt to dynamically allocate auxiliary storage when a user requests it through calling sequence arguments, such as *aux* and *naux*. For information on how you could reduce memory constraints on the system or increase the amount of memory available before rerunning the application program, see "ESSL Resource Error Messages" on page 185.

You can determine the resource errors that can occur in a subroutine by looking under "Error Conditions" in each subroutine description. Error messages for all resource errors are listed in "Resource Error Messages(2400-2499)" on page 196.

What Can You Do about ESSL Attention Messages?

This gives an overview on how you can handle attention messages.

All Attention Messages

ESSL returns an attention message to describe a condition that occurred, however, ESSL is able to continue processing. For information on how you could reduce memory constraints on the system or increase the amount of memory available, see "ESSL Resource Error Messages" on page 185.

For example, an attention message may be issued when enough work area was available to continue processing, but was not the amount initially requested. An attention message would be issued to indicate that performance may be degraded.

For a list of attention messages, see "Informational and Attention Error Messages(2600-2699)" on page 196.

How Do You Control Error Handling by Setting Values in the ESSL Error Option Table?

This explains all aspects of using the ESSL error option table.

What Values Are Set in the ESSL Error Option Table?

The ESSL error option table contains information that tells ESSL what to do every time it encounters an ESSL-generated error. Table 40 on page 67 shows the default values established in the table when ESSL is installed.

Table 40: ESSL Error Option Table Default Values.

Range of Error Messages (From–To)	Number of Allowable Errors (ALLOW)	Number of Messages Printed (PRINT)	Modifiable Table Entry (MODENT)
2538–2000	Unlimited	255	NO
2538–2001 through 2538–2073	Unlimited	255	YES
2538–2074	Unlimited	5	YES
2538–2075 through 2538–2098	Unlimited	255	YES
2538–2099	1	255	YES
2538–2100 through 2538–2101	1	255	YES
2538–2102	Unlimited	255	YES
2538–2103 through 2538–2113	1	255	YES
2538–2114	Unlimited	255	YES
2538–2115 through 2538–2122	1	255	YES
2538–2123 through 2538–2124	Unlimited	255	YES
2538–2125 through 2538–2126	1	255	YES

Table 40: ESSL Error Option Table Default Values. (continued)

Range of Error Messages (From–To)	Number of Allowable Errors (ALLOW)	Number of Messages Printed (PRINT)	Modifiable Table Entry (MODENT)
2538–2127	Unlimited	255	YES
2538–2128 through 2538–2137	1	255	YES
2538–2138 through 2538–2143	Unlimited	255	YES
2538–2144 through 2538–2145	1	255	YES
2538–2146 through 2538–2149	Unlimited	255	YES
2538–2150	1	255	YES
2538–2151 through 2538–2166	Unlimited	255	YES
2538–2167 through 2538–2198	1	255	YES
2538–2199	1	255	YES
2538–2200 through 2538–2299	Unlimited	255	YES
2538–2400 through 2538–2499	1	255	NO
2538–2600 through 2538–2609	Unlimited	255	NO
2538–2610 through 2538–2612	Unlimited	-1	YES
2538–2613 through 2538–2613	Unlimited	255	NO
2538–2614 through 2538–2615	Unlimited	1	NO
2538–2616 through 2538–2699	Unlimited	255	NO
2538–2700 through 2538–2799	1	255	NO

How Can You Change the Values in the Error Option Table?

You can change any of the values in the ESSL error option table by calling the ERRSET subroutine in your program. This dynamically changes values at run time. You can also save and restore entries in the table by using the ERRSAV and ERRSTR subroutines, respectively. For a description of the ERRSET, ERRSAV, and ERRSTR subroutines see Chapter 17, “Utilities subroutines,” on page 1187.

When Do You Change the Values in the Error Option Table?

Because you can change the information in the error option table, you can control what happens when any of the ESSL errors occur. There are a number of instances when you may want to do this:

To Customize Your Error-Handling Environment

You may simply want to adjust the number of times an error is allowed to occur before your program terminates. You can use any of the capabilities available in ERRSET.

To Obtain Auxiliary Storage Sizes and Transform Lengths

You may want to make ESSL input-argument error 2015 or 2030 recoverable, so ESSL returns updated auxiliary storage sizes or transform lengths, respectively, to your program. For a more detailed discussion, see “What Can You Do about ESSL Input-Argument Errors?” on page 64. For how to use ERRSET to do this, see the information specific to your programming language in Chapter 4, “Coding programs,” on page 125.

To Obtain the Minimal Size of the Array AP for DBSTRF and DBSSV

You may want to make ESSL input-argument error 2200 recoverable, so ESSL returns an updated size to your program. For a more detailed discussion, see "What Can You Do about ESSL Input-Argument Errors?" on page 64. For how to use ERRSET to do this, see the information specific to your programming language in Chapter 4, "Coding programs," on page 125.

To Get More Information About a Computational Error

You may want ESSL to return information about a computational error to your program. For a more detailed discussion, see "What Can You Do about ESSL Computational Errors?" on page 65. For how to do use ERRSET to do this, see the information specific to your programming language in Chapter 4, "Coding programs," on page 125.

To Allow Parts of Your Application to Have Unique Error-Handling Environments

If your program is part of a large application, you may want to dynamically save and restore entries in the error option table that have been altered by ERRSET. This ensures the integrity of the error option table when it is used by multiple programs within an application. For a more detailed discussion, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 69 For how to use ERRSAV and ERRSTR, see the information specific to your programming language in Chapter 4, "Coding programs," on page 125.

How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?

When your program is part of a larger application, you should consider that one of the following can occur:

- If you use ERRSET in your program to reset any of the values in the error option table for any of the ESSL input-argument errors or computational errors, some other program in the application may be adversely affected. It may be expecting its original values.
- If some other program in the application uses ERRSET to reset any of the values in the error option table for any of the ESSL input-argument errors or computational errors, your program may be adversely affected. You may need a certain value in the error option table, and the application may have reset that value.

These situations can be avoided if every program that uses ERRSET, in the large application, also uses the ERRSAV and ERRSTR facilities. For a particular error number, ERRSAV saves an entry from the error option table in an area accessible to your program. ERRSTR then stores the entry back into the error option table from the storage area. You code an ERRSAV and ERRSTR for each input-argument error number and computational error number for which you do an ERRSET to reset the values in the error option table. Call ERRSAV at the beginning of your program after you call EINFO, and then call ERRSTR at the end of your program after all ESSL computations are completed. This saves the original contents of the error option table while your program is running with different values, and then restores it to its original contents when your program is done. For details on how to code these statements in your program, see Chapter 4, "Coding programs," on page 125.

How does Error Handling Work in a Threaded Environment?

When your application program or the open MP library first creates a thread, ESSL initializes the error option table information to the default settings shown in "What Values Are Set in the ESSL Error Option Table?" on page 67. You can change the default settings for each thread you created by calling the appropriate error handling subroutines (ERRSET, ERRSAV, or ERRSTR) from each thread. An example of how to initialize the error option table and change the default settings on multiple threads is shown in "Example of Handling Errors in a Multithreaded Application Program " on page 141.

ESSL issues error messages as they occur in a threaded environment. Error messages issued from any of the existing threads are written to standard output in the order in which they occur.

When a terminating condition occurs on any of the existing threads (for example, the number of allowable errors was exceeded), ESSL terminates your application program. One set of summary information corresponding to the terminating thread is always printed. Summary information corresponding to other threads may also be printed.

Where Can You Find More Information on Errors?

Information about errors and how to handle them can be found in the following places:

- How to code your program to use the ESSL error-handling facilities is described in Chapter 4, “Coding programs,” on page 125.
- All ESSL error messages are listed under “Messages” on page 186.
- The errors and return codes associated with each ESSL subroutine are listed under “Error Conditions” in each subroutine description.
- Complete diagnostic procedures for all types of ESSL programming and documentation problems, along with how to collect information and report a problem, are provided in Chapter 7, “Troubleshooting ESSL,” on page 183.

Chapter 3. Setting up data structures

This provides you with information that you need to configure your data structures, consisting of vectors, matrices, and sequences. These techniques apply to programs in all programming languages.

Vector concepts

Vectors, matrices, and sequences are conceptual data structures contained in arrays. In many cases, ESSL uses stride or leading dimension to select the elements of the vector, matrix, or sequence from an array. In other cases, ESSL uses a specific mapping, or storage layout, that identifies the elements of the vector, matrix, or sequence in an array, sometimes requiring several arrays to help define the mapping. These elements selected from the array(s) make up the conceptual data structure.

When you call an ESSL subroutine, it assumes that the data structure is set up properly in the array(s) you pass to it. If it is not, your results are unpredictable. ESSL also uses these same storage layouts for data structures passed back to your program.

The use of the terms vector, matrix, and sequence here is consistent with standard mathematical definitions, and their representations are consistent with conventions used in mathematical texts.

Overlapping Data Structures: Most of the subroutines do not allow vectors, matrices, or sequences to overlap. If this occurs, results are unpredictable. This means the elements of the data structure cannot reside in the same storage locations as any of the other data structures. It is possible, however, to have elements of different data structures in the same array, as long as the elements are interleaved through storage using strides greater than 1. For example, using vectors ***x*** and ***y*** with strides of 2, where ***x*** starts at `A(1)` and ***y*** starts at `A(2)`, the elements reside in array A in the order x_1, y_1, x_2, y_2, x_3, y_3, ... and so forth.

When you use this technique, you should be careful that you specify different starting locations for each data structure contained in the array.

Vectors

A vector is a one-dimensional, ordered collection of numbers. It can be a column vector, which represents an *n* by 1 ordered collection, or a row vector, which represents a 1 by *n* ordered collection.

The column vector appears symbolically as follows:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

A row vector appears symbolically as follows:

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 & \dots & x_n \end{bmatrix}$$

Vectors can contain either real or complex numbers. When they contain real numbers, they are sometimes called real vectors. When they contain complex numbers, they are called complex vectors.

Transpose of a Vector

The transpose of a vector changes a column vector to a row vector, or vice versa:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{bmatrix} \quad \boldsymbol{x}^{\mathrm{T}} = \begin{bmatrix} x_1 & x_2 & x_3 \ldots x_n \end{bmatrix} \quad (\boldsymbol{x}^{\mathrm{T}})^{\mathrm{T}} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

The ESSL subroutines use the vector as it is intended in the computation, as either a column vector or a row vector; therefore, no movement of data is necessary.

In the examples provided with the subroutine descriptions, both types of vectors are represented in the same way, showing the elements of the array that make up the vector ***x***, as follows:

```
                         (1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
```

Conjugate Transpose of a Vector

The conjugate transpose of a vector ***x***, containing complex numbers, is denoted by $\boldsymbol{x}^{\mathrm{H}}$ and is expressed as follows:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{bmatrix} \quad \boldsymbol{x}^{\mathrm{H}} = \begin{bmatrix} \bar{x}_1 & \bar{x}_2 & \bar{x}_3 \ldots \bar{x}_n \end{bmatrix}$$

Just as for the transpose of a vector, no movement of data is necessary for the conjugate transpose of a vector.

Vector Storage Representation

A vector is usually stored within a one- or two-dimensional array. Its elements are stored sequentially in the array, but not necessarily contiguously.

The **location** of the vector in the array is specified by the argument for the vector in the ESSL calling sequence. It can be specified in a number of ways. For example, if A is an array of length 12, and you want to specify vector ***x*** as starting at the first element of array A, specify A as the argument, such as in:

```
    X = SASUM (4,A,2)
```

where the number of elements to be summed in the vector is 4, the location of the vector is A, and the stride is 2.

If you want to specify vector ***x*** as starting at element 3 in array A, which is declared as `A(1:12)`, specify:

```
   X = SASUM (4,A(3),2)
```

If A is declared as `A(-1:8)`, specify the following for element 3:

```
   X = SASUM (4,A(1),2)
```

If A is a two-dimensional array and declared as `A(1:4,1:10)`, and you want vector ***x*** to start at the second row and third column of A, specify the following:

```
   X = SASUM (4,A(2,3),2)
```

The **stride** specified in the ESSL calling sequence is used to step through the array to select the vector elements. The direction in which the vector elements are selected from the array—that is, front to back or back to front—is indicated by the sign (+ or -) of the stride. The absolute value of the stride gives the spacing between each element selected from the array.

To calculate the total number of elements needed in an array for a vector, you can use the following formula, which takes into account the number of elements, *n*, in the array and the stride, *inc*, specified for the vector:

$1+(n-1)|inc|$

An array can be much larger than the vector that it contains; that is, there can be many elements following the vector in the array, as well as elements preceding the vector.

For a complete description of how vectors are stored within arrays, see "How Stride Is Used for Vectors" on page 73.

For a complex vector, a special storage arrangement is used to accommodate the two parts, *a* and *b*, of each complex number (*a*+*bi*) in the array. For each complex number, two sequential storage locations are required in the array. Therefore, exactly twice as much storage is required for complex vectors and matrices as for real vectors and matrices of the same precision. See "How Do You Set Up Your Scalar Data?" on page 47 for a description of real and complex numbers, and "How Do You Set Up Your Arrays?" on page 47 for a description of how real and complex data is stored in arrays.

How Stride Is Used for Vectors

The stride for a vector is an increment that is used to step through array storage to select the vector elements from an array. To define exactly which elements become the conceptual vector in the array, the following items are used together:

- The location of the vector within the array
- The stride for the vector
- The number of elements, *n*, to be processed

The stride can be positive, negative, or 0. For positive and negative strides, if you specify vector elements beyond the range of the array, your results are be unpredictable, and you may get program errors.

This explains how each of the three types of stride is used to select the vector elements from the array.

Positive Stride

When a positive stride is specified for a vector, the location specified by the argument for the vector is the location of the first element in the vector, element x_1. The vector is in forward order in the array: (x_1, x_2, ..., x_n). For example, if you specify `X(1)` for vector ***x***, where X is declared as `X(0:12)` and defined as:

```
X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0)
```

then processing begins at the second element in X, which is 2.0.

To find each successive element, the stride is added cumulatively to the starting point of vector ***x*** in the array. In this case, the starting point is `X(1)`. If the stride specified for vector ***x*** is 3 and the number of elements to be processed is 4, then the resulting elements selected from X for vector ***x*** are: `X(1),X(4),X(7),` and `X(10)`.

Vector ***x*** is then:

```
                     (2.0, 5.0, 8.0, 11.0)
```

As shown in this example, a vector does not have to extend to the end of the array. Elements are selected from the second to the eleventh element of the array, and the array elements after that are not used.

This element selection can be expressed in general terms. Using BEGIN as the starting point in an array X and *inc* as the stride, this results in the following elements being selected from the array:

X(BEGIN)
X(BEGIN+*inc*)
X(BEGIN+(2)*inc*)
X(BEGIN+(3)*inc*)
.
.
.
X(BEGIN+(*n*-1)*inc*)

The following general formula can be used to calculate each vector element position in a one-dimensional array:

x_i = X(BEGIN + (*i*-1)(*inc*)) for *i* = 1, *n*

When using an array with more than one dimension, you should understand how the array elements are stored to ensure that elements are selected properly. For a description of array storage, see "Setting Up Arrays in Fortran" on page 126. You should remember that the elements of an array are selected as they are arranged in storage, regardless of the number of dimensions defined in the array. Stride is used to step through array storage until *n* elements are selected. ESSL processing stops at that point. For example, given the following two-dimensional array, declared as `A(1:7,1:4)`.

Matrix A is:

```
                     | 1.0   8.0  15.0  22.0 |
                     | 2.0   9.0  16.0  23.0 |
                     | 3.0  10.0  17.0  24.0 |
                     | 4.0  11.0  18.0  25.0 |
                     | 5.0  12.0  19.0  26.0 |
                     | 6.0  13.0  20.0  27.0 |
                     | 7.0  14.0  21.0  28.0 |
```

with `A(3,1)` specified for vector ***x***, a stride of 2, and the number of elements to be processed as 12, the resulting vector ***x*** is:

```
    (3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0, 17.0, 19.0, 21.0, 23.0, 25.0)
```

This is not a conventional use of arrays, and you should be very careful when using this technique.

Zero Stride

When a zero stride is specified for a vector, the starting point for the vector is the only element used in the computation. The starting point for the vector is at the location specified by the argument for the vector, just as though you had specified a positive stride. For example, if you specify X for vector ***x***, where X is defined as:

```
                     X = (5.0, 4.0, 3.0, 2.0, 1.0)
```

and you specify the number of elements, *n*, to be processed as 6, then processing begins at the first element, which is 5.0. This element is used for each of the six elements in vector ***x***.

This makes the conceptual vector ***x*** appear as:

```
                    (5.0, 5.0, 5.0, 5.0, 5.0, 5.0)
```

The following general formula shows how to calculate each vector position in a one-dimensional array:

x_i = X(BEGIN) for *i* = 1, *n*

Negative Stride

When a negative stride is specified for a vector, the location specified for the vector is actually the location of the last element in the vector. In other words, the vector is in reverse order in the array: (x_n, x_{n-1}, ..., x_1). You specify the end of the vector, (x_n). ESSL then calculates where the starting point (x_1) is by using the following arguments:

- The location of the vector in the array
- The stride for the vector, *inc*
- The number of elements, *n*, to be processed

If you specify vector ***x*** at location `X(BEGIN)` in array X with a negative stride of *inc* and *n* elements to be processed, then the following formula gives the starting point of vector ***x*** in the array:

X(BEGIN + (-*n*+1)(*inc*))

For example, if you specify `X(2)` for vector ***x***, where X is declared as `X(1:9)` and defined as:

```
               X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0)
```

and if you specify a stride of -2, and four elements to be processed, processing begins at the following element in `X`:

X(2+(-4+1)(-2)) = X(8)

where element `X(8)` is 8.0.

To find each of the *n* successive element positions in the array, you successively add the stride to the starting point *n*-1 times. Suppose the formula calculated a starting point of `X(SP)`; the elements selected are:

X(SP)
X(SP+*inc*)
X(SP+(2)*inc*)
X(SP+(3)*inc*)
.
.
.
X(SP+(*n*-1)*inc*)

In the above example, the resulting elements selected from X for vector ***x*** are `X(8), X(6), X(4)`, and `X(2)`. This makes the resulting vector ***x*** appear as follows:

```
                    (8.0, 6.0, 4.0, 2.0)
```

The following general formula can be used to calculate each vector element position in a one-dimensional array:

x_i = X(BEGIN + (-*n*+*i*)(*inc*)) for *i* = 1, *n*

Sparse Vector

A sparse vector is a vector having a relatively small number of nonzero elements.

Consider the following as an example of a sparse vector ***x*** with *n* elements, where *n* is 11, and vector ***x*** is:

```
              (0.0, 0.0, 1.0, 0.0, 2.0, 3.0, 0.0, 4.0, 0.0, 5.0, 0.0)
```

In Storage

There are two storage modes that apply to sparse vectors: full-vector storage mode and compressed-vector storage mode. When a sparse vector is stored in **full-vector storage mode**, all its elements, including its zero elements, are stored in an array.

For example, sparse vector ***x*** is stored in full-vector storage mode in a one-dimensional array X, as follows:

```
       X  = (0.0, 0.0, 1.0, 0.0, 2.0, 3.0, 0.0, 4.0, 0.0, 5.0, 0.0)
```

When a sparse vector is stored in **compressed-vector storage mode**, it is stored without its zero elements. It consists of two one-dimensional arrays, each with a length of *nz*, where *nz* is the number of nonzero elements in vector ***x***:

- The first array contains the nonzero elements of the sparse vector ***x***, stored contiguously within the array.

  Note: The ESSL subroutines do not check that all elements are nonzero. You do not get an error if any elements are zero.
- The second array contains a sequence of integers indicating the element positions (indices) of the nonzero elements of the sparse vector ***x*** stored in full-vector storage mode. This is referred to as the indices array.

For example, the sparse vector ***x*** shown above might have its five nonzero elements stored in ascending order in array X of length 5, as follows:

```
                    X = (1.0, 2.0, 3.0, 4.0, 5.0)
```

in which case, the array of indices, `INDX`, also of length 5, contains:

```
                     INDX = (3, 5, 6, 8, 10)
```

If the sparse vector ***x*** has its elements stored in random order in the array X as:

```
                   X  = (5.0, 3.0, 4.0, 1.0, 2.0)
```

then the array `INDX` contains:

```
                       INDX = (10, 6, 8, 3, 5)
```

In general terms, this storage technique can be expressed as follows:

For each $x_j \neq 0$, for $j = 1, n$
there exists i, where $1 \leq i \leq nz$,
such that $X(i) = x_j$ and $\texttt{INDX}(i) = j$.

where:

$x_1, ..., x_n$ are the *n* elements of sparse vector ***x***, stored in full-vector storage mode.

X is the array containing the *nz* nonzero elements of sparse vector ***x***; that is, vector ***x*** is stored in compressed-vector storage mode.

`INDX` is the array containing the *nz* indices indicating the element positions.

To avoid an error when using the `INDX` array to access the elements in any other target vector, the length of the target vector must be greater than or equal to max(`INDX`(*i*)) for *i* = 1, *nz*.

Matrices

A matrix, also referred to as a general matrix, is an *m* by *n* ordered collection of numbers. It is represented symbolically as:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & . & . & . & a_{1n} \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ a_{m1} & . & . & . & a_{mn} \end{bmatrix}$$

where the matrix is named ***A*** and has *m* rows and *n* columns. The elements of the matrix are a_{ij}, where i = 1, m and j = 1, n.

Matrices can contain either real or complex numbers. Those containing real numbers are called real matrices; those containing complex numbers are called complex matrices.

Transpose of a Matrix

The transpose of a matrix ***A*** is a matrix formed from ***A*** by interchanging the rows and columns such that row *i* of matrix ***A*** becomes column *i* of the transposed matrix. The transpose of ***A*** is denoted by $\boldsymbol{A}^{\mathrm{T}}$. Each element a_{ij} in ***A*** becomes element a_{ji} in $\boldsymbol{A}^{\mathrm{T}}$. If ***A*** is an *m* by *n* matrix, then $\boldsymbol{A}^{\mathrm{T}}$ is an *n* by *m* matrix. The following represents a matrix and its transpose:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & . & . & . & a_{1n} \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ a_{m1} & . & . & . & a_{mn} \end{bmatrix} \qquad \boldsymbol{A}^{\mathrm{T}} = \begin{bmatrix} a_{11} & . & . & . & a_{m1} \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ a_{1n} & . & . & . & a_{mn} \end{bmatrix}$$

ESSL assumes that all matrices are stored in untransformed format, such as matrix ***A*** shown above. No movement of data is necessary in your application program when you are processing transposed matrices. The ESSL subroutines adjust their selection of elements from the matrix when an argument in the calling sequence indicates that the transposed matrix is to be used in the computation. Examples of this are the *transa* and *transb* arguments specified for SGEADD, matrix addition.

Conjugate Transpose of a Matrix

The conjugate transpose of a matrix ***A***, containing complex numbers, is denoted by $\boldsymbol{A}^{\mathrm{H}}$ and is expressed as follows:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & . & . & . & a_{1n} \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ a_{m1} & . & . & . & a_{mn} \end{bmatrix} \qquad \boldsymbol{A}^{\mathrm{H}} = \begin{bmatrix} \bar{a}_{11} & . & . & . & \bar{a}_{m1} \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ \bar{a}_{1n} & . & . & . & \bar{a}_{mn} \end{bmatrix}$$

Just as for the transpose of a matrix, the conjugate transpose of a matrix is stored in untransformed format. No movement of data is necessary in your program.

Matrix Storage Representation

A matrix is usually stored in a two-dimensional array. Its elements are stored successively within the array. Each column of the matrix is stored successively in the array. The leading dimension argument is used to select the matrix elements from each successive column of the array. The starting point of the matrix in the array is specified as the argument for the matrix in the ESSL calling sequence. For example, if matrix ***A*** is contained in array A and starts at the first element in the first row and first column of A, you should specify A as the argument for matrix ***A***, such as in:

```
   CALL  SGEMX (5,2,1.0,A,6,X,1,Y,1)
```

where, in the matrix-vector product, the number of rows in matrix ***A*** is 5, the number of columns in matrix ***A*** is 2, the scaling constant is 1.0, the location of the matrix is A, the leading dimension is 6, the vectors used in the matrix-vector product are X and Y, and their strides are 1.

If matrix ***A*** is contained in the array `BIG`, declared as `BIG(1:20,1:30)`, and starts at the second row and third column of `BIG`, you should specify `BIG(2,3)` as the argument for matrix ***A***, such as in:

```
   CALL  SGEMX (5,2,1.0,BIG(2,3),6,X,1,Y,1)
```

See "How Leading Dimension Is Used for Matrices" on page 78 for a complete description of how matrices are stored within arrays.

For a complex matrix, a special storage arrangement is used to accommodate the two parts, *a* and *b*, of each complex number (*a*+*bi*) in the array. For each complex number, two sequential storage locations are required in the array. Therefore, exactly twice as much storage is required for complex matrices as for real matrices of the same precision. See "How Do You Set Up Your Scalar Data?" on page 47 for a description of real and complex numbers, and "How Do You Set Up Your Arrays?" on page 47 for a description of how real and complex data is stored in arrays.

How Leading Dimension Is Used for Matrices

The leading dimension for a two-dimensional array is an increment that is used to find the starting point for the matrix elements in each successive column of the array. To define exactly which elements become the conceptual matrix in the array, the following items are used together:

- The location of the matrix within the array
- The leading dimension
- The number of rows, *m*, to be processed in the array
- The number of columns, *n*, to be processed in the array

The leading dimension must always be positive. It must always be greater than or equal to *m*, the number of rows in the matrix to be processed. For an array, A, declared as `A(E1:E2,F1:F2)`, the leading dimension is equal to:

(E2-E1+1)

The starting point for selecting the matrix elements from the array is at the location specified by the argument for the matrix in the ESSL calling sequence. For example, if you specify `A(3,0)` for a 4 by 4 matrix ***A***, where A is declared as `A(1:7,0:4)`:

```
                 | 1.0   8.0  15.0  22.0  29.0 |
                 | 2.0   9.0  16.0  23.0  30.0 |
                 | 3.0  10.0  17.0  24.0  31.0 |
                 | 4.0  11.0  18.0  25.0  32.0 |
                 | 5.0  12.0  19.0  26.0  33.0 |
                 | 6.0  13.0  20.0  27.0  34.0 |
                 | 7.0  14.0  21.0  28.0  35.0 |
```

then processing begins at the element at row 3 and column 0 in array A, which is 3.0.

The leading dimension is used to find the starting point for the matrix elements in each of the *n* successive columns in the array. ESSL subroutines assume that the arrays are stored in column-major order, as described under "How Do You Set Up Your Arrays?" on page 47, and they add the leading dimension (times the size of the element in bytes) to the starting point. They do this *n*-1 times. This finds the starting point in each of the *n* columns of the array.

In the above example, the leading dimension is:

E2-E1+1 = 7-1+1 = 7

If the number of columns, *n*, to be processed is 4, the starting points are: `A(3,0), A(3,1), A(3,2),` and `A(3,3)`. These are elements 3.0, 10.0, 17.0, and 24.0 for a_{11}, a_{12}, a_{13}, and a_{14}, respectively.

In general terms, this results in the following starting positions of each column in the matrix being calculated as follows:

```
A(BEGINI, BEGINJ)
A(BEGINI, BEGINJ+1)
A(BEGINI, BEGINJ+2)
.
.
.
A(BEGINI, BEGINJ+n-1)
```

To find the elements in each column of the array, 1 is added successively to the starting point in the column until *m* elements are selected. This is why the leading dimension must be greater than or equal to *m*; otherwise, you go past the end of each dimension of the array. In the above example, if the number of elements, *m*, to be processed in each column is 4, the following elements are selected from array A for the first column of the matrix: `A(3,0), A(4,0), A(5,0)`, and `A(6,0)`. These are elements 3.0, 4.0, 5.0, and 6.0, corresponding to the matrix elements a_{11}, a_{21}, a_{31}, and a_{41}, respectively.

Column element selection can also be expressed in general terms. Using `A(BEGINI,BEGINJ)` as the starting point in the array, this results in the following elements being selected from each column in the array:

```
A(BEGINI, BEGINJ)
A(BEGINI+1, BEGINJ)
A(BEGINI+2, BEGINJ)
.
.
.
A(BEGINI+m-1, BEGINJ)
```

Combining this with the technique already described for finding the starting point in each column of the array, the resulting matrix in the example is:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \cdots & a_{14} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{41} & \cdots & a_{44} \end{bmatrix} = \begin{bmatrix} 3.0 & 10.0 & 17.0 & 24.0 \\ 4.0 & 11.0 & 18.0 & 25.0 \\ 5.0 & 12.0 & 19.0 & 26.0 \\ 6.0 & 13.0 & 20.0 & 27.0 \end{bmatrix}$$

As shown in this example, a matrix does not have to include all columns and rows of an array. The elements of matrix ***A*** are selected from rows 3 through 6 and columns 0 through 3 of the array. Rows 1, 2, and 7 and column 4 of the array are not used.

Symmetric Matrix

The matrix $\boldsymbol{A}$ is symmetric if it has the property $\boldsymbol{A} = \boldsymbol{A}^T$, which means:

- It has the same number of rows as it has columns; that is, it has n rows and n columns.
- The value of every element a_{ij} on one side of the main diagonal equals its mirror image a_{ji} on the other side: $a_{ij} = a_{ji}$ for $1 \le i \le n$ and $1 \le j \le n$.

The following matrix illustrates a symmetric matrix of order n; that is, it has n rows and n columns. The subscripts on each side of the diagonal appear the same to show which elements are equal:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{21} & a_{31} & \cdot & \cdot & \cdot & a_{n1} \\ a_{21} & a_{22} & a_{32} & & & & \cdot \\ a_{31} & a_{32} & a_{33} & & & & \cdot \\ \cdot & & & \cdot & & & \cdot \\ \cdot & & & & \cdot & & \cdot \\ \cdot & & & & & \cdot & \cdot \\ a_{n1} & \cdot & \cdot & \cdot & \cdot & \cdot & a_{nn} \end{bmatrix}$$

Symmetric Matrix Storage Representation

The four storage modes used for storing symmetric matrices are described in the following:

- “Lower-Packed Storage Mode” on page 80
- “Upper-Packed Storage Mode” on page 82
- “Lower Storage Mode” on page 83
- “Upper Storage Mode” on page 83

The storage technique you should use depends on the ESSL subroutine you are using.

Lower-Packed Storage Mode

When a symmetric matrix is stored in lower-packed storage mode, the lower triangular part of the symmetric matrix is stored, including the diagonal, in a one-dimensional array. The lower triangle is packed by columns. (This is equivalent to packing the upper triangle by rows.) The matrix is packed sequentially column by column in $n(n+1)/2$ elements of a one-dimensional array. To calculate the location of each element a_{ij} of matrix $\boldsymbol{A}$ in an array, AP, using the lower triangular packed technique, use the following formula:

AP($i + ((2n-j)(j-1)/2)$) = a_{ij} where $i \ge j$

This results in the following storage arrangement for the elements of a symmetric matrix $\boldsymbol{A}$ in an array AP:

`AP(1)`
  = a_{11} (start the first column)

`AP(2)`
  = a_{21}

`AP(3)`
  = a_{31}

.
  .

.
  .

.
    .

AP(*n*)
    = a_{n1}

AP(*n*+1)
    = a_{22} (start the second column)

AP(*n*+2)
    = a_{32}

.
    .

.
    .

.
    .

AP(2*n*-1)
    = a_{n2}

AP(2*n*)
    = a_{33} (start the third column and so forth)

AP(2*n*+1)
    = a_{43}

.
    .

.
    .

.
    .

AP(*n*(*n*+1)/2)
    = a_{nn}

Following is an example of a symmetric matrix that uses the element values to show the order in which the matrix elements are stored in the array.

Given the following matrix ***A***:

```
                        ┌                  ┐
                        | 1  2   3   4   5 |
                        | 2  6   7   8   9 |
                        | 3  7  10  11  12 |
                        | 4  8  11  13  14 |
                        | 5  9  12  14  15 |
                                           ┘
```

the array is:

```
          AP = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

Note: Additional work storage is required in the array for some ESSL subroutines; for example, in the simultaneous linear algebraic equation subroutines SPPF, DPPF, SPPS, and DPPS.

Following is an example of how to transform your symmetric matrix to lower-packed storage mode:

```
        K = 0
        DO 1 J=1,N
           DO 2 I=J,N
              K = K+1
              AP(K)=A(I,J)
    2      CONTINUE
    1   CONTINUE
```

Upper-Packed Storage Mode

When a symmetric matrix is stored in upper-packed storage mode, the upper triangular part of the symmetric matrix is stored, including the diagonal, in a one-dimensional array. The upper triangle is packed by columns. (This is equivalent to packing the lower triangle by rows.) The matrix is packed sequentially column by column in $n(n+1)/2$ elements of a one-dimensional array. To calculate the location of each element a_{ij} of matrix ***A*** in an array AP using the upper triangular packed technique, use the following formula:

AP(i+(j(j-1)/2)) = a_{ij} where $j \geq i$

This results in the following storage arrangement for the elements of a symmetric matrix ***A*** in an array AP:

AP(1)
  = a_{11} (start the first column)

AP(2)
  = a_{12} (start the second column)

AP(3)
  = a_{22}

AP(4)
  = a_{13} (start the third column)

AP(5)
  = a_{23}

AP(6)
  = a_{33}

AP(7)
  = a_{14} (start the fourth column)

•
  .

•
  .

•
  .

AP(*j*(*j*-1)/2+1)
  = a_{1j} (start the j-th column)

AP(*j*(*j*-1)/2+2)
  = a_{2j}

AP(*j*(*j*-1)/2+3)
  = a_{3j}

•
  .

•
  .

•
  .

AP(*j*(*j*-1)/2+*j*)
  = a_{jj} (end of the j-th column)

•
  .

•
  .

•
  .

AP(n(n+1)/2)
    = a_{nn}

Following is an example of a symmetric matrix that uses the element values to show the order in which the matrix elements are stored in the array. Given the following matrix ***A***:

```
                      |  1   2   4   7  11 |
                      |  2   3   5   8  12 |
                      |  4   5   6   9  13 |
                      |  7   8   9  10  14 |
                      | 11  12  13  14  15 |
```

the array is:

```
            AP = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

Following is an example of how to transform your symmetric matrix to upper-packed storage mode:

```
        K = 0
        DO 1 J=1,N
           DO 2 I=1,J
              K = K+1
              AP(K)=A(I,J)
    2      CONTINUE
    1   CONTINUE
```

Lower Storage Mode

When a symmetric matrix is stored in lower storage mode, the lower triangular part of the symmetric matrix is stored, including the diagonal, in a two-dimensional array. These elements are stored in the array in the same way they appear in the matrix. The upper part of the matrix is not required to be stored in the array.

Following is an example of a symmetric matrix ***A*** of order 5 and how it is stored in an array AL.

Given the following matrix ***A***:

```
                       | 1  2   3   4   5 |
                       | 2  6   7   8   9 |
                       | 3  7  10  11  12 |
                       | 4  8  11  13  14 |
                       | 5  9  12  14  15 |
```

the array is:

```
                       | 1  *   *   *   * |
                       | 2  6   *   *   * |
                AL  =  | 3  7  10   *   * |
                       | 4  8  11  13   * |
                       | 5  9  12  14  15 |
```

where "*" means you do not have to store a value in that position in the array. However, these storage positions are required.

Upper Storage Mode

When a symmetric matrix is stored in upper storage mode, the upper triangular part of the symmetric matrix is stored, including the diagonal, in a two-dimensional array. These elements are stored in the array in the same way they appear in the matrix. The lower part of the matrix is not required to be stored in the array.

Following is an example of a symmetric matrix ***A*** of order 5 and how it is stored in an array AU.

Given the following matrix ***A***:

```
                    | 1  2   3   4   5 |
                    | 2  6   7   8   9 |
                    | 3  7  10  11  12 |
                    | 4  8  11  13  14 |
                    | 5  9  12  14  15 |
```

the array is:

```
                    | 1  2   3   4   5 |
                    | *  6   7   8   9 |
              AU =  | *  *  10  11  12 |
                    | *  *   *  13  14 |
                    | *  *   *   *  15 |
```

where "*" means you do not have to store a value in that position in the array. However, these storage positions are required.

Positive Definite or Negative Definite Symmetric Matrix

A real symmetric matrix ***A*** is positive definite if and only if $\boldsymbol{x}^T\boldsymbol{Ax}$ is positive for all nonzero vectors ***x***.

A real symmetric matrix ***A*** is negative definite if and only if $\boldsymbol{x}^T\boldsymbol{Ax}$ is negative for all nonzero vectors ***x***.

Positive Definite or Negative Definite Symmetric Matrix Storage Representation

The positive definite or negative definite symmetric matrix is stored in the same way the symmetric matrix is stored. For a description of this storage technique, see "Symmetric Matrix" on page 80.

Indefinite Symmetric Matrix

A symmetric matrix ***A*** is indefinite if and only if $(\boldsymbol{x}^T\boldsymbol{Ax})\ (\boldsymbol{y}^T\boldsymbol{Ay}) < 0$ for some non-zero vectors ***x*** and ***y***.

Indefinite Symmetric Matrix Storage Representation

The indefinite symmetric matrix is stored in the same way the symmetric matrix is stored. For a description of this storage technique, see "Symmetric Matrix" on page 80.

Complex Hermitian Matrix

A complex matrix is Hermitian if it is equal to its conjugate transpose:

$\boldsymbol{H} = \boldsymbol{H}^H$

Complex Hermitian Matrix Storage Representation

The complex Hermitian matrix is stored using the same four techniques used for symmetric matrices:

- Lower-packed storage mode, as described in "Lower-Packed Storage Mode" on page 80. (The complex Hermitian matrix is not symmetric; therefore, lower-packed storage mode is not equivalent to packing the upper triangle by rows, as it is for a symmetric matrix.)
- Upper-packed storage mode, as described in "Upper-Packed Storage Mode" on page 82. (The complex Hermitian matrix is not symmetric; therefore, upper-packed storage mode is not equivalent to packing the lower triangle by rows, as it is for a symmetric matrix.)
- Lower storage mode, as described in "Lower Storage Mode" on page 83.
- Upper storage mode, as described in "Upper Storage Mode" on page 83.

Following is an example of a complex Hermitian matrix ***H*** of order 5.

Given the following matrix ***H***:

```
        | (11,  0) (21, -1) (31,  1) (41, -1) (51, -1) |
        | (21,  1) (22,  0) (32, -1) (42, -1) (52,  1) |
        | (31, -1) (32,  1) (33,  0) (43, -1) (53, -1) |
        | (41,  1) (42,  1) (43,  1) (44,  0) (54, -1) |
        | (51,  1) (52, -1) (53,  1) (54,  1) (55,  0) |
```

it is stored in a one-dimensional array, HP, in *n*(*n*+1)/2 = 15 elements as follows:

- In lower-packed storage mode:

```
        HP = ((11, *), (21, 1), (31, -1), (41, 1), (51, 1),
              (22, *), (32, 1), (42, 1), (52, -1), (33, *),
              (43, 1), (53, 1), (44, *), (54, 1), (55, *))
```

- In upper-packed storage mode:

```
        HP = ((11, *), (21, -1), (22, *), (31, 1), (32, -1),
              (33, *), (41, -1), (42, -1), 43, -1), (44, *),
              (51, -1), (52, 1), (53, -1), (54, -1), (55, *))
```

or it is stored in a two-dimensional array, HP, as follows:

- In lower storage mode:

```
             | (11,  *)   *        *       *       *     |
             | (21,  1) (22,  *)   *       *       *     |
      HP =   | (31, -1) (32,  1) (33, *)   *       *     |
             | (41,  1) (42,  1) (43, 1) (44, *)   *     |
             | (51,  1) (52, -1) (53, 1) (54, 1) (55, *) |
```

- In upper storage mode

```
             | (11, *) (21, -1) (31,  1) (41, -1) (51, -1) |
             |    *    (22,  *) (32, -1) (42, -1) (52,  1) |
      HP =   |    *       *     (33,  *) (43, -1) (53, -1) |
             |    *       *        *     (44,  *) (54, -1) |
             |    *       *        *        *     (55,  *) |
```

where "*" means you do not have to store a value in that position in the array. The imaginary parts of the diagonal elements of a complex Hermitian matrix are always 0, so you do not need to set these values. The ESSL subroutines always assume that the values in these positions are 0.

Positive Definite or Negative Definite Complex Hermitian Matrix

A complex Hermitian matrix ***A*** is positive definite if and only if $\boldsymbol{x}^H\boldsymbol{Ax}$ is positive for all nonzero vectors ***x***.

A complex Hermitian matrix ***A*** is negative definite if and only if $\boldsymbol{x}^H\boldsymbol{Ax}$ is negative for all nonzero vectors ***x***.

Positive Definite or Negative Definite Complex Hermitian Matrix Storage Representation

The positive definite or negative definite complex Hermitian matrix is stored in the same way the complex Hermitian matrix is stored. For a description of this storage technique, see “Complex Hermitian Matrix” on page 84.

Indefinite Complex Hermitian Matrix

A complex Hermitian matrix A is indefinite if and only if $(\boldsymbol{x}^H\boldsymbol{Ax})$ $(\boldsymbol{y}^H\boldsymbol{Ay}) < 0$ for some non-zero vectors ***x*** and ***y***.

Indefinite Complex Hermitian Matrix Storage Representation

The indefinite complex Hermitian matrix is stored in the same way the complex Hermitian matrix is stored. For a description of this storage technique, see "Complex Hermitian Matrix" on page 84.

Positive Definite or Negative Definite Symmetric Toeplitz Matrix

A positive definite or negative definite symmetric matrix ***A*** of order *n* is also a Toeplitz matrix if and only if:

$a_{ij} = a_{i-1,j-1}$ for $i = 2, n$ and $j = 2, n$

The elements on each diagonal of the Toeplitz matrix have a constant value. For the definition of a positive definite or negative definite symmetric matrix, see "Positive Definite or Negative Definite Symmetric Matrix" on page 84.

The following matrix illustrates a symmetric Toeplitz matrix of order *n*; that is, it has *n* rows and *n* columns:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{21} & a_{31} & . & . & . & a_{n1} \\ a_{21} & a_{11} & a_{21} & . & & & . \\ a_{31} & a_{21} & a_{11} & & . & & . \\ . & . & & . & & . & . \\ . & & . & & . & & a_{31} \\ . & & & . & & . & a_{21} \\ a_{n1} & . & . & . & a_{31} & a_{21} & a_{11} \end{bmatrix}$$

A symmetric Toeplitz matrix of order *n* is represented by a vector ***x*** of length *n* containing the elements of the first column of the matrix (or the elements of the first row), such that $x_i = a_{i1}$ for $i = 1, n$.

The following vector represents the matrix ***A*** shown above:

$$\boldsymbol{x} = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ . \\ . \\ a_{n1} \end{bmatrix}$$

Positive Definite or Negative Definite Symmetric Toeplitz Matrix Storage Representation

The elements of the vector ***x***, which represent a positive definite symmetric Toeplitz matrix, are stored sequentially in an array. This is called packed-symmetric-Toeplitz storage mode. Following is an example of a positive definite symmetric Toeplitz matrix ***A*** and how it is stored in an array X.

Given the following matrix ***A***:

```
                 | 99  12  13  14  15  16 |
                 | 12  99  12  13  14  15 |
                 | 13  12  99  12  13  14 |
                 | 14  13  12  99  12  13 |
                 | 15  14  13  12  99  12 |
                    | 16  15  14  13  12  99 |
```

the array is:

```
                    X = (99, 12, 13, 14, 15, 16)
```

Positive Definite or Negative Definite Complex Hermitian Toeplitz Matrix

A positive definite or negative definite complex Hermitian matrix ***A*** of order *n* is also a Toeplitz matrix if and only if:

$a_{ij} = a_{i-1,j-1}$ for $i = 2, n$ and $j = 2, n$

The real part of the diagonal elements of the Toeplitz matrix must have a constant value. The imaginary part of the diagonal elements must be zero.

For the definition of a positive definite of negative definite complex Hermitian matrix, see "Positive Definite or Negative Definite Complex Hermitian Matrix" on page 85.

The following matrix illustrates a complex Hermitian Toeplitz matrix of order *n*; that is, it has *n* rows and *n* columns:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdot & \cdot & \cdot & a_{1n} \\ \bar{a}_{12} & a_{11} & a_{12} & \cdot & & & \cdot \\ \bar{a}_{13} & \bar{a}_{12} & a_{11} & & \cdot & & \cdot \\ \cdot & \cdot & & \cdot & & \cdot & \cdot \\ \cdot & & \cdot & & \cdot & & a_{13} \\ \cdot & & & \cdot & & \cdot & a_{12} \\ \bar{a}_{1n} & \cdot & \cdot & \cdot & \bar{a}_{13} & \bar{a}_{12} & a_{11} \end{bmatrix}$$

A complex Hermitian Toeplitz matrix of order *n* is represented by a vector ***x*** of length *n* containing the elements of the first row of the matrix.

The following vector represents the matrix ***A*** shown above.

$$\boldsymbol{x} = \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ \cdot \\ \cdot \\ a_{1n} \end{bmatrix}$$

Positive Definite or Negative Definite Complex Hermitian Toeplitz Matrix Storage Representation

The elements of the vector ***x***, which represent a positive definite complex Hermitian Toeplitz matrix, are stored sequentially in an array. This is called packed-Hermitian-Toeplitz storage mode. Following is an example of a positive definite complex Hermitian Toeplitz matrix ***A*** and how it is stored in an array X.

Given the following matrix ***A***:

```
     |  (10.0, 0.0)   (2.0, -3.0)  (-3.0, 1.0)   (1.0, 1.0) |
     |   (2.0, 3.0)   (10.0, 0.0)  (2.0, -3.0)  (-3.0, 1.0) |
     | (-3.0, -1.0)    (2.0, 3.0)  (10.0, 0.0)  (2.0, -3.0) |
     |  (1.0, -1.0)  (-3.0, -1.0)   (2.0, 3.0)  (10.0, 0.0) |
```

the array is:

```
        X = ((10.0, 0.0), (2.0, -3.0), (-3.0, 1.0), (1.0, 1.0))
```

Triangular Matrix

There are two types of triangular matrices: upper triangular matrix and lower triangular matrix. Triangular matrices have the same number of rows as they have columns; that is, they have *n* rows and *n* columns.

A matrix ***U*** is an upper triangular matrix if its nonzero elements are found only in the upper triangle of the matrix, including the main diagonal; that is:

$u_{ij} = 0 \quad \text{if } i > j$

A matrix ***L*** is an lower triangular matrix if its nonzero elements are found only in the lower triangle of the matrix, including the main diagonal; that is:

$l_{ij} = 0 \quad \text{if } i < j$

The following matrices, ***U*** and ***L***, illustrate upper and lower triangular matrices of order *n*, respectively:

$$
\boldsymbol{U} = \begin{bmatrix}
u_{11} & u_{12} & u_{13} & \cdot & \cdot & \cdot & u_{1n} \\
0 & u_{22} & u_{23} & & & & \cdot \\
0 & 0 & u_{33} & & & & \cdot \\
\cdot & & & \cdot & & & \cdot \\
\cdot & & & & \cdot & & \cdot \\
\cdot & & & & & \cdot & \cdot \\
0 & \cdot & \cdot & \cdot & \cdot & 0 & u_{nn}
\end{bmatrix}
\qquad
\boldsymbol{L} = \begin{bmatrix}
l_{11} & 0 & 0 & \cdot & \cdot & \cdot & 0 \\
l_{21} & l_{22} & 0 & & & & \cdot \\
l_{31} & l_{32} & l_{33} & & & & \cdot \\
\cdot & & & \cdot & & & \cdot \\
\cdot & & & & \cdot & & \cdot \\
\cdot & & & & & \cdot & 0 \\
l_{n1} & \cdot & \cdot & \cdot & \cdot & \cdot & l_{nn}
\end{bmatrix}
$$

A unit triangular matrix is a triangular matrix in which all the diagonal elements have a value of one; that is:

- For an upper triangular matrix, $u_{ij} = 1$ if $i = j$.
- For an lower triangular matrix, $l_{ij} = 1$ if $i = j$.

The following matrices, ***U*** and ***L***, illustrate upper and lower unit real triangular matrices of order *n*, respectively:

$$U = \begin{bmatrix} 1 & u_{12} & u_{13} & . & . & . & u_{1n} \\ 0 & 1 & u_{23} & & & & . \\ 0 & 0 & 1 & & & & . \\ . & & & . & & & . \\ . & & & & . & & . \\ . & & & & & . & . \\ 0 & . & . & . & . & 0 & 1 \end{bmatrix} \qquad L = \begin{bmatrix} 1 & 0 & 0 & . & . & . & 0 \\ l_{21} & 1 & 0 & & & & . \\ l_{31} & l_{32} & 1 & & & & . \\ . & & & . & & & . \\ . & & & & . & & . \\ . & & & & & . & 0 \\ l_{n1} & . & . & . & . & . & 1 \end{bmatrix}$$

Triangular Matrix Storage Representation

The four storage modes used for storing triangular matrices are described in the following:

- "Upper-Triangular-Packed Storage Mode" on page 89
- "Lower-Triangular-Packed Storage Mode" on page 89
- "Upper-Triangular Storage Mode" on page 90
- "Lower-Triangular Storage Mode" on page 90

It is important to note that because the diagonal elements of a unit triangular matrix are always one, you do not need to set these values in the array for these four storage modes. ESSL always assumes that the values in these positions are one.

Upper-Triangular-Packed Storage Mode

When an upper-triangular matrix is stored in upper-triangular-packed storage mode, the upper triangle of the matrix is stored, including the diagonal, in a one-dimensional array. The upper triangle is packed by columns. The elements are packed sequentially, column by column, in *n*(*n*+1)/2 elements of a one-dimensional array. To calculate the location of each element of the triangular matrix in the array, use the technique described in "Upper-Packed Storage Mode" on page 82.

Following is an example of an upper triangular matrix ***U*** of order 5 and how it is stored in array UP. It uses the element values to show the order in which the elements are stored in the one-dimensional array.

Given the following matrix ***U***:

```
                        | 1  2  4   7  11 |
                        | 0  3  5   8  12 |
                        | 0  0  6   9  13 |
                        | 0  0  0  10  14 |
                        | 0  0  0   0  15 |
```

the array is:

```
           UP = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

Lower-Triangular-Packed Storage Mode

When a lower-triangular matrix is stored in lower-triangular-packed storage mode, the lower triangle of the matrix is stored, including the diagonal, in a one-dimensional array. The lower triangle is packed by columns. The elements are packed sequentially, column by column, in *n*(*n*+1)/2 elements of a one-dimensional array. To calculate the location of each element of the triangular matrix in the array, use the technique described in "Lower-Packed Storage Mode" on page 80.

Following is an example of a lower triangular matrix ***L*** of order 5 and how it is stored in array LP. It uses the element values to show the order in which the elements are stored in the one-dimensional array.

Given the following matrix ***L***:

```
                        | 1  0   0   0   0 |
                        | 2  6   0   0   0 |
                        | 3  7  10   0   0 |
                        | 4  8  11  13   0 |
                        | 5  9  12  14  15 |
```

the array is:

```
           LP = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

Upper-Triangular Storage Mode

A triangular matrix is stored in upper-triangular storage mode in a two-dimensional array. Only the elements in the upper triangle of the matrix, including the diagonal, are stored in the upper triangle of the array.

Following is an example of an upper triangular matrix ***U*** of order 5 and how it is stored in array UTA.

Given the following matrix ***U***:

```
                       | 11  12  13  14  15 |
                       |  0  22  23  24  25 |
                       |  0   0  33  34  35 |
                       |  0   0   0  44  45 |
                       |  0   0   0   0  55 |
```

the array is:

```
                       | 11  12  13  14  15 |
                       |  *  22  23  24  25 |
                UTA =  |  *   *  33  34  35 |
                       |  *   *   *  44  45 |
                       |  *   *   *   *  55 |
```

where "*" means you do not have to store a value in that position in the array.

Lower-Triangular Storage Mode

A triangular matrix is stored in lower-triangular storage mode in a two-dimensional array. Only the elements in the lower triangle of the matrix, including the diagonal, are stored in the lower triangle of the array.

Following is an example of a lower triangular matrix ***L*** of order 5 and how it is stored in array LTA.

Given the following matrix ***L***:

```
                       | 11   0   0   0   0 |
                       | 21  22   0   0   0 |
                       | 31  32  33   0   0 |
                       | 41  42  43  44   0 |
                       | 51  52  53  54  55 |
```

the array is:

```
                       | 11   *   *   *   * |
                       | 21  22   *   *   * |
                 LTA = | 31  32  33   *   * |
                       | 41  42  43  44   * |
                       | 51  52  53  54  55 |
```

where "*" means you do not have to store a value in that position in the array.

Trapezoidal Matrix

There are two types of trapezoidal matrices: upper trapezoidal matrix and lower trapezoidal matrix. Trapezoidal matrices have *m* rows and *n* columns.

A matrix ***U*** is an upper trapezoidal matrix if its nonzero elements are found only in the upper triangle of the matrix, including the main diagonal; that is:

$u_{ij} = 0 \quad \text{if } i > j$

A matrix ***L*** is an lower trapezoidal matrix if its nonzero elements are found only in the lower triangle of the matrix, including the main diagonal; that is:

$l_{ij} = 0 \quad \text{if } i < j$

The following matrices, ***U*** and ***L***, illustrate upper and lower trapezoidal matrices with *m* rows and *n* columns, respectively:

If $m \geq n$:

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & & \vdots \\ 0 & 0 & u_{33} & & \vdots \\ \vdots & & & \ddots & \vdots \\ 0 & \cdots & & 0 & u_{nn} \\ 0 & \cdots & & \cdots & 0 \\ \vdots & & & & \vdots \\ 0 & \cdots & & \cdots & 0 \end{bmatrix} \qquad L = \begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & & \vdots \\ l_{31} & l_{32} & l_{33} & & \vdots \\ \vdots & & & \ddots & 0 \\ l_{n1} & \cdots & & \cdots & l_{nn} \\ l_{n+1,1} & \cdots & & \cdots & l_{n+1,n} \\ l_{n+2,1} & \cdots & & \cdots & l_{n+2,n} \\ \vdots & & & & \vdots \\ l_{m1} & \cdots & & \cdots & l_{mn} \end{bmatrix}$$

If $m < n$:

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1m} & u_{1,m+1} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & & \vdots & u_{2,m+1} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & & \vdots & \vdots & & \vdots \\ \vdots & & & \ddots & \vdots & \vdots & & \vdots \\ 0 & \cdots & & 0 & u_{mm} & u_{m,m+1} & \cdots & u_{mn} \end{bmatrix} \qquad L = \begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & & \vdots & \vdots & & \vdots \\ l_{31} & l_{32} & l_{33} & & \vdots & \vdots & & \vdots \\ \vdots & & & \ddots & 0 & \vdots & & \vdots \\ l_{m1} & \cdots & & \cdots & l_{mm} & 0 & \cdots & 0 \end{bmatrix}$$

A unit trapezoidal matrix is a trapezoidal matrix in which all the diagonal elements have a value of one; that is:

- For an upper trapezoidal matrix, $u_{ij} = 1$ if $i = j$.
- For a lower trapezoidal matrix, $l_{ij} = 1$ if $i = j$.

The following matrices, ***U*** and ***L***, illustrate upper and lower unit real trapezoidal matrices with *m* and *n* columns, respectively:

If *m* ≥ *n*:

$$U = \begin{bmatrix} 1 & u_{12} & u_{13} & \cdot & \cdot & \cdot & u_{1n} \\ 0 & 1 & u_{23} & & & & \cdot \\ 0 & 0 & 1 & & & & \cdot \\ \cdot & & & \cdot & & & \cdot \\ \cdot & & & & \cdot & & \cdot \\ \cdot & & & & & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot & \cdot & 0 & 1 \\ 0 & \cdot & \cdot & \cdot & \cdot & \cdot & 0 \\ \cdot & & & & & & \cdot \\ \cdot & & & & & & \cdot \\ \cdot & & & & & & \cdot \\ 0 & \cdot & \cdot & \cdot & \cdot & \cdot & 0 \end{bmatrix} \qquad L = \begin{bmatrix} 1 & 0 & 0 & \cdot & \cdot & \cdot & 0 \\ l_{21} & 1 & 0 & & & & \cdot \\ l_{31} & l_{32} & 1 & & & & \cdot \\ \cdot & & & \cdot & & & \cdot \\ \cdot & & & & \cdot & & \cdot \\ \cdot & & & & & \cdot & 0 \\ l_{n1} & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\ l_{n+1,1} & \cdot & \cdot & \cdot & \cdot & \cdot & l_{n+1,n} \\ l_{n+2,1} & \cdot & \cdot & \cdot & \cdot & \cdot & l_{n+2,n} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ l_{m1} & \cdot & \cdot & \cdot & \cdot & \cdot & l_{mn} \end{bmatrix}$$

If *m* < *n*:

$$U = \begin{bmatrix} 1 & u_{12} & u_{13} & \cdot & \cdot & \cdot & u_{1n} & u_{1,m+1} & \cdot & \cdot & \cdot & \cdot & u_{1n} \\ 0 & 1 & u_{23} & & & & \cdot & u_{2,m+1} & \cdot & \cdot & \cdot & \cdot & u_{2n} \\ 0 & 0 & 1 & & & & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & & & \cdot & & & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & & & & \cdot & & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & & & & & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot & \cdot & 0 & 1 & u_{m,m+1} & \cdot & \cdot & \cdot & \cdot & u_{mn} \end{bmatrix} \qquad L = \begin{bmatrix} 1 & 0 & 0 & \cdot & \cdot & \cdot & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & 0 \\ l_{21} & 1 & 0 & & & & \cdot & \cdot & & & & & & \cdot \\ l_{31} & l_{32} & 1 & & & & \cdot & \cdot & & & & & & \cdot \\ \cdot & & & \cdot & & & \cdot & \cdot & & & & & & \cdot \\ \cdot & & & & \cdot & & \cdot & \cdot & & & & & & \cdot \\ \cdot & & & & & \cdot & 0 & \cdot & & & & & & \cdot \\ l_{n1} & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & 0 \end{bmatrix}$$

Trapezoidal Matrix Storage Representation

The storage modes used for storing trapezoidal matrices are described in the following:

- "Upper-Trapezoidal Storage Mode" on page 92
- "Lower-Trapezoidal Storage Mode" on page 93

It is important to note that because the diagonal elements of a unit trapezoidal matrix are always one, you do not need to set these values in the array for these storage modes. ESSL always assumes that the values in these positions are one.

Upper-Trapezoidal Storage Mode

A trapezoidal matrix is stored in upper-trapezoidal storage mode in a two-dimensional array. Only the elements in the upper trapezoid of the matrix, including the diagonal, are stored in the upper trapezoid of the array.

Following is an example of an upper trapezoidal matrix ***U*** of order 5 and how it is stored in array UTA.

Given the following matrix ***U***:

```
| 11  12  13  14  15 |
|  0  22  23  24  25 |
|  0   0  33  34  35 |
|  0   0   0  44  45 |
|  0   0   0   0  55 |
|  0   0   0   0   0 |
|  0   0   0   0   0 |
```

the array is:

```
                 | 11  12  13  14  15 |
                 |  *  22  23  24  25 |
                 |  *   *  33  34  35 |
          UTA =  |  *   *   *  44  45 |
                 |  *   *   *   *  55 |
                 |  *   *   *   *   * |
                 |  *   *   *   *   * |
```

where "*" means you do not have to store a value in that position in the array.

Following is an example of an upper trapezoidal matrix ***U*** with 5 rows and 7 columns and how it is stored in array ***UTA***.

Given the following matrix ***U***:

```
                 | 11  12  13  14  15 16  17 |
                 |  0  22  23  24  25 26  27 |
                 |  0   0  33  34  35 36  37 |
                 |  0   0   0  44  45 46  47 |
                 |  0   0   0   0  55 56  57 |
```

the array is:

```
                 | 11  12  13  14  15 16  17 |
                 |  *  22  23  24  25 26  27 |
          UTA =  |  *   *  33  34  35 36  37 |
                 |  *   *   *  44  45 46  47 |
                 |  *   *   *   *  55 56  57 |
```

where "*" means you do not have to store a value in that position in the array.

Lower-Trapezoidal Storage Mode

A trapezoidal matrix is stored in lower-trapezoidal storage mode in a two-dimensional array. Only the elements in the lower trapezoid of the matrix, including the diagonal, are stored in the lower trapezoid of the array.

Following is an example of a lower trapezoidal matrix ***L*** of order 5 and how it is stored in array LTA.

Given the following matrix ***L***:

```
                 | 11   0   0   0   0 |
                 | 21  22   0   0   0 |
                 | 31  32  33   0   0 |
                 | 41  42  43  44   0 |
                 | 51  52  53  54  55 |
                 | 61  62  63  64  65 |
                 | 71  72  73  74  75 |
```

the array is:

```
                 | 11   *   *   *   * |
                 | 21  22   *   *   * |
           LTA = | 31  32  33   *   * |
                 | 41  42  43  44   * |
                 | 51  52  53  54  55 |
                 | 61  62  63  64  65 |
                 | 71  72  73  74  75 |
```

where "⋆" means you do not have to store a value in that position in the array.

Following is an example of an lower trapezoidal matrix ***U*** with 5 rows and 7 columns and how it is stored in array LTA.

Given the following matrix ***L***:

```
                         | 11   0   0   0   0   0   0 |
                         | 21  22   0   0   0   0   0 |
                         | 31  32  33   0   0   0   0 |
                         | 41  42  43  44   0   0   0 |
                         | 51  52  53  54  55   0   0 |
```

the array is:

```
                         | 11   ⋆   ⋆   ⋆   ⋆   ⋆   ⋆ |
                         | 21  22   ⋆   ⋆   ⋆   ⋆   ⋆ |
                   LTA = | 31  32  33   ⋆   ⋆   ⋆   ⋆ |
                         | 41  42  43  44   ⋆   ⋆   ⋆ |
                         | 51  52  53  54  55   ⋆   ⋆ |
```

where "⋆" means you do not have to store a value in that position in the array.

General Band Matrix

A general band matrix has its nonzero elements arranged uniformly near the diagonal, such that:

$a_{ij} = 0 \quad \text{if } (i\text{-}j) > ml \text{ or } (j\text{-}i) > mu$

where *ml* and *mu* are the lower and upper band widths, respectively, and *ml*+*mu*+1 is the total band width.

The following matrix illustrates a **square** general band matrix of order *n*, where the band widths are *ml* = *q*-1 and *mu* = *p*-1:

$$\begin{array}{cc|cccccccccc|}
 & & & \multicolumn{5}{c}{\big| \leftarrow mu \rightarrow \big|} & & & & \\
 & & a_{11} & a_{12} & a_{13} & . & . & a_{1p} & 0 & . & . & 0 \\
 & - & a_{21} & a_{22} & a_{23} & & & & . & 0 & & . \\
 & \uparrow & a_{31} & a_{32} & a_{33} & & & & & . & 0 & . \\
 & ml & . & & & . & & & & & . & 0 \\
\boldsymbol{A} = & \downarrow & . & & & & . & & & & & . \\
 & - & a_{q1} & & & & & . & & & & . \\
 & & 0 & . & & & & & . & & & . \\
 & & . & 0 & . & & & & & . & & . \\
 & & . & & 0 & . & & & & & . & \\
 & & 0 & . & . & 0 & . & . & . & & & a_{nn}
\end{array}$$

Some special types of band matrices are:

- Tridiagonal matrix: *ml* = *mu* = 1
- 9-diagonal matrix: *ml* = *mu* = 4

The following two matrices illustrate *m* by *n* **rectangular** general band matrices, where the band widths are *ml* = *q*-1 and *mu* = *p*-1. For both matrices, the leading diagonal is a_{11}, a_{22}, a_{33}, ..., a_{nn}. Following is a general band matrix with *m* > *n*:

$$\begin{array}{cc}
 & \begin{array}{l} \Big| \leftarrow mu \rightarrow \Big| \end{array} \\
A = \begin{array}{c} \\ \\ \\ - \\ \uparrow \\ ml \\ \downarrow \\ - \\ \\ \\ \\ \\ \\ \end{array} &
\left[\begin{array}{cccccccccc}
a_{11} & a_{12} & a_{13} & . & . & a_{1p} & 0 & . & . & 0 \\
a_{21} & a_{22} & a_{23} & & & & . & 0 & & . \\
a_{31} & a_{32} & a_{33} & & & & & . & 0 & . \\
. & & & . & & & & & . & 0 \\
. & & & & . & & & & & . \\
a_{q1} & & & & & . & & & & . \\
0 & . & & & & & . & & & . \\
. & 0 & . & & & & & . & & . \\
. & & 0 & . & & & & & . & \\
. & & & 0 & . & & & & & a_{nn} \\
. & & & & 0 & . & & & & . \\
. & & & & & 0 & . & & & . \\
0 & . & . & . & . & . & 0 & . & . & a_{mn}
\end{array}\right]
\end{array}$$

Following is a general band matrix with *m* < *n*:

$$\begin{array}{cc}
 & \begin{array}{l} \Big| \leftarrow mu \rightarrow \Big| \end{array} \\
A = \begin{array}{c} \\ - \\ \uparrow \\ ml \\ \downarrow \\ - \\ \\ \\ \\ \\ \end{array} &
\left[\begin{array}{ccccccccccccc}
a_{11} & a_{12} & a_{13} & . & . & a_{1p} & 0 & . & . & . & . & . & 0 \\
a_{21} & a_{22} & a_{23} & & & & . & 0 & & & & & . \\
a_{31} & a_{32} & a_{33} & & & & & . & 0 & & & & . \\
. & & & . & & & & & . & 0 & & & . \\
. & & & & . & & & & & . & 0 & & . \\
a_{q1} & & & & & . & & & & & . & 0 & . \\
0 & . & & & & & . & & & & & . & 0 \\
. & 0 & . & & & & & . & & & & & . \\
. & & 0 & . & & & & & . & & & & . \\
0 & . & . & 0 & . & . & . & . & . & a_{nn} & . & . & a_{mn}
\end{array}\right]
\end{array}$$

General Band Matrix Storage Representation

The two storage modes used for storing general band matrices are described in the following:

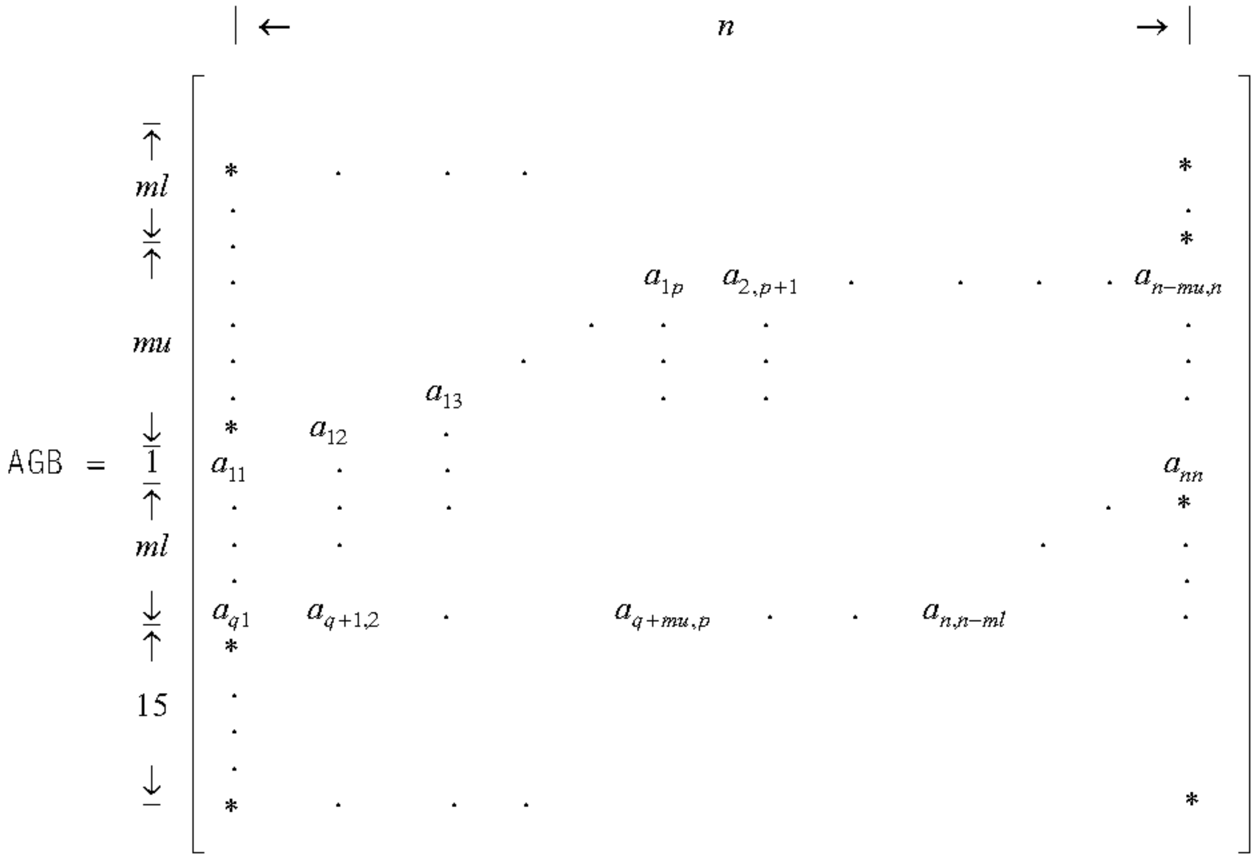
- “General-Band Storage Mode” on page 96
- “BLAS-General-Band Storage Mode” on page 97

General-Band Storage Mode

(This storage mode is used only for square matrices.) Only the band elements of a general band matrix are stored for general-band storage mode. Additional storage must also be provided for fill- in. General-band storage mode packs the matrix elements by columns into a two-dimensional array, such that each diagonal of the matrix appears as a row in the packed array.

For a matrix ***A*** of order *n* with band widths *ml* and *mu*, the array must have a leading dimension, *lda*, greater than or equal to 2*ml*+*mu*+16. The size of the second dimension must be (at least) *n*, the number of columns in the matrix.

Using array AGB, which is declared as AGB(2*ml*+*mu*+16, *n*), the columns of elements in matrix ***A*** are stored in each column in array AGB as follows, where a_{11} is stored at AGB(*ml*+*mu*+1, 1):



where "*" means you do not store an element in that position in the array.

In the ESSL subroutine computation, some of the positions in the array indicated by an "*" are used for fill- in. Other positions may not be accessed at all.

Following is an example of a band matrix ***A*** of order 9 and band widths of *ml* = 2 and *mu* = 3.

Given the following matrix ***A***:

```
                | 11  12  13  14   0   0   0   0   0 |
                | 21  22  23  24  25   0   0   0   0 |
                | 31  32  33  34  35  36   0   0   0 |
                |  0  42  43  44  45  46  47   0   0 |
                |  0   0  53  54  55  56  57  58   0 |
                |  0   0   0  64  65  66  67  68  69 |
                |  0   0   0   0  75  76  77  78  79 |
                |  0   0   0   0   0  86  87  88  89 |
             | 0   0   0   0   0   0  97  98  99 |
```

you store it in general-band storage mode in a 23 by 9 array AGB as follows, where a_{11} is stored in `AGB(6,1)`:

```
             |  *   *   *   *   *   *   *   *   * |
             |  *   *   *   *   *   *   *   *   * |
             |  *   *   *  14  25  36  47  58  69 |
             |  *   *  13  24  35  46  57  68  79 |
             |  *  12  23  34  45  56  67  78  89 |
             | 11  22  33  44  55  66  77  88  99 |
             | 21  32  43  54  65  76  87  98   * |
             | 31  42  53  64  75  86  97   *   * |
             |  *   *   *   *   *   *   *   *   * |
             |  *   *   *   *   *   *   *   *   * |
             |  *   *   *   *   *   *   *   *   * |
       AGB = |  *   *   *   *   *   *   *   *   * |
             |  *   *   *   *   *   *   *   *   * |
             |  *   *   *   *   *   *   *   *   * |
             |  *   *   *   *   *   *   *   *   * |
             |  *   *   *   *   *   *   *   *   * |
             |  *   *   *   *   *   *   *   *   * |
             |  *   *   *   *   *   *   *   *   * |
             |  *   *   *   *   *   *   *   *   * |
             |  *   *   *   *   *   *   *   *   * |
             |  *   *   *   *   *   *   *   *   * |
             |  *   *   *   *   *   *   *   *   * |
             |  *   *   *   *   *   *   *   *   * |
```

Following is an example of how to transform your general band matrix, of order *n*, to general-band storage mode:

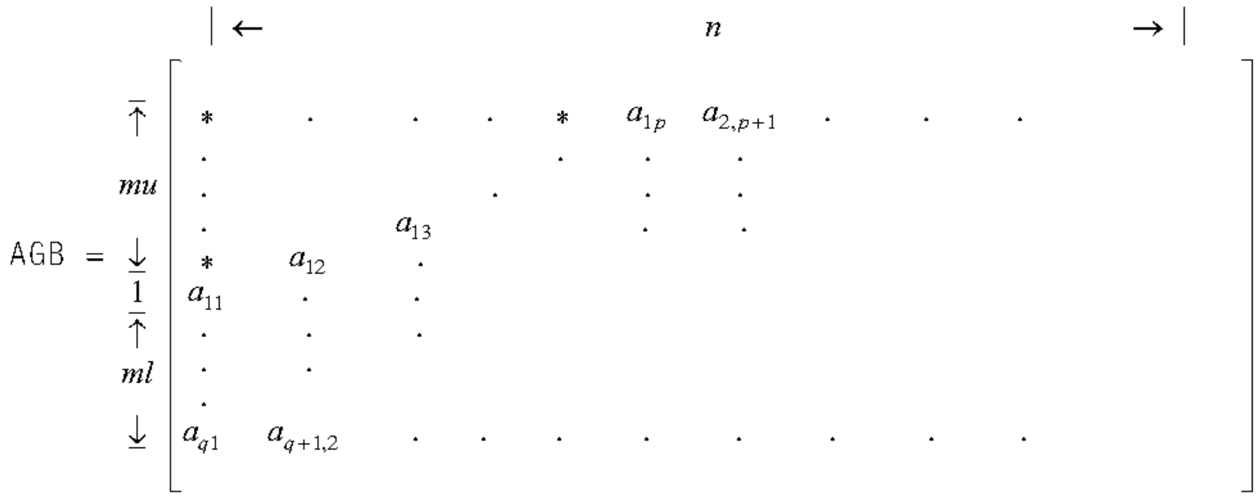
```
      MD=ML+MU+1
      DO 1 J=1,N
         DO 1 I=MAX(J-MU,1),MIN(J+ML,N)
            AGB(I-J+MD,J)=A(I,J)
  1   CONTINUE
```

BLAS-General-Band Storage Mode

(This storage mode is used for both square and rectangular matrices.) Only the band elements of a general band matrix are stored for BLAS-general-band storage mode. The storage mode packs the matrix elements by columns into a two-dimensional array, such that each diagonal of the matrix appears as a row in the packed array.

For an *m* by *n* matrix ***A*** with band widths *ml* and *mu*, the array AGB must have a leading dimension, *lda*, greater than or equal to *ml*+*mu*+1. The size of the second dimension must be (at least) *n*, the number of columns in the matrix.

Using the array AGB, which is declared as AGB(*ml*+*mu*+1, *n*), the columns of elements in matrix ***A*** are stored in each column in array AGB as follows, where a_{11} is stored at AGB(*mu*+1, 1):



where "*" means you do not store an element in that position in the array. These positions are not accessed by ESSL. **Unused positions in the array always occur in the upper left triangle of the array, but may not occur in the lower right triangle of the array, as you can see from the examples given here.**

Following is an example where *m* > *n*, and general band matrix ***A*** is 9 by 8 with band widths of *ml* = 2 and *mu* = 3.

Given the following matrix ***A***:

```
                | 11  12  13  14   0   0   0   0 |
                | 21  22  23  24  25   0   0   0 |
                | 31  32  33  34  35  36   0   0 |
                |  0  42  43  44  45  46  47   0 |
                |  0   0  53  54  55  56  57  58 |
                |  0   0   0  64  65  66  67  68 |
                |  0   0   0   0  75  76  77  78 |
                |  0   0   0   0   0  86  87  88 |
                |  0   0   0   0   0   0  97  98 |
```

you store it in array AGB, declared as `AGB(6,8)`, as follows, where a_{11} is stored in `AGB(4,1)`:

```
                |  *   *   *  14  25  36  47  58 |
                |  *   *  13  24  35  46  57  68 |
         AGB =  |  *  12  23  34  45  56  67  78 |
                | 11  22  33  44  55  66  77  88 |
                | 21  32  43  54  65  76  87  98 |
                | 31  42  53  64  75  86  97   * |
```

Following is an example where *m* < *n*, and general band matrix ***A*** is 7 by 9 with band widths of *ml* = 2 and *mu* = 3.

Given the following matrix ***A***:

```
               | 11  12  13  14   0   0   0   0   0 |
               | 21  22  23  24  25   0   0   0   0 |
               | 31  32  33  34  35  36   0   0   0 |
               |  0  42  43  44  45  46  47   0   0 |
               |  0   0  53  54  55  56  57  58   0 |
               |  0   0   0  64  65  66  67  68  69 |
               |  0   0   0   0  75  76  77  78  79 |
```

you store it in array AGB, declared as AGB(6,9), as follows, where a_{11} is stored in AGB(4,1) and the leading diagonal does not fill up the whole row:

```
          | *   *   *  14  25  36  47  58  69 |
          | *   *  13  24  35  46  57  68  79 |
    AGB = | *  12  23  34  45  56  67  78   * |
          |11  22  33  44  55  66  77   *   * |
          |21  32  43  54  65  76   *   *   * |
          |31  42  53  64  75   *   *   *   * |
```

and where "*" means you do not store an element in that position in the array.

Following is an example of how to transform your general band matrix, for all values of *m* and *n*, to BLAS-general-band storage mode:

```
      DO 20 J=1,N
         K=MU+1-J
         DO 10 I=MAX(1,J-MU),MIN(M,J+ML)
            AGB(K+I,J)=A(I,J)
 10      CONTINUE
 20   CONTINUE
```

Symmetric Band Matrix

A symmetric band matrix is a symmetric matrix whose nonzero elements are arranged uniformly near the diagonal, such that:

$a_{ij} = 0 \quad \text{if } |i\text{-}j| > k$

where *k* is the half band width.

The following matrix illustrates a symmetric band matrix of order *n*, where the half band width *k* = *q*-1:

$$
\begin{array}{c}
\qquad\qquad |\leftarrow \; k \; \rightarrow| \\
A = \left[\begin{array}{cccccccccc}
a_{11} & a_{21} & a_{31} & . & . & a_{q1} & 0 & . & . & 0 \\
a_{21} & a_{22} & a_{32} & & & & & 0 & & . \\
a_{31} & a_{32} & a_{33} & & & & & & 0 & . \\
. & & & . & & & & & & 0 \\
. & & & & . & & & & & . \\
a_{q1} & & & & & . & & & & . \\
0 & & & & & & . & & & . \\
. & 0 & & & & & & . & & . \\
. & & 0 & & & & & & . & \\
0 & . & . & 0 & . & . & . & & & a_{nn}
\end{array}\right]
\end{array}
$$

Symmetric Band Matrix Storage Representation

The two storage modes used for storing symmetric band matrices are described in the following:

- "Upper-Band-Packed Storage Mode" on page 100
- "Lower-Band-Packed Storage Mode" on page 100

Upper-Band-Packed Storage Mode

Only the band elements of the upper triangular part of a symmetric band matrix, including the main diagonal, are stored for upper-band-packed storage mode.

For a matrix ***A*** of order *n* and a half band width of *k*, the array must have a leading dimension, *lda*, greater than or equal to *k*+1, and the size of the second dimension must be (at least) *n*.

Using array ASB, which is declared as ASB(*lda*,*n*), where *p* = *lda* = *k*+1, the elements of a symmetric band matrix are stored as follows:

$$\mathrm{ASB} = \left[\begin{array}{cccccccccc}
* & . & . & . & & * & a_{1p} & a_{2,p+1} & \cdot\;\cdot\;\cdot & a_{n-k,n} \\
. & & & & . & & & & & \\
. & & & . & & & & & & \\
. & & . & & & & & & & . \\
 & * & a_{13} & a_{24} & . & . & . & & & . \\
* & a_{12} & a_{23} & . & . & . & & & & . \\
a_{11} & a_{22} & . & . & . & & & & & a_{nn}
\end{array}\right]$$

where "*" means you do not store an element in that position in the array.

Following is an example of a symmetric band matrix ***A*** of order 6 and a half band width of 3.

Given the following matrix ***A***:

```
                    | 11  12  13  14   0   0 |
                    | 12  22  23  24  25   0 |
                    | 13  23  33  34  35  36 |
                    | 14  24  34  44  45  46 |
                    |  0  25  35  45  55  56 |
                    |  0   0  36  46  56  66 |
```

you store it in upper-band-packed storage mode in array ASB, declared as ASB(4,6), as follows.

```
                    |  *   *   *  14  25  36 |
              ASB = |  *   *  13  24  35  46 |
                    |  *  12  23  34  45  56 |
                    | 11  22  33  44  55  66 |
```

Following is an example of how to transform your symmetric band matrix to upper-band-packed storage mode:

```
       DO 20 J=1,N
          M=K+1-J
          DO 10 I=MAX(1,J-K),J
             ASB(M+I,J)=A(I,J)
       10 CONTINUE
    20 CONTINUE
```

Lower-Band-Packed Storage Mode

Only the band elements of the lower triangular part of a symmetric band matrix, including the main diagonal, are stored for lower-band-packed storage mode.

For a matrix ***A*** of order *n* and a half band width of *k*, the array must have a leading dimension, *lda*, greater than or equal to *k*+1, and the size of the second dimension must be (at least) *n*.

Using array ASB, which is declared as ASB(*lda*,*n*), where *q* = *lda* = *k*+1, the elements of a symmetric band matrix are stored as follows:

$$\text{ASB} = \begin{bmatrix} a_{11} & a_{22} & \cdots & & & a_{nn} \\ a_{21} & a_{32} & & & & * \\ a_{31} & a_{42} & & & & \cdot \\ \cdot & \cdot & & & & \cdot \\ \cdot & \cdot & & & & \cdot \\ \cdot & \cdot & & & & \\ a_{q1} & a_{q+1,2} & \cdots & a_{n,n-k} & * \; \cdots & * \end{bmatrix}$$

where "*" means you do not store an element in that position in the array.

Following is an example of a symmetric band matrix ***A*** of order 6 and a half band width of 2.

Given the following matrix ***A***:

```
                 | 11  21  31   0   0   0 |
                 | 21  22  32  42   0   0 |
                 | 31  32  33  43  53   0 |
                 |  0  42  43  44  54  64 |
                 |  0   0  53  54  55  65 |
                 |  0   0   0  64  65  66 |
```

you store it in lower-band-packed storage mode in array ASB, declared as `ASB(3,6)`, as follows:

```
                 | 11  22  33  44  55  66 |
           ASB = | 21  32  43  54  65   * |
                 | 31  42  53  64   *   * |
```

Following is an example of how to transform your symmetric band matrix to lower-band-packed storage mode:

```
      DO 20 J=1,N
         DO 10 I=J,MIN(J+K,N)
            ASB(I-J+1,J)=A(I,J)
      10 CONTINUE
   20 CONTINUE
```

Positive Definite Symmetric Band Matrix

A real symmetric band matrix ***A*** is positive definite if and only if $\boldsymbol{x}^{\mathrm{T}}\boldsymbol{Ax}$ is positive for all nonzero vectors ***x***.

Positive Definite Symmetric Band Matrix Storage Representation

The positive definite symmetric band matrix is stored in the same way a symmetric band matrix is stored. For a description of this storage technique, see "Symmetric Band Matrix" on page 99.

Complex Hermitian Band Matrix

A complex band matrix is Hermitian if it is equal to its conjugate transpose:

$\boldsymbol{H} = \boldsymbol{H}^{\mathrm{H}}$

Complex Hermitian Band Matrix Storage Representation

The complex Hermitian band matrix is stored using the same two techniques used for symmetric band matrices:

- Lower-band-packed storage mode, as described in "Lower-Band-Packed Storage Mode" on page 100
- Upper-band-packed storage mode, as described in "Upper-Band-Packed Storage Mode" on page 100

Following is an example of a complex Hermitian band matrix ***H*** of order 5, having a half band width of 2.

Given the following matrix ***H***:

```
| (11,  0) (21, -1) (31,  1)  (0,  0)  (0,  0) |
| (21,  1) (22,  0) (32, -1) (42, -1)  (0,  0) |
| (31, -1) (32,  1) (33,  0) (43, -1) (53, -1) |
|  (0,  0) (42,  1) (43,  1) (44,  0) (54, -1) |
|  (0,  0)  (0,  0) (53,  1) (54,  1) (55,  0) |
```

you store it in a two-dimensional array HP, as follows:

- In lower-band-packed storage mode:

```
     | (11,  *) (22, *) (33, *) (44,  *) (55, *) |
HP = | (21,  1) (32, 1) (43, 1) (54,  1)    *    |
     | (31, -1) (42, 1) (53, 1)    *        *    |
```

- In upper-band-packed storage mode:

```
       |    *        *      (31,  1) (42, -1) (53, -1) |
HP =   |    *     (21, -1) (32, -1) (43, -1) (54, -1) |
       | (11, *)  (22,  *) (33,  *) (44,  *) (55,  *) |
```

where "*" means you do not have to store a value in that position in the array. The imaginary parts of the diagonal elements of a complex Hermitian band matrix are always 0, so you do not need to set these values. The ESSL subroutines always assume that the values in these positions are 0.

Positive Definite Complex Hermitian Band Matrix

A complex Hermitian band matrix A is positive definite if and only if $\boldsymbol{x}^H\boldsymbol{Ax}$ is positive for all nonzero vectors $\boldsymbol{x}$.

Positive Definite Complex Hermitian Band Matrix Storage Representation

The positive definite complex Hermitian band matrix is stored in the same way a complex Hermitian band matrix is stored. For a description of this storage technique, see "Complex Hermitian Band Matrix" on page 101.

Triangular Band Matrix

There are two types of triangular band matrices: upper triangular band matrix and lower triangular band matrix. Triangular band matrices have the same number of rows as they have columns; that is, they have *n* rows and *n* columns. They have an upper or lower band width of *k*.

A band matrix ***U*** is an upper triangular band matrix if its nonzero elements are found only in the upper triangle of the matrix, including the main diagonal; that is:

$u_{ij} = 0 \quad \text{if } i > j$

Its band elements are arranged uniformly near the diagonal in the upper triangle of the matrix, such that:

$u_{ij} = 0 \quad \text{if } j\text{-}i > k$

The following matrix ***U*** illustrates an upper triangular band matrix of order *n* with an upper band width *k* = *q*-1:

$$\begin{array}{c} \quad\quad\quad\quad\quad \Big|\leftarrow k \rightarrow\Big| \end{array}$$

$$\boldsymbol{U} = \begin{bmatrix}
u_{11} & u_{21} & u_{31} & . & . & u_{q1} & 0 & . & . & 0 \\
0 & u_{22} & u_{32} & & & & & 0 & & . \\
. & 0 & u_{33} & & & & & & 0 & . \\
. & & & . & & & & & & 0 \\
. & & & & . & & & & & . \\
. & & & & & . & & & & . \\
. & & & & & & . & & & . \\
 & & & & & & & . & & \\
 & & & & & & & & . & \\
0 & . & . & . & & & & & 0 & u_{nn}
\end{bmatrix}$$

A band matrix ***L*** is a lower triangular band matrix if its nonzero elements are found only in the lower triangle of the matrix, including the main diagonal; that is:

$l_{ij} = 0 \quad \text{if } i < j$

Its band elements are arranged uniformly near the diagonal in the lower triangle of the matrix such that:

$l_{ij} = 0 \quad \text{if } i\text{-}j > k$

The following matrix ***L*** illustrates an upper triangular band matrix of order *n* with a lower band width *k* = *q*-1:

$$\boldsymbol{L} = \begin{array}{c}
 \\ - \\ \uparrow \\ k \\ \downarrow \\ - \\ \\ \\ \\ \\
\end{array}
\begin{bmatrix}
l_{11} & 0 & . & . & . & & & & & 0 \\
l_{21} & l_{22} & 0 & & & & & & & . \\
l_{31} & l_{32} & l_{33} & & & & & & & . \\
. & & & . & & & & & & . \\
. & & & & . & & & & & . \\
l_{q1} & & & & & . & & & & . \\
0 & & & & & & . & & & \\
. & 0 & & & & & & . & & \\
. & & 0 & & & & & & . & 0 \\
0 & . & . & 0 & . & . & . & & & l_{nn}
\end{bmatrix}$$

A triangular band matrix can also be a unit triangular band matrix if all the diagonal elements have a value of 1. For an illustration of a unit triangular matrix, see "Triangular Matrix" on page 88.

Triangular Band Matrix Storage Representation

The two storage modes used for storing triangular band matrices are described in the following:

- "Upper-Triangular-Band-Packed Storage Mode" on page 104
- "Lower-Triangular-Band-Packed Storage Mode" on page 105

It is important to note that because the diagonal elements of a unit triangular band matrix are always one, you do not need to set these values in the array for these two storage modes. ESSL always assumes that the values in these positions are one.

Upper-Triangular-Band-Packed Storage Mode

Only the band elements of the upper triangular part of an upper triangular band matrix, including the main diagonal, are stored for upper-triangular-band-packed storage mode.

For a matrix ***U*** of order *n* and an upper band width of *k*, the array must have a leading dimension, *lda*, greater than or equal to *k*+1, and the size of the second dimension must be (at least) *n*.

Using array UTB, which is declared as UTB(*lda*,*n*), where *p* = *lda* = *k*+1, the elements of an upper triangular band matrix are stored as follows:

$$\mathrm{UTB} = \left[\begin{array}{ccccccccc}
* & \cdot & \cdot & \cdot & * & u_{1p} & u_{2,p+1} & \cdots & u_{n-k,n} \\
\cdot & & & \cdot & & & & & \\
\cdot & & \cdot & & & & & & \\
\cdot & \cdot & & & & & & & \cdot \\
 & * & u_{13} & u_{24} & \cdots & & & & \cdot \\
* & u_{12} & u_{23} & \cdots & & & & & \cdot \\
u_{11} & u_{22} & \cdots & & & & & & u_{nn}
\end{array}\right]$$

where "*" means you do not store an element in that position in the array.

Following is an example of an upper triangular band matrix ***U*** of order 6 and an upper band width of 3.

Given the following matrix ***U***:

```
                     | 11  12  13  14   0   0 |
                     |  0  22  23  24  25   0 |
                     |  0   0  33  34  35  36 |
                     |  0   0   0  44  45  46 |
                     |  0   0   0   0  55  56 |
                     |  0   0   0   0   0  66 |
```

you store it in upper-triangular-band-packed storage mode in array UTB, declared as `UTB(4,6)`, as follows:

```
                     |  *   *   *  14  25  36 |
              UTB =  |  *   *  13  24  35  46 |
                     |  *  12  23  34  45  56 |
                     | 11  22  33  44  55  66 |
```

Following is an example of how to transform your upper triangular band matrix to upper-triangular-band-packed storage mode:

```
      DO 20 J=1,N
         M=K+1-J
         DO 10 I=MAX(1,J-K),J
            UTB(M+I,J)=U(I,J)
      10 CONTINUE
   20 CONTINUE
```

Lower-Triangular-Band-Packed Storage Mode

Only the band elements of the lower triangular part of a lower triangular band matrix, including the main diagonal, are stored for lower-triangular-band-packed storage mode.

Note: As an alternative to this storage mode, you can specify your arguments in your subroutine in a special way so that ESSL selects the matrix elements properly, and you can leave your matrix stored in full-matrix storage mode.

For a matrix ***L*** of order *n* and a lower band width of *k*, the array must have a leading dimension, *lda*, greater than or equal to *k*+1, and the size of the second dimension must be (at least) *n*.

Using array LTB, which is declared as LTB(*lda*,*n*), where *q* = *lda* = *k*+1, the elements of a lower triangular band matrix are stored as follows:

$$\mathrm{LTB} = \begin{bmatrix} l_{11} & l_{22} & \cdots & & & l_{nn} \\ l_{21} & l_{32} & & & & * \\ l_{31} & l_{42} & & & & . \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & \\ l_{q1} & l_{q+1,2} & \cdots & l_{n,n-k} & * \cdots & * \end{bmatrix}$$

where "*" means you do not store an element in that position in the array.

Following is an example of a lower triangular band matrix ***L*** of order 6 and a lower band width of 2.

Given the following matrix ***L***:

```
                      | 11   0   0   0   0   0 |
                      | 21  22   0   0   0   0 |
                      | 31  32  33   0   0   0 |
                      |  0  42  43  44   0   0 |
                      |  0   0  53  54  55   0 |
                      |  0   0   0  64  65  66 |
```

you store it in lower-triangular-band-packed storage mode in array LTB, declared as `LTB(3,6)`, as follows:

```
                      | 11  22  33  44  55  66 |
               LTB =  | 21  32  43  54  65   * |
                      | 31  42  53  64   *   * |
```

Following is an example of how to transform your lower triangular band matrix to lower-triangular-band-packed storage mode:

```
      DO 20 J=1,N
         M=1-J
         DO 10 I=J,MIN(N,J+K)
            LTB(M+I,J)=L(I,J)
      10 CONTINUE
   20 CONTINUE
```

General Tridiagonal Matrix

A general tridiagonal matrix is a matrix whose nonzero elements are found only on the diagonal, subdiagonal, and superdiagonal of the matrix; that is:

$a_{ij} = 0 \quad \text{if } |i-j| > 1$

The following matrix illustrates a general tridiagonal matrix of order *n*:

$$\boldsymbol{A} = \left[\begin{array}{cccccccc}
a_{11} & a_{12} & 0 & \cdot & \cdot & \cdot & & 0 \\
a_{21} & a_{22} & a_{23} & 0 & & & & \cdot \\
0 & a_{32} & a_{33} & a_{34} & 0 & & & \cdot \\
\cdot & 0 & a_{43} & a_{44} & \cdot & \cdot & & \cdot \\
\cdot & & 0 & \cdot & \cdot & \cdot & \cdot & \\
\cdot & & & \cdot & \cdot & \cdot & \cdot & \cdot \\
 & & & & \cdot & \cdot & \cdot & \cdot \\
0 & \cdot & \cdot & \cdot & & \cdot & \cdot & a_{nn}
\end{array}\right]$$

General Tridiagonal Matrix Storage Representation

The storage modes used for storing trapezoidal matrices are described in the following:

- "LAPACK-General Tridiagonal Storage Mode" on page 106
- "General Tridiagonal Storage Mode" on page 107

LAPACK-General Tridiagonal Storage Mode

This storage mode is for use with LAPACK compatible tridiagonal subroutines.

Only the diagonal, subdiagonal, and superdiagonal elements of the general tridiagonal matrix are stored for LAPACK-general-tridiagonal storage mode. The diagonal elements of a general tridiagonal matrix, ***A***, of order *n* are stored in a one-dimensional array D of length *n*.

The subdiagonal and superdiagonal elements of a general tridiagonal matrix ***A*** of order *n* are stored in one dimensional arrays DL and DU of length *n*-1, respectively. DL, D, and DU are stored as follows:

$$\begin{aligned}
\text{DL} &= (a_{21},\ a_{32},\ a_{43}\ \ldots a_{n,n-1}) \\
\text{D} &= (a_{11},\ a_{22},\ a_{33}\ \ldots a_{n,n}) \\
\text{DU} &= (a_{12},\ a_{23},\ a_{34}\ \ldots a_{n-1,n})
\end{aligned}$$

Following is an example of a general tridiagonal matrix ***A*** of order 5:

```
                         | 11  12   0   0   0 |
                         | 21  22  23   0   0 |
                         |  0  32  33  34   0 |
                         |  0   0  43  44  45 |
                       |  0   0   0  54  55 |
```

which you store in LAPACK-general tridiagonal storage mode in arrays DL, D, and DU, as follows:

```
                        DL = (21, 32, 43, 54)

                        D = (11, 22, 33, 44, 55)

                        DU = (12, 23, 34, 45)
```

General Tridiagonal Storage Mode

This storage mode is for use with non-LAPACK compatible tridiagonal subroutines.

Only the diagonal, subdiagonal, and superdiagonal elements of the general tridiagonal matrix are stored. This is called tridiagonal storage mode. The elements of a general tridiagonal matrix, ***A***, of order *n* are stored in three one-dimensional arrays, C, D, and E, each of length *n*, where array C contains the subdiagonal elements, stored as follows:

C = ($*$, a_{21}, a_{32}, a_{43}, ..., $a_{n,n\text{-}1}$)

and array D contains the main diagonal elements, stored as follows:

D = (a_{11}, a_{22}, a_{33}, ..., a_{nn})

and array E contains the superdiagonal elements, stored as follows:

E = (a_{12}, a_{23}, a_{34}, ..., $a_{n\text{-}1,n}$, $*$)

where "*" means you do not store an element in that position in the array.

Following is an example of a general tridiagonal matrix ***A*** of order 5:

```
                       | 11  12   0   0   0 |
                       | 21  22  23   0   0 |
                       |  0  32  33  34   0 |
                       |  0   0  43  44  45 |
                       |  0   0   0  54  55 |
```

which you store in tridiagonal storage mode in arrays C, D, and E, each of length 5, as follows:

```
                        C = (*, 21, 32, 43, 54)

                        D = (11, 22, 33, 44, 55)

                        E = (12, 23, 34, 45, *)
```

Note: Some ESSL subroutines provide an option for specifying at least *n* additional locations at the end of each of the arrays C, D, and E. These additional locations are used for working storage by the ESSL subroutine. The reasons for choosing this option are explained in the subroutine descriptions.

Symmetric Tridiagonal Matrix

A tridiagonal matrix ***A*** is also symmetric if and only if its nonzero elements are found only on the diagonal, subdiagonal, and superdiagonal of the matrix, and its subdiagonal elements and superdiagonal elements are equal; that is:

($a_{ij} = 0$ if $|i\text{-}j| > 1$)   and   ($a_{ij} = a_{ji}$ if $|i\text{-}j| = 1$)

The following matrix illustrates a symmetric tridiagonal matrix of order *n*:

$$\boldsymbol{A} = \left[\begin{array}{cccccccc}
a_{11} & a_{21} & 0 & \cdot & \cdot & \cdot & & 0 \\
a_{21} & a_{22} & a_{32} & 0 & & & & \cdot \\
0 & a_{32} & a_{33} & a_{43} & 0 & & & \cdot \\
\cdot & 0 & a_{43} & a_{44} & \cdot & \cdot & & \cdot \\
\cdot & & 0 & \cdot & \cdot & \cdot & \cdot & \\
\cdot & & & \cdot & \cdot & \cdot & \cdot & \cdot \\
 & & & & \cdot & \cdot & \cdot & \cdot \\
0 & \cdot & \cdot & \cdot & & \cdot & \cdot & a_{nn}
\end{array}\right]$$

Symmetric Tridiagonal Matrix Storage Representation

The two storage modes used for storing symmetric tridiagonal matrices are described in the following:

- "LAPACK-Symmetric-Tridiagonal Storage Mode" on page 108
- "Symmetric-Tridiagonal Storage Mode" on page 108

LAPACK-Symmetric-Tridiagonal Storage Mode

This storage mode is for use with LAPACK compatible tridiagonal subroutines.

Only the diagonal and subdiagonal elements of the symmetric tridiagonal matrix are stored for LAPACK-symmetric-tridiagonal storage mode. The diagonal elements of a symmetric tridiagonal matrix ***A*** of order *n* are stored in a one dimensional array D length *n*. The subdiagonal elements of a symmetric matrix ***A*** are stored in a one dimensional array E of length *n*-1. D and E are stored as follows:

D = (a_{11}, a_{22}, a_{33}, ..., a_{nn})
E = (a_{21}, a_{32}, a_{43}, ..., $a_{n,n-1}$)

Following is an example of a symmetric tridiagonal matrix ***A*** of order 5:

```
                        | 10   1   0   0   0 |
                        |  1  20   2   0   0 |
                        |  0   2  30   3   0 |
                        |  0   0   3  40   4 |
                        |  0   0   0   4  50 |
```

which you store in LAPACK-symmetric-tridiagonal storage mode in arrays D and E, each of length 4, as follows:

```
                         D = (10, 20, 30, 40, 50)
                         E = (1, 2, 3, 4)
```

Symmetric-Tridiagonal Storage Mode

This storage mode is for use with non-LAPACK compatible tridiagonal subroutines.

Only the diagonal and subdiagonal elements of the symmetric tridiagonal matrix are stored for symmetric-tridiagonal storage mode. The elements of a symmetric tridiagonal matrix ***A*** of order *n* are stored in two one dimensional arrays C and D, each of length *n*, where C contains the subdiagonal elements, stored as follows:

C = (*, a_{21}, a_{32}, a_{43}, ..., $a_{n,n-1}$)

where "*" means you do not store an element in that position in the array. Then array D contains the main diagonal elements, stored as follows:

D = (a_{11}, a_{22}, a_{33}, ..., a_{nn})

Following is an example of a symmetric tridiagonal matrix ***A*** of order 5:

```
| 10   1   0   0   0 |
|  1  20   2   0   0 |
|  0   2  30   3   0 |
|  0   0   3  40   4 |
|  0   0   0   4  50 |
```

which you store in symmetric-tridiagonal storage mode in arrays C and D, each of length 5, as follows:

```
C = (*, 1, 2, 3, 4)

D = (10, 20, 30, 40, 50)
```

Note: Some ESSL subroutines provide an option for specifying at least *n* additional locations at the end of each of the arrays C and D. These additional locations are used for working storage by the ESSL subroutine. The reasons for choosing this option are explained in the subroutine descriptions.

Positive Definite Symmetric Tridiagonal Matrix

A real symmetric tridiagonal matrix ***A*** is positive definite if and only if $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x}$ is positive for all nonzero vectors ***x***.

Positive Definite Symmetric Tridiagonal Matrix Storage Representation

The positive definite symmetric tridiagonal matrix is stored in the same way the symmetric tridiagonal matrix is stored. For a description of this storage technique, see "Symmetric Tridiagonal Matrix" on page 107.

Complex Hermitian Tridiagonal Matrix

A complex tridiagonal matrix is Hermitian if it is equal to its conjugate transpose: $\boldsymbol{H} = \boldsymbol{H}^H$.

Complex Hermitian Tridiagonal Storage Representation

Only the diagonal and subdiagonal elements of the complex Hermitian tridiagonal matrix are stored for LAPACK-complex-Hermitian-tridiagonal storage mode. The diagonal elements of a complex Hermitian tridiagonal matrix ***A*** of order *n* are stored in a one dimensional array D of length *n*. The subdiagonal elements of a complex Hermitian matrix ***A*** are stored in a one dimensional array E of length *n*-1. D and E are stored as follows:

D = (a_{11}, a_{22}, a_{33}, ..., a_{nn})
E = (*, a_{21}, a_{32}, a_{43}, ..., $a_{n,n-1}$)

Following is an example of a symmetric tridiagonal matrix ***A*** of order 5:

```
┌                                             ┐
| (10,  0) ( 1,  1) ( 1,  2) ( 1,  3) ( 1,  4) |
| ( 1, -1) (20,  0) ( 2,  1) ( 2,  2) ( 2,  3) |
| ( 1, -2) ( 2, -1) (30,  0) ( 3,  1) ( 3,  2) |
| ( 1, -3) ( 2, -2) ( 3, -1) (40,  0) ( 4,  1) |
| ( 1, -4) ( 2, -3) ( 3, -2) ( 4, -1) (50,  0) |
└                                             ┘
```

which you store in LAPACK-complex-Hermitian-tridiagonal storage mode in arrays D of length 5 and complex array E, each of length 4, as follows:

```
D = (10, 20, 30, 40, 50)
E = ((1,-1), (2, -1), (3, -1), (4, -1))
```

Postive Definite Complex Hermitian Tridiagonal Matrix

A complex Hermitian tridiagonal matrix is positive definite if and only if x^H***A****x* is positive for all nonzero vectors *x*.

Positive Definite Complex Hermitian Tridiagonal Matrix Storage Representation

The positive definite complex Hermitian tridiagonal matrix is stored in the same way a complex Hermitian tridiagonal matrix is stored. For a description of this storage technique, see "Complex Hermitian Tridiagonal Matrix" on page 109.

Sparse Matrix

A sparse matrix is a matrix having a relatively small number of nonzero elements.

Consider the following as an example of a sparse matrix ***A***:

```
                    | 11   0  13   0   0   0 |
                    | 21  22   0  24   0   0 |
                    |  0  32  33   0  35   0 |
                    |  0   0  43  44   0  46 |
                    | 51   0   0  54  55   0 |
                    | 61  62   0   0  65  66 |
```

Sparse Matrix Storage Representation

A sparse matrix can be stored in full-matrix storage mode or a packed storage mode. When a sparse matrix is stored in **full-matrix storage mode**, all its elements, including its zero elements, are stored in an array.

The seven packed storage modes used for storing sparse matrices are described in the following:

- "Compressed-Matrix Storage Mode" on page 110
- "Compressed-Diagonal Storage Mode" on page 111
- "Storage-by-Indices" on page 114
- "Storage-by-Columns" on page 114
- "Storage-by-Rows" on page 115
- "Diagonal-Out Skyline Storage Mode" on page 117
- "Profile-In Skyline Storage Mode" on page 119

Note: When the elements of a sparse matrix are stored using any of these storage modes, the ESSL subroutines do not check that all elements are nonzero. You do not get an error if any elements are zero.

Compressed-Matrix Storage Mode

The sparse matrix ***A***, stored in compressed-matrix storage mode, uses two two-dimensional arrays to define the sparse matrix storage, AC and KA. See reference ["89" on page 1258]. Given the *m* by *n* sparse matrix ***A***, having a maximum of *nz* nonzero elements in each row:

- AC is defined as AC(*lda*,*nz*), where the leading dimension, *lda*, must be greater than or equal to *m*. Each row of array AC contains the nonzero elements of the corresponding row of matrix ***A***. For each row in matrix ***A*** containing less than *nz* nonzero elements, the corresponding row in array AC is padded with zeros. The elements in each row can be stored in any order.
- KA is an integer array defined as KA(*lda*,*nz*), where the leading dimension, *lda*, must be greater than or equal to *m*. It contains the column numbers of the matrix ***A*** elements that are stored in the corresponding positions in array AC. For each row in matrix ***A*** containing less than *nz* nonzero elements, the corresponding row in array KA is padded with any values from 1 to *n*. **Because this array is used by the ESSL subroutines to access other target vectors in the computation, you must adhere to these required values to avoid errors.**

Unless all the rows of sparse matrix *A* contain approximately the same number of nonzero elements, this storage mode requires a large amount of storage. This diminishes the performance you can obtain by using this storage mode.

Consider the following as an example of a 6 by 6 sparse matrix ***A*** with a maximum of four nonzero elements in each row. It shows how matrix ***A*** can be stored in arrays AC and KA.

Given the following matrix ***A***:

```
                     | 11   0  13   0   0   0 |
                     | 21  22   0  24   0   0 |
                     |  0  32  33   0  35   0 |
                     |  0   0  43  44   0  46 |
                     | 51   0   0  54  55   0 |
                     | 61  62   0   0  65  66 |
```

the arrays are:

```
                   | 11  13   0   0 |
                   | 22  21  24   0 |
            AC =   | 33  32  35   0 |
                   | 44  43  46   0 |
                   | 55  51  54   0 |
                   | 66  61  62  65 |
```

```
                     | 1  3  *  * |
                     | 2  1  4  * |
              KA =   | 3  2  5  * |
                     | 4  3  6  * |
                     | 5  1  4  * |
                     | 6  1  2  5 |
```

where "*" means you can store any value from 1 to 6 in that position in the array.

Symmetric sparse matrices use the same storage technique as nonsymmetric sparse matrices; that is, all nonzero elements of a symmetric matrix ***A*** must be stored in array AC, not just the elements of the upper triangle and diagonal of matrix ***A***.

In general terms, this storage technique can be expressed as follows:

For each $a_{ij} \neq 0$, for $i = 1, m$ and $j = 1, n$
there exists k, where $1 \leq k \leq nz$,
such that AC(i,k) = a_{ij} and KA(i,k) = j.

For all other elements of AC and KA,
AC(i,k) ≤ n

where:

- a_{ij} are the elements of the m by n matrix ***A*** that has a maximum of nz nonzero elements in each row.
- Array AC is defined as AC(lda,nz), where $lda \geq m$.
- Array KA is defined as KA(lda,nz), where $lda \geq m$.

Compressed-Diagonal Storage Mode

The storage mode used for square sparse matrices stored in compressed-diagonal storage mode has two variations, depending on whether the matrix is a general sparse matrix or a symmetric sparse matrix. This explains both of these variations; however, the conventions used for numbering the diagonals in the matrix, which apply to the storage descriptions, are explained first.

Matrix ***A*** of order *n* has 2*n*-1 diagonals. Because $k = j-i$ is constant for the elements a_{ij} along each diagonal, each diagonal can be assigned a diagonal number, *k*, having a value from 1-*n* to *n*-1. Then the diagonals can be referred to as $\boldsymbol{d}_k$, where $k = 1-n, n-1$.

The following matrix shows the starting position of each diagonal, $\boldsymbol{d}_k$:

$$\boldsymbol{A} = \begin{array}{c|ccccccc|}
 & \boldsymbol{d}_0 & \boldsymbol{d}_1 & \boldsymbol{d}_2 & \cdot & \cdot & \cdot & \boldsymbol{d}_{n-1} \\
 & a_{11} & a_{12} & a_{13} & \cdot & \cdot & \cdot & a_{1n} \\
\boldsymbol{d}_{-1} & a_{21} & a_{22} & a_{23} & & & & \cdot \\
\boldsymbol{d}_{-2} & a_{31} & a_{32} & a_{33} & & & & \cdot \\
\cdot & \cdot & & & \cdot & & & \cdot \\
\cdot & \cdot & & & & \cdot & & \cdot \\
\cdot & \cdot & & & & & \cdot & \cdot \\
\boldsymbol{d}_{1-n} & a_{n1} & \cdot & \cdot & \cdot & \cdot & \cdot & a_{nn}
\end{array}$$

For a **general** (square) sparse matrix ***A***, compressed-diagonal storage mode uses two arrays to define the sparse matrix storage, AD and LA. Using the above convention for numbering the diagonals, and given that sparse matrix ***A*** contains *nd* diagonals having nonzero elements, arrays AD and LA are set up as follows:

- AD is defined as AD(*lda*,*nd*), where the leading dimension, *lda*, must be greater than or equal to *n*. Each diagonal of matrix ***A*** that has at least one nonzero element is stored in a column of array AD. All of the elements of the diagonal, including its zero elements, are stored in *n* contiguous locations in the array, in the same order as they appear in the diagonal. Padding with zeros is required as follows to fill the *n* locations in each column of array AD:
  - Each superdiagonal ($k > 0$), which has *n*-*k* elements, is padded with *k* trailing zeros.
  - The main diagonal ($k = 0$), which has *n* elements, does not require padding.
  - Each subdiagonal ($k < 0$), which has *n*-|*k*| elements, is padded with |*k*| leading zeros.

  The diagonals can be stored in any columns in array AD.
- LA is a one-dimensional integer array of length *nd*, containing the diagonal numbers *k* for the diagonals stored in each corresponding column in array AD.

Because this storage mode requires entire diagonals to be stored, if the nonzero elements in matrix *A* are not concentrated along a few diagonals, this storage mode requires a large amount of storage. This diminishes the performance you obtain by using this storage mode.

Consider the following as an example of how a 6 by 6 general sparse matrix ***A*** with 5 nonzero diagonals is stored in arrays AD and LA.

Given the following matrix ***A***:

```
                     | 11   0  13   0   0   0 |
                     | 21  22   0  24   0   0 |
                     |  0  32  33   0  35   0 |
                     |  0   0  43  44   0  46 |
                     | 51   0   0  54  55   0 |
                     | 61  62   0   0  65  66 |
```

the arrays are:

```
                           | 11  13   0   0   0 |
                           | 22  24  21   0   0 |
                 AD   =    | 33  35  32   0   0 |
                           | 44  46  43   0   0 |
                    | 55   0  54  51   0 |
                    | 66   0  65  62  61 |


     LA    =  (0, 2, -1, -4, -5)
```

For a **symmetric** sparse matrix, where each superdiagonal *k* is equal to subdiagonal -*k*, compressed-diagonal storage mode uses the same storage technique as for the general sparse matrix, except that only the nonzero main diagonal and one diagonal of each couple of nonzero diagonals, *k* and -*k*, are used in setting up arrays AD and LA. You can store either the upper or the lower diagonal of each couple.

Consider the following as an example of a symmetric sparse matrix of order 6 and how it is stored in arrays AD and LA, using only three nonzero diagonals in the matrix.

Given the following matrix ***A***:

```
      | 11   0  13   0  51   0 |
      |  0  22   0  24   0  62 |
      | 13   0  33   0  35   0 |
      |  0  24   0  44   0  46 |
      | 51   0  35   0  55   0 |
      |  0  62   0  46   0  66 |
```

the arrays are:

```
               | 11  13   0 |
               | 22  24   0 |
       AD   =  | 33  35   0 |
               | 44  46   0 |
               | 55   0  51 |
               | 66   0  62 |


        LA   = (0, 2, -4)
```

In general terms, this storage technique can be expressed as follows:

For each $\boldsymbol{d}_k \neq (0, ..., 0)$, for $k = 1\text{-}n, n\text{-}1$
for **general** square sparse matrices, or

for each unique $\boldsymbol{d}_k \neq (0, ..., 0)$, for $k = 1\text{-}n, n\text{-}1$
for **symmetric** sparse matrices,

there exists *l*, where $1 \leq l \leq nd$,
such that LA(*l*) = *k* and column *l* in array AD contains $\boldsymbol{dp}_k$.

where:

- Array AD is defined as AD(*lda*,*nd*), where $lda \geq n$, and where *nd* is the number of nonzero diagonals, $\boldsymbol{d}_k$ that are stored in array AD.
- Array LA has *nd* elements.
- *k* is the diagonal number of each diagonal, $\boldsymbol{d}_k$, where $k = i\text{-}j$.
- $\boldsymbol{dp}_k$ are the diagonals, $\boldsymbol{d}_k$, with padding, which are constructed from the sparse matrix ***A*** elements, a_{ij}, for *i*, *j* = 1, *n* as follows:

  For superdiagonals ($k > 0$), $\boldsymbol{dp}_k$ has *k* trailing zeros: $\boldsymbol{dp}_k = (a_{1,k+1}, a_{2,k+2}, ..., a_{n\text{-}k,n}, 0_1, ..., 0_k)$

  For the main diagonal ($k = 0$), $\boldsymbol{dp}_0$ has no padding: $\boldsymbol{dp}_0 = (a_{11}, a_{22}, ..., a_{nn})$

  For subdiagonals ($k < 0$), $\boldsymbol{dp}_k$ has $|k|$ leading zeros: $\boldsymbol{dp}_k = (0_1, ..., 0_{|k|}, a_{|k|+1,1}, a_{|k|+2,2}, ..., a_{n,\, n\text{-}|k|})$

Storage-by-Indices

For a sparse matrix ***A***, storage-by-indices uses three one-dimensional arrays to define the sparse matrix storage, AR, IA, and JA. Given the *m* by *n* sparse matrix ***A*** having *ne* nonzero elements, the arrays are set up as follows:

- AR of (at least) length *ne* contains the *ne* nonzero elements of the sparse matrix ***A***, stored contiguously in **any** order.
- IA, an integer array of (at least) length *ne* contains the corresponding row numbers of each nonzero element, a_{ij}, in matrix ***A***.
- JA, an integer array of (at least) length *ne* contains the corresponding column numbers of each nonzero element, a_{ij}, in matrix ***A***.

Consider the following as an example of a 6 by 6 sparse matrix ***A*** and how it can be stored in arrays AR, IA, and JA.:

Given the following matrix ***A***:

```
┌                        ┐
| 11   0  13   0   0   0 |
| 21  22   0  24   0   0 |
|  0  32  33   0  35   0 |
|  0   0  43  44   0  46 |
|  0   0   0   0   0   0 |
| 61  62   0   0  65  66 |
└                        ┘
```

the arrays are:

```
AR = (11, 22, 32, 33, 13, 21, 43, 24, 66, 46, 35, 62, 61, 65, 44)

IA = (1, 2, 3, 3, 1, 2, 4, 2, 6, 4, 3, 6, 6, 6, 4)

JA = (1, 2, 2, 3, 3, 1, 3, 4, 6, 6, 5, 2, 1, 5, 4)
```

In general terms, this storage technique can be expressed as follows:

For each $a_{ij} \neq 0$, for $i = 1, m$ and $j = 1, n$
there exists k, where $1 \leq k \leq ne$, such that:

AR(k) = a_{ij}
IA(k) = i
JA(k) = j

where:

a_{ij} are the elements of the *m* by *n* sparse matrix ***A***.
Arrays AR, IA, and JA each have *ne* elements.

Storage-by-Columns

For a sparse matrix, ***A***, storage-by-columns uses three one-dimensional arrays to define the sparse matrix storage, AR, IA, and JA. Given the *m* by *n* sparse matrix ***A*** having *ne* nonzero elements, the arrays are set up as follows:

- AR of (at least) length *ne* contains the *ne* nonzero elements of the sparse matrix ***A***, stored contiguously. The columns of matrix ***A*** are stored consecutively from 1 to *n* in AR. The elements in each column of ***A*** are stored in any order in AR.
- IA, an integer array of (at least) length *ne* contains the corresponding row numbers of each nonzero element, a_{ij}, in matrix ***A***.
- JA, an integer array of (at least) length *n*+1 contains the relative starting position of each column of matrix ***A*** in array AR; that is, each element JA(j) of the column pointer array indicates where column j begins in array AR. If all elements in column j are zero, then JA(j) = JA(j+1). The last element, JA(n+1), indicates the position after the last element in array AR, which is *ne*+1.

Consider the following as an example of a 6 by 6 sparse matrix ***A*** and how it can be stored in arrays AR, IA, and JA.

Given the following matrix ***A***:

```
                    ┌                        ┐
                    | 11   0  13   0  0   0 |
                    | 21  22   0  24  0   0 |
                    |  0  32  33   0  0   0 |
                    |  0   0  43  44  0  46 |
                    |  0   0   0   0  0   0 |
                    | 61  62   0   0  0  66 |
                    └                        ┘
```

the arrays are:

```
    AR = (11, 61, 21, 62, 32, 22, 13, 33, 43, 44, 24, 46, 66)

    IA = (1, 6, 2, 6, 3, 2, 1, 3, 4, 4, 2, 4, 6)

    JA = (1, 4, 7, 10, 12, 12, 14)
```

In general terms, this storage technique can be expressed as follows:

For each $a_{ij} \neq 0$, for $i = 1, m$ and $j = 1, n$
there exists k, where $1 \leq k \leq ne$, such that

AR(k) = a_{ij}
IA(k) = i

And for $j = 1, n$,
JA(j) = k, where a_{ij}, in AR(k), is the first element stored in AR for column j
JA(j) = JA(j+1), where all $a_{ij} = 0$ in column j
JA(n+1) = ne+1

where:

a_{ij} are the elements of the m by n sparse matrix ***A***.
Arrays AR and IA each have ne elements.
Array JA has n+1 elements.

Storage-by-Rows

The storage mode used for sparse matrices stored by rows has three variations, depending on whether the matrix is a general sparse matrix or a symmetric sparse matrix. This explains these variations.

For a **general** sparse matrix ***A***, storage-by-rows uses three one-dimensional arrays to define the sparse matrix storage, AR, IA, and JA. Given the m by n sparse matrix ***A*** having ne nonzero elements, the arrays are set up as follows:

- AR of (at least) length ne contains the ne nonzero elements of the sparse matrix ***A***, stored contiguously. The rows of matrix ***A*** are stored consecutively from 1 to m in AR. The elements in each row of ***A*** are stored in any order in AR.
- IA, an integer array of (at least) length m+1 contains the relative starting position of each row of matrix ***A*** in array AR; that is, each element IA(i) of the row pointer array indicates where row i begins in array AR. If all elements in row i are zero, then IA(i) = IA(i+1). The last element, IA(m+1), indicates the position after the last element in array AR, which is ne+1.
- JA, an integer array of (at least) length ne contains the corresponding column numbers of each nonzero element, a_{ij}, in matrix ***A***.

Consider the following as an example of a 6 by 6 general sparse matrix ***A*** and how it can be stored in arrays AR, IA, and JA.

Given the following matrix ***A***:

```
                         ┌                        ┐
                         | 11   0  13   0  0   0 |
                         | 21  22   0  24  0   0 |
                         |  0  32  33   0  0   0 |
                         |  0   0  43  44  0  46 |
                         |  0   0   0   0  0   0 |
                         | 61  62   0   0  0  66 |
                         └                        ┘
```

the arrays are:

```
AR = (11, 13, 24, 22, 21, 32, 33, 44, 43, 46, 61, 62, 66)

IA = (1, 3, 6, 8, 11, 11, 14)

JA = (1, 3, 4, 2, 1, 2, 3, 4, 3, 6, 1, 2, 6)
```

For a **symmetric** sparse matrix of order *m*, storage-by-rows uses the same storage technique as for the general sparse matrix, except that only the upper or lower triangle and diagonal elements are used in setting up arrays AR, IA, and JA.

Consider the following as an example of a symmetric sparse matrix ***A*** of order 6 and how it can be stored in arrays AR, IA, and JA using upper-storage-by-rows, which stores only the upper triangle and diagonal elements.

Given the following matrix A:

```
                         ┌                         ┐
                         | 11   0  13   0   0   0 |
                         |  0  22  23  24   0   0 |
                         | 13  23  33   0  35   0 |
                         |  0  24   0  44   0  46 |
                         |  0   0  35   0  55   0 |
                         |  0   0   0  46   0   0 |
                         └                         ┘
```

the arrays are:

```
AR = (11, 13, 22, 24, 23, 33, 35, 46, 44, 55)

IA = (1, 3, 6, 8, 10, 11, 11)

JA = (1, 3, 2, 3, 4, 3, 5, 4, 6, 5)
```

Using the same symmetric matrix ***A***, consider the following as an example of how it can be stored in arrays AR, IA, and JA using lower-storage-by-rows, which stores only the lower triangle and diagonal elements:

```
AR = (11, 22, 23, 33, 13, 24, 44, 55, 35, 46)

IA = (1, 2, 3, 6, 8, 10, 11 )

JA = (1, 2, 2, 3, 1, 2, 4, 5, 3, 4)
```

In general terms, this storage technique can be expressed as follows:

For each $a_{ij} \neq 0$,
for $i = 1, m$ and $j = 1, n$ for general sparse matrices
or
for $i = 1, m$ and $j = i, m$ for symmetric sparse matrices using the lower triangle
or
for $i = 1, m$ and $j = 1, i$ for symmetric sparse matrices using the upper triangle
there exists k, where $1 \leq k \leq ne$, such that
AR(k) = a_{ij}
JA(k) = j

And for i = 1, m,
IA(i) = k, where a_{ij}, in AR(k), is the first element stored in AR for row i
IA(i) = IA(i+1), where all a_{ij} = 0 in row i
IA(m+1) = ne+1

where:

- a_{ij} are the elements of sparse matrix ***A***, which is either an m by n general sparse matrix or a symmetric sparse matrix of order m containing ne nonzero elements.
- Arrays AR and JA each have ne elements.
- Array IA has m+1 elements.

Diagonal-Out Skyline Storage Mode

The diagonal-out skyline storage mode used for sparse matrices has two variations, depending on whether the matrix is a general sparse matrix or a symmetric sparse matrix. Both of these variations are explained here.

For a **general** sparse matrix ***A***, diagonal-out skyline storage mode uses four one-dimensional arrays to define the sparse matrix storage, AU, IDU, AL, and IDL. Given the sparse matrix ***A*** of order n, containing nu+nl-n elements under the top and left profiles, the arrays are set up as follows:

- AU of (at least) length nu contains the upper triangle of the sparse matrix ***A***, where the columns are stored consecutively from 1 to n in AU in the following way. For each column, the elements starting at the diagonal element and ending at the topmost nonzero element in the column are stored contiguously in AU. The elements stored may include zero elements along with the nonzero elements. If all elements in the column to be stored are zero, the diagonal element, a_{ii}, having a value of zero, is stored in AU for that column. A total of nu elements are stored for the upper triangle of ***A***.
- IDU, an integer array of (at least) length n+1 contains the relative position of each diagonal element of matrix ***A*** in array AU; that is, each element IDU(i) of the diagonal pointer array indicates where diagonal element a_{ii} is stored in array AU. One-origin is used, so the first element of IDU is always 1. The last element, IDU(n+1), indicates the position after the last element in array AU, which is nu+1.
- AL of (at least) length nl contains the lower triangle of the sparse matrix ***A***, where the rows are stored consecutively from 1 to n in AL in the following way. For each row, the elements starting at the diagonal element and ending at the leftmost nonzero element in the row are stored contiguously in AL. The elements stored may include zero elements along with the nonzero elements. If all elements in the row to be stored are zero, the diagonal element, a_{ii}, having a value of zero, is stored in AL for that row. A total of nl elements are stored for the lower triangle of ***A***. The values of the diagonal elements are meaningless, so you can store any values in those positions in AL.
- IDL, an integer array of (at least) length n+1 contains the relative position of each diagonal element of matrix ***A*** in array AL; that is, each element IDL(i) of the diagonal pointer array indicates where diagonal element a_{ii} is stored in array AL. One-origin is used, so the first element of IDL is always 1. The last element, IDL(n+1), indicates the position after the last element in array AL, which is nl+1.

Consider the following as an example of a 6 by 6 general sparse matrix ***A*** and how it is stored in arrays AU, IDU, AL, and IDL.

Given the following matrix ***A***:

```
                    |  0  12  13   0   0   0 |
                    | 21  22   0  24   0   0 |
                    | 31   0  33  34   0  36 |
                    | 41  42  43  44  45   0 |
                    |  0   0   0  54  55  56 |
                    |  0   0  63   0  65  66 |
```

the arrays are:

```
    AU = (0, 22, 12, 33, 0, 13, 44, 34, 24, 55, 45, 66, 56, 0, 36)
```

`IDU = (1, 2, 4, 7, 10, 12, 16)` where *nu*=15

```
    AL = (*, *, 21, *, 0, 31, *, 43, 42, 41, *, 54, *, 65, 0, 63)
```

`IDL = (1, 2, 4, 7, 11, 13, 17)` where *nl*=16

and where "*" means you do not have to store a value in that position in the array. However, these storage positions are required.

For a **symmetric** sparse matrix of order *n*, diagonal-out skyline storage mode uses the same storage technique as for the upper triangle and diagonal elements of the general sparse matrix; therefore, only the AU and IDU arrays are needed.

Consider the following as an example of a symmetric sparse matrix ***A*** of order 6 and how it is stored in arrays AU and IDU.

Given the following matrix ***A***:

```
                     |  0  12  13   0   0   0 |
                     | 12  22   0  24   0   0 |
                     | 13   0  33  34   0  36 |
                     |  0  24  34  44  45   0 |
                     |  0   0   0  45  55  56 |
                     |  0   0  36   0  56  66 |
```

the arrays are:

```
    AU = (0, 22, 12, 33, 0, 13, 44, 34, 24, 55, 45, 66, 56, 0, 36)
```

`IDU = (1, 2, 4, 7, 10, 12, 16)` where *nu*=15

In general terms, this storage technique can be expressed as follows:

For general sparse matrices and symmetric sparse matrices:

For each a_{ij} for $j = 1, n$ and $i = j, k$,
where a_{kj} is the topmost $a_{ij} \neq 0$ in each column j,
there exists m, where $1 \leq m \leq nu$, such that
AU($m+j-i$) = a_{ij}
IDU(j) = m for each a_{jj}
IDU(n+1) = nu+1

Also, for general sparse matrices:
For each a_{ij} for $i = 1, n$ and $i = j, k$,
where a_{ik} is the leftmost $a_{ij} \neq 0$ in each row i,
there exists m, where $1 \leq m \leq nl$, such that
AL($m+i-j$) = a_{ij}
IDL(i) = m for each a_{ii}
IDL(n+1) = nl+1

where:

a_{ij} are the elements of sparse matrix ***A***, of order n.
Array AU has *nu* elements.
Array AL has *nl* elements.
Arrays IDU and IDL each have n+1 elements.

Profile-In Skyline Storage Mode

The profile-in skyline storage mode used for sparse matrices has two variations, depending on whether the matrix is a general sparse matrix or a symmetric sparse matrix. Both of these variations are explained here.

For a **general** sparse matrix ***A***, profile-in skyline storage mode uses four one-dimensional arrays to define the sparse matrix storage, AU, IDU, AL, and IDL. Given the sparse matrix ***A*** of order *n*, containing *nu*+*nl*-*n* elements under the top and left profiles, the arrays are set up as follows:

- AU of (at least) length *nu* contains the upper triangle of the sparse matrix ***A***, where the columns are stored consecutively from 1 to *n* in AU in the following way. For each column, the elements starting at the topmost nonzero element in the column and ending at the diagonal element are stored contiguously in AU. The elements stored may include zero elements along with the nonzero elements. If all elements in the column to be stored are zero, the diagonal element, a_{ii}, having a value of zero, is stored in AU for that column. A total of *nu* elements are stored for the upper triangle of ***A***.
- IDU, an integer array of (at least) length *n*+1 contains the relative position of each diagonal element of matrix ***A*** in array AU; that is, each element IDU(*i*) of the diagonal pointer array indicates where diagonal element a_{ii} is stored in array AU. One-origin is used, so the first element of IDU is always 1. The last element, IDU(*n*+1), indicates the position after the last element in array AU, which is *nu*+1.
- AL of (at least) length *nl* contains the lower triangle of the sparse matrix ***A***, where the rows are stored consecutively from 1 to *n* in AL in the following way. For each row, the elements starting at the leftmost nonzero element in the row and ending at the diagonal element are stored contiguously in AL. The elements stored may include zero elements along with the nonzero elements. If all elements in the row to be stored are zero, the diagonal element, a_{ii}, having a value of zero, is stored in AL for that row. A total of *nl* elements are stored for the lower triangle of ***A***. The values of the diagonal elements are meaningless, so you can store any values in those positions in AL.
- IDL, an integer array of (at least) length *n*+1 contains the relative position of each diagonal element of matrix ***A*** in array AL; that is, each element IDL(*i*) of the diagonal pointer array indicates where diagonal element a_{ii} is stored in array AL. One-origin is used, so the first element of IDL is always 1. The last element, IDL(*n*+1), indicates the position after the last element in array AL, which is *nl*+1.

Consider the following as an example of a 6 by 6 general sparse matrix ***A*** and how it is stored in arrays AU, IDU, AL, and IDL.

Given the following matrix ***A***:

```
                    |  0  12  13   0   0   0 |
                    | 21  22   0  24   0   0 |
                    | 31   0  33  34   0  36 |
                    | 41  42  43  44  45   0 |
                    |  0   0   0  54  55  56 |
                    |  0   0  63   0  65  66 |
```

the arrays are:

```
    AU = (0, 12, 22, 13, 0, 33, 24, 34, 44, 45, 55, 36, 0, 56, 66)
```

IDU = (1, 3, 6, 9, 11, 15, 16) where *nu*=15

```
    AL = (*, 21, *, 31, 0, *, 41, 42, 43, *, 54, *, 63, 0, 65, *)
```

IDL = (1, 3, 6, 10, 12, 16, 17) where *nl*=16

and where "*" means you do not have to store a value in that position in the array. However, these storage positions are required.

For a **symmetric** sparse matrix of order *n*, profile-in skyline storage mode uses the same storage technique as for the upper triangle and diagonal elements of the general sparse matrix; therefore, only the AU and IDU arrays are needed.

Consider the following as an example of a symmetric sparse matrix ***A*** of order 6 and how it is stored in arrays AU and IDU.

Given the following matrix ***A***:

```
                    |  0  12  13   0   0   0 |
                    | 12  22   0  24   0   0 |
                    | 13   0  33  34   0  36 |
                    |  0  24  34  44  45   0 |
                    |  0   0   0  45  55  56 |
                    |  0   0  36   0  56  66 |
```

the arrays are:

```
    AU = (0, 12, 22, 13, 0, 33, 24, 34, 44, 45, 55, 36, 0, 56, 66)
```

`IDU = (1, 3, 6, 9, 11, 15, 16)` where nu=15

In general terms, this storage technique can be expressed as follows:

For general sparse matrices and symmetric sparse matrices:
For each a_{ij} for $j = 1, n$ and $i = k, j$,
where a_{kj} is the topmost $a_{ij} \neq 0$ in each column j,
there exists m, where $1 \leq m \leq nu$, such that
AU(m-j+i) = a_{ij}
IDU(j) = m for each a_{jj}
IDU(n+1) = nu+1

Also, for general sparse matrices:
For each a_{ij} for $i = 1, n$ and $j = k, i$,
where a_{ik} is the leftmost $a_{ij} \neq 0$ in each row i,
there exists m, where $1 \leq m \leq nl$, such that
AL(m-i+j) = a_{ij}
IDL(i) = m for each a_{ii}
IDL(n+1) = nl+1

where:

a_{ij} are the elements of sparse matrix ***A***, of order n.
Array AU has nu elements.
Array AL has nl elements.
Arrays IDU and IDL each have n+1 elements.

Sequences

A sequence is an ordered collection of numbers. It can be a one-, two-, or three-dimensional sequence. Sequences are used in the areas of sorting, searching, Fourier transforms, convolutions, and correlations.

Real and Complex Elements in Storage

Sequences can contain either real or complex data. For sequences containing complex data, a special storage arrangement is used to accommodate the two parts, a and b, of each complex number, a+bi, in the array. For each complex number, two sequential storage locations are required in the array. Therefore, exactly twice as much storage is required for complex sequences as for real sequences of the same precision. See "How Do You Set Up Your Scalar Data?" on page 47 for a description of real and complex numbers, and "How Do You Set Up Your Arrays?" on page 47 for a description of how real and complex data is stored in arrays.

One-Dimensional Sequences

A one-dimensional sequence appears symbolically as follows, where the subscripts indicate the element positions within the sequence:

$(x_1, x_2, x_3, \dots, x_n)$

One-Dimensional Sequence Storage Representation

A one-dimensional sequence is stored in an array using stride in the same way a vector uses stride. For details, see "How Stride Is Used for Vectors" on page 73.

Two-Dimensional Sequences

A two-dimensional sequence appears symbolically as a series of columns of elements. (They are represented in the same way as a matrix without the square brackets.) The two subscripts indicate the element positions in the first and second dimensions, respectively:

$$\begin{matrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{matrix}$$

Two-Dimensional Sequence Storage Representation

A two-dimensional sequence is stored in an array using the stride for the second dimension in the same way that a matrix uses leading dimension. In the simplest form, it uses a stride of 1 for the first dimension; however, certain subroutines may allow you to specify a stride for the first dimension that is greater than 1. For details, see "How Leading Dimension Is Used for Matrices" on page 78. (In the area of Fourier transforms, a two-dimensional sequence may be stored in transposed form in an array. In this case, the stride for the second dimension is 1, and the stride for the first dimension is the leading dimension of the array.)

Three-Dimensional Sequences

A three-dimensional sequence is represented as a series of blocks of elements. Each block is equivalent to a two-dimensional sequence. The number of blocks indicates the length of the third dimension. The three subscripts indicate the element positions in the first, second, and third dimensions, respectively:

Plane 0:

$$\begin{array}{ccc} a_{0,0,0} & \cdots & a_{0,n-1,0} \\ a_{1,0,0} & \cdots & a_{1,n-1,0} \\ \vdots & & \vdots \\ a_{m-1,0,0} & \cdots & a_{m-1,n-1,0} \end{array}$$

Plane 1:

$$\begin{array}{ccc} a_{0,0,1} & \cdots & a_{0,n-1,1} \\ a_{1,0,1} & \cdots & a_{1,n-1,1} \\ \vdots & & \vdots \\ a_{m-1,0,1} & \cdots & a_{m-1,n-1,1} \end{array}$$

$$\vdots$$

Plane $(p-1)$:

$$\begin{array}{ccc} a_{0,0,p-1} & \cdots & a_{0,n-1,p-1} \\ a_{1,0,p-1} & \cdots & a_{1,n-1,p-1} \\ \vdots & & \vdots \\ a_{m-1,0,p-1} & \cdots & a_{m-1,n-1,p-1} \end{array}$$

Three-Dimensional Sequence Storage Representation

Each block of elements in a three-dimensional sequence is stored successively in an array. The stride for the third dimension is used to select the elements for each successive block of elements in the array. The starting point of the three-dimensional sequence is specified as the argument for the sequence in the ESSL calling statement. For example, if the three-dimensional sequence is contained in array `BIG`, declared as `BIG(1:20,1:30,1:10)`, and starts at the second element in the first dimension, the third element in the second dimension, and the first element in the third dimension of array `BIG`, you should specify `BIG(2,3,1)` as the argument for the sequence, such as in:

```
   CALL  SCFT3 (BIG(2,3,1),20,600,Y,32,2056,16,20,10,1,1.0,AUX,30000)
```

See "How Stride Is Used for Three-Dimensional Sequences" on page 123 for a detailed description of how three-dimensional sequences are stored within arrays using strides.

How Stride Is Used for Three-Dimensional Sequences

The elements of the three-dimensional sequence can be defined as a_{ijk} for $i = 1, m$, $j = 1, n$, and $k = 1, p$. The first two subscripts, *i* and *j*, define the elements in the first two dimensions of the sequence, and the third subscript, *k*, defines the elements in the third dimension. Using this definition of three-dimensional sequences, this explains how these elements are mapped into an array using the concepts of stride. (Remember that the elements a_{ijk} are the elements of the conceptual data structure, the three-dimensional sequence to be processed by ESSL. The sequence does not have to include all the elements in the array. Strides are used by the ESSL subroutines to select the desired elements to be processed in the array.)

The sequence elements in the first two dimensions are mapped into an array in the same way a matrix or two-dimensional sequence is mapped into an array. It uses all the items listed in "How Leading Dimension Is Used for Matrices" on page 78, such as the starting point, the number of rows and columns, and the leading dimension. In the simplest form, the stride for the first dimension, *inc1*, of a three-dimensional sequence is assumed to be 1, as for matrices; however, certain subroutines may allow you to specify a stride for the first dimension that is greater than 1. The stride for the second dimension, *inc2*, of a three-dimensional sequence is equivalent to the leading dimension for a matrix.

The stride for the third dimension, *inc3*, is used to define the array elements that make up the third dimension of the three-dimensional sequence. The stride for the third dimension is used as an increment to step through the array to find the starting point for each of the *p* successive blocks of elements in the array. The stride, *inc3*, must always be positive. It must always be greater than or equal to the number of elements to be processed in the first two dimensions; that is, $inc3 \geq (inc2)(n)$.

A three-dimensional sequence is usually stored in a one-, two-, or three-dimensional array; however, for the sake of this discussion, a three-dimensional array is used here. For an array, A, declared as `A(E1:E2,F1:F2,G1:G2)`, the strides in the first, second, and third dimensions are:

inc1 = 1
inc2 = (E2-E1+1)
inc3 = (E2-E1+1)(F2-F1+1)

Given an array A, declared as `A(1:7,1:3,0:3)`, where the lengths of the first, second, and third dimensions are 7, 3, and 4, respectively, the resulting strides are *inc1* = 1, *inc2* = 7, and *inc3* = 21.

The starting point for a three-dimensional sequence in an array is at the location specified by the argument for the sequence in the ESSL calling statement. Using the array A, described above, if you specify `A(2,2,1)` for a three-dimensional sequence, where A is defined as follows, in four blocks, for planes 0 - 3, respectively:

```
1.0  8.0 15.0    22.0 29.0 36.0    43.0 50.0 57.0    64.0 71.0 78.0
2.0  9.0 16.0    23.0 30.0 37.0    44.0 51.0 58.0    65.0 72.0 79.0
3.0 10.0 17.0    24.0 31.0 38.0    45.0 52.0 59.0    66.0 73.0 80.0
4.0 11.0 18.0    25.0 32.0 39.0    46.0 53.0 60.0    67.0 74.0 81.0
5.0 12.0 19.0    26.0 33.0 40.0    47.0 54.0 61.0    68.0 75.0 82.0
```

```
 6.0 13.0 20.0    27.0 34.0 41.0    48.0 55.0 62.0    69.0 76.0 83.0
 7.0 14.0 21.0    28.0 35.0 42.0    49.0 56.0 63.0    70.0 77.0 84.0
```

then processing begins in the second block of elements at row 2 and column 2 in array A, which is 30.0. The stride in the third dimension is then used to find the starting point for each of the next *p*-1 successive blocks of elements in the array. The stride, *inc3*, is added to the starting point *p*-1 times. In this example, the stride for the third dimension is 21, and the number of blocks of elements, *p*, to be processed is 3, so the starting points in array A are `A(2,2,1)`, `A(2,2,2)`, and `A(2,2,3)`. These are elements 30.0, 51.0, and 72.0. These array elements then correspond to the sequence elements a_{111}, a_{112}, and a_{113}, respectively.

In general terms, this results in the following starting positions for the blocks of elements in the array:

A(BEGINI, BEGINJ, BEGINK)
A(BEGINI, BEGINJ, BEGINK+1)
A(BEGINI, BEGINJ, BEGINK+2)
.
.
A(BEGINI, BEGINJ, BEGINK+*p*-1)

Using *m* = 4, *n* = 2, and *p* = 3 to define the elements of the three-dimensional data structure in this example, the resulting three-dimensional sequence is defined as follows, in three blocks, for planes 0 - 2, respectively:

Plane 0:	**Plane 1:**	**Plane 2:**
a_{000} a_{010}	a_{001} a_{011}	a_{002} a_{012}
a_{100} a_{110}	a_{101} a_{111}	a_{102} a_{112}
a_{200} a_{210}	a_{201} a_{211}	a_{202} a_{212}
a_{300} a_{310}	a_{301} a_{311}	a_{302} a_{312}

Plane 0:	**Plane 1:**	**Plane 2:**
30.0 37.0	51.0 58.0	72.0 79.0
31.0 38.0	52.0 59.0	73.0 80.0
32.0 39.0	53.0 60.0	74.0 81.0
33.0 40.0	54.0 61.0	75.0 82.0

As shown in this example, the three-dimensional sequence does not have to include all the blocks of elements in the array. In this case, the three-dimensional sequence includes only the second through the fourth block of elements in the array. The first block is not used. Elements of an array are selected as they are arranged in storage, regardless of the number of dimensions defined in the array. Therefore, when using a one- or two-dimensional array to store your three-dimensional sequence, you should understand how your array elements are stored to ensure that elements are selected properly. See "Setting Up Arrays in Fortran" on page 126 for a description of array storage.

Note: Three-dimensional sequences are used by the three-dimensional Fourier transform subroutines and the Multidimensional Fourier transform subroutines. By specifying certain stride values for *inc1*, *inc2*, and *inc3* and declaring your arrays to have certain number of dimensions, you achieve optimal performance in these subroutines. For details, see "Setting Up Your Data" on page 959 for each subroutine.

Chapter 4. Coding programs

This provides you with information you need to code your Fortran, C, and C++ programs.

Fortran Programs

This describes how to code your Fortran program using any of the ESSL run-time libraries.

Calling ESSL Subroutines and Functions in Fortran

In Fortran programs, most ESSL subroutines are invoked with the CALL statement:

```
    CALL   subroutine-name  (argument-1, . . . , argument-n)
```

An example of a calling sequence for the SAXPY subroutine might be:

```
    CALL SAXPY (5,A,X,J+INC,Y,1)
```

The remaining ESSL subroutines are invoked as functions by coding a function reference. You first declare the type of value returned by the function: short- or long-precision real, short- or long-precision complex, or integer. Then you code the function reference as part of an expression in a statement. An example of declaring and invoking the DASUM function might be:

```
    DOUBLE PRECISION DASUM,SUM,X
        .
        .
        .
    SUM = DASUM (N,X,INCX)
```

Values are returned differently for ESSL subroutines and functions. For subroutines, the results of the computation are returned in an argument specified in the calling sequence. In the CALL statement above, the result is returned in argument Y. For functions, the result is returned as the value of the function. In the assignment statement above, the result is assigned to SUM.

See the Fortran publications for details on how to code the CALL statement and a function reference.

Setting Up a User-Supplied Subroutine for ESSL in Fortran

Some ESSL numerical quadrature subroutines call a user-supplied subroutine, *subf*, identified in the ESSL calling sequence. If your program that calls the numerical quadrature subroutines is coded in Fortran, there are some coding rules you must follow:

- You must declare *subf* as EXTERNAL in your program.
- You should code the *subf* subroutine to the specifications given in "Programming Considerations for the SUBF Subroutine" on page 1144. For examples of coding a *subf* subroutine in Fortran, see the subroutine descriptions there.

Setting Up Scalar Data in Fortran

Table 41 on page 126 lists the scalar data types in Fortran that are used for ESSL. Only those types and lengths used by ESSL are listed.

Table 41: Scalar Data Types in Fortran Programs.

Terminology Used by ESSL	Fortran Equivalent
Character item[1] 'N', 'T', 'C' or 'n', 't', 'c'	CHARACTER*1 'N', 'T', 'C'
32-bit logical item [4] .TRUE., .FALSE.	LOGICAL or LOGICAL*4 .TRUE., .FALSE.
64-bit logical item [4] .TRUE., .FALSE.	LOGICAL or LOGICAL*8 .TRUE., .FALSE.
32-bit integer [2, 4] 12345, -12345	INTEGER or INTEGER*4 12345, -12345
64-bit integer [4] 12345, -12345	INTEGER or INTEGER*8 12345_8, -12345_8
Short-precision real number[3] 12.345	REAL or REAL*4 0.12345E2
Long-precision real number[3] 12.345	DOUBLE PRECISION, REAL, or REAL*8 0.12345D2
Short-precision complex number[3] (123.45, -54321.0)	COMPLEX or COMPLEX*8 (123.45E0, -543.21E2)
Long-precision complex number[3] (123.45, -54321.0)	COMPLEX or COMPLEX*16 (123.45D0, -543.21D2)

Note:

1. ESSL accepts character data in either upper- or lowercase in its calling sequences.
2. For a 32-bit integer, 64-bit pointer environment, in accordance with the LP64 data model, all ESSL integer arguments remain 32 bits except for the **iusadr** argument for ERRSET.
3. Short- and long-precision numbers look the same in this documentation.
4. The default size for INTEGER and LOGICAL data entities that have no length or kind specified is 32 bits. However, the `-qintsize=8` compiler option sets the size of such INTEGER and LOGICAL data entities to 64 bits.

Setting Up Arrays in Fortran

Arrays are declared in Fortran by specifying the array name, the number of dimensions, and the range of each dimension in a DIMENSION statement or an explicit data type statement, such as REAL, DOUBLE PRECISION, and so forth.

Real and Complex Array Elements

Each array element can be either a real or complex data item of short or long precision. The type of the array determines the size of the element storage locations. Short-precision data requires 4 bytes, and long-precision data requires 8 bytes. Complex data requires two storage locations of either 4 or 8 bytes each, for short or long precision, respectively, to accommodate the two parts of the complex number: $c = a+bi$. Therefore, exactly twice as much storage is required for complex data as for real data of the same precision. See “How Do You Set Up Your Scalar Data?” on page 47 for a description of real and complex numbers.

Even though complex data items require two storage locations, the same number of elements exist in the array as for real data. A reference to an element—for example, `C(3)`—in an array containing complex data gives you the whole complex number; that is, it contains both *a* and *b*, where the complex number is expressed as follows:

`C(I)`$\leftarrow(a_i, b_i)$ for a one-dimensional array
`C(I,J)`$\leftarrow(a_{ij}, b_{ij})$ for a two-dimensional array
`C(I,J,K)`$\leftarrow(a_{ijk}, b_{ijk})$ for a three-dimensional array

One-Dimensional Array

For a one-dimensional array in Fortran 77, you can code:

```
   DIMENSION  A(E1:E2)
```

where A is the name of the array, E1 is the lower bound, and E2 is the upper bound of the single dimension in the array. If the lower bound is not specified, such as in `A(E2)`, the value is assumed to be 1. The upper bound is required.

A one-dimensional array is stored in ascending storage locations (relative to some base storage address) in the following order:

Relative Location	**Array Element**
1	`A(E1)`
2	`A(E1+1)`
3	`A(E1+2)`
.	.
.	.
.	.
E2–E1+1	`A(E2)`

For example, the array A of length 4 specified in the DIMENSION statement as `A(0:3)` and containing the following elements:

```
A = (1, 2, 3, 4)
```

has its elements arranged in storage as follows:

Relative Location	**Array Element Value**
1	1
2	2
3	3

4
: 4

Two-Dimensional Array

For a two-dimensional array in Fortran 77, you can code:

```
   DIMENSION  A(E1:E2,F1:F2)
```

where A is the name of the array. E1 and F1 are the lower bounds of the first and second dimensions, respectively, and E2 and F2 are the upper bounds of the first and second dimensions, respectively. If either of the lower bounds is not specified, such as in `A(E2,F1:F2)`, the value is assumed to be 1. The upper bounds are always required for each dimension. For examples of Fortran 77 usage, see “SGEMV, DGEMV, CGEMV, ZGEMV, SGEMX, DGEMX, SGEMTX, and DGEMTX (Matrix-Vector Product for a General Matrix, Its Transpose, or Its Conjugate Transpose)” on page 295.

The elements of a two-dimensional array are stored in column-major order; that is, they are stored in the following ascending storage locations (relative to some base storage address) with the value of the first (row) subscript expression increasing most rapidly and the value of the second (column) subscript expression increasing least rapidly. Following are the locations of the elements in the array:

Relative Location	Array Element
1	`A(E1,F1)` (starting column 1)
2	`A(E1+1,F1)`
.	.
.	.
.	.
E2−E1+1	`A(E2,F1)`
(E2−E1+1)+1	`A(E1,F1+1)` (starting column 2)
(E2−E1+1)+2	`A(E1+1,F1+1)`
.	.
.	.
.	.
(E2−E1+1)(2)	`A(E2,F1+1)`
(E2−E1+1)(2)+1	`A(E1,F1+2)` (starting column 3)
(E2−E1+1)(2)+2	`A(E1+1,F1+2)`
.	.

•
    .

•
    .

(E2–E1+1)(F2–F1)
    `A(E2,F2-1)`

(E2–E1+1)(F2–F1)+1
    `A(E1,F2)` (starting column F2-F1+1)

(E2–E1+1)(F2–F1)+2
    `A(E1+1,F2)`

•
    .

•
    .

•
    .

(E2–E1+1)(F2–F1+1)
    `A(E2,F2)`

For example, the 3 by 4 array A specified in the DIMENSION statement as `A(2:4,1:4)` and containing the following elements:

```
          ┌                ┐
          | 11  12  13  14 |
 A    =   | 21  22  23  24 |
          | 31  32  33  34 |
                          ┘
```

has its elements arranged in storage as follows:

Relative Location	Array Element Value
1	11 (starting column 1)
2	21
3	31
4	12 (starting column 2)
5	22
6	32
7	13 (starting column 3)
8	23
9	33
10	14 (starting column 4)
11	24

12
    34

Each element A(I,J) of the array A, declared A(1:*n*, 1:*m*), containing real or complex data, occupies the storage location whose address is given by the following formula:

address {A(I,J)} = address {A} + (I-1 + *n*(J-1))*f*

for:

I = 1, *n* and
J = 1, *m*

where:

f = 4 for short-precision real numbers
f = 8 for long-precision real numbers
f = 8 for short-precision complex numbers
f = 16 for long-precision complex numbers

Three-Dimensional Array

For a three-dimensional array in Fortran 77, you can code:

```
   DIMENSION  A(E1:E2,F1:F2,G1:G2)
```

where A is the name of the array. E1, F1, and G1 are the lower bounds of the first, second, and third dimensions, respectively, and E2, F2, and G2 are the upper bounds of the first, second, and third dimensions, respectively. If any of the lower bounds are not specified, such as in A(E1:E2,F1:F2,G2), the value is assumed to be 1. The upper bounds are always required for each dimension. For examples of Fortran 77 usage, see “SCFT3 and DCFT3 (Complex Fourier Transform in Three Dimensions)” on page 1041.

The elements of a three-dimensional array can be thought of as a set of two-dimensional arrays, stored sequentially in ascending storage locations in the array. In the three-dimensional array, the value of the first (row) subscript expression increases most rapidly, the second (column) subscript expression increases less rapidly, and the third subscript expression (set of rows and columns) increases least rapidly. Following are the locations of the elements in the array:

Relative Location
    Array Element

1
    A(E1,F1,G1) (starting the first set)

2
    A(E1+1,F1,G1)

.
    .

.
    .

.
    .

(E2–E1+1)(F2–F1+1)
    A(E2,F2,G1)

(E2–E1+1)(F2–F1+1)+1
    A(E1,F1,G1+1) (starting the second set)

(E2–E1+1)(F2–F1+1)+2
    A(E1+1,F1,G1+1)

.
    .

.
    .

.
    .

(E2−E1+1)(F2−F1+1)(2)
    `A(E2,F2,G1+1)`

(E2−E1+1)(F2−F1+1)(2)+1
    `A(E1,F1,G1+2)` (starting the third set)

(E2−E1+1)(F2−F1+1)(2)+2
    `A(E1+1,F1+2)`

.
    .

.
    .

.
    .

(E2−E1+1)(F2−F1+1)(G2−G1)
    `A(E2,F2,G2−1)`

(E2−E1+1)(F2−F1+1)(G2−G1)+1
    `A(E1,F1,G2)` (starting the last set*)

(E2−E1+1)(F2−F1+1)(G2−G1)+2
    `A(E1+1,F1,G2)`

.
    .

.
    .

.
    .

(E2−E1+1)(F2−F1+1)(G2−G1+1)
    `A(E2,F2,G2)`

* The last set is the G2−G1+1 set.

For example, the 3 by 2 by 4 array A specified in the DIMENSION statement as `A(1:3,0:1,2:5)` and containing the following sets of rows and columns of elements:

```
     | 111 121 |    | 112 122 |  | 113 123 |  | 114 124 |
A =  | 211 221 |    | 212 222 |  | 213 223 |  | 214 224 |
     | 311 321 |    | 312 322 |  | 313 323 |  | 314 324 |
```

has its elements arranged in storage as follows:

Relative Location
    Array Element Value

1
    111 (starting the first set)

2
    211

3
    311

4
    121

5
221

6
321

7
112 (starting the second set)

8
212

9
312

10
122

11
222

12
322

13
113 (starting the third set)

14
213

15
313

16
123

17
223

18
323

19
114 (starting the fourth set)

20
214

21
314

22
124

23
224

24
324

Each element `A(I,J,K)` of the array `A`, declared `A`(1:*n*, 1:*m*, 1:*p*), containing real or complex data, occupies the storage location whose address is given by the following formula:

address {A(I,J,K)} = address {A} + (I-1 + n(J-1) + mn(K-1))f

for:

I = 1, n
J = 1, m
K = 1, p

where:

f = 4 for short-precision real numbers
f = 8 for long-precision real numbers
f = 8 for short-precision complex numbers
f = 16 for long-precision complex numbers

Creating Multiple Threads and Calling ESSL from Your Fortran Program

The following example shows how to create up to a maximum of eight threads, where each thread calls the DURAND and DGEICD subroutines.

Note: Be sure to compile this program with the `xlf_r` command and the `-qnosave` option.

```
       program matinv_example
       implicit none
!
!  program to invert m nxn random matrices
!
       real(8), allocatable :: A(:,:,:), det(:,:), rcond(:)
       real(8)              :: dummy_aux, seed=1998, sd
       integer              :: rc, i, m=8, n=500, iopt=3, naux=0
!
! allocate storage
!
       allocate(A(n,n,m),stat=rc)
       call error_exit(rc,"Allocation of matrix A")
       allocate(det(2,m),stat=rc)
       call error_exit(rc,"Allocation of det")
       allocate(rcond(m),stat=rc)
       call error_exit(rc,"Allocation of rcond")
!
! Calculate inverses in parallel
!
!SMP$  parallel do private(i,sd), schedule(static),
!SMP$&    share(n,a,iopt,rcond,det,dummy_aux,naux)
       do i=1,m

     sd = seed + 100*i
         call durand(sd,n*n,A(1,1,i))
         call dgeicd(A(1,1,i),n,n,iopt,rcond(i),det(1,i),
     &               dummy_aux,naux)
       enddo

       write(*,*)'Reciprocal condition numbers of the matrices are:'
       write(*,'(4E12.4)') rcond
!
       deallocate(A,stat=rc)

       call error_exit(rc,"Deallocation of matrix A")
       deallocate(det,stat=rc)
       call error_exit(rc,"Deallocation of det")
       deallocate(rcond,stat=rc)
       call error_exit(rc,"Deallocation of rcond")
       stop

         contains
         subroutine error_exit(error_code,string)
         character(*)    :: string
         integer         :: error_code
         if(error_code .eq. 0 ) return
         write(0,*)string,": failing return code was ",error_code
         stop 1
         end subroutine error_exit
       end
```

Handling Errors in Your Fortran Program

ESSL provides you with flexibilities in handling both input-argument errors and computational errors:

- For input-argument errors 2015, 2030, and 2200 which are optionally-recoverable errors, ESSL allows you to obtain corrected input-argument values and react at run time.

Note: In the case where error 2015 is unrecoverable, you have the option of dynamic allocation for most of the *aux* arguments. For details see the subroutine descriptions.

- For computational errors, ESSL provides a return code and additional information to help you analyze the problem in your program and react at run time.

"Input-Argument Errors in Fortran" on page 134 and "Computational Errors in Fortran" on page 136 explain how to use these facilities by describing the additional statements you must code in your program.

For multithreaded application programs, if you want to initialize the error option table and change the default settings for input-argument and computational errors, you need to implement the steps shown in "Input-Argument Errors in Fortran" on page 134 and "Computational Errors in Fortran" on page 136 on each thread that calls ESSL. An example is shown in "Example of Handling Errors in a Multithreaded Application Program " on page 141.

Input-Argument Errors in Fortran

To obtain corrected input-argument values in a Fortran program and to avert program termination for the optionally-recoverable input-argument errors 2015, 2030, and 2200 add the statements in the following steps your program. Steps 3 and 7 for ERRSAV and ERRSTR, respectively, are optional. Adding these steps makes the effect of the call to ERRSET temporary.

Step 1. Declare ENOTRM as External

```
EXTERNAL ENOTRM
```

This declares the ESSL error exit routine ENOTRM as an external reference in your program. This should be coded in the beginning of your program before any of the following statements.

Step 2. Call EINFO for Initialization

```
CALL EINFO (0)
```

This calls the EINFO subroutine with one argument of value 0 to initialize the ESSL error option table. It is required only if you call ERRSET in your program. It is coded only once in the beginning of your program before any calls to ERRSET. For a description of EINFO, see "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188.

Step 3. Call ERRSAV

```
CALL  ERRSAV (ierno,tabent)
```

(This is an optional step.) This calls the ERRSAV subroutine, which stores the error option table entry for error number *ierno* in an 8-byte storage area, *tabent*, which is accessible to your program. ERRSAV must be called for each entry you want to save. This step is used, along with step 7, for ERRSTR. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 69. For an example, see "Computational Errors in Fortran Example 3" on page 141, as the use is the same as for computational errors.

Step 4. Call ERRSET

```
CALL  ERRSET (ierno,inoal,inomes,itrace,iusadr,irange)
```

This calls the ERRSET subroutine, which allows you to dynamically modify the action taken when an error occurs. For optionally-recoverable ESSL input-argument errors, you need to call ERRSET only if you want to avoid terminating your program and you want the input arguments associated with this error to be assigned correct values in your program when the error occurs. For one error (*ierno*) or a range of errors (*irange*), you can specify:

- How many times each error can occur before execution terminates (*inoal*)
- How many times each error message can be printed (*inomes*)
- The ESSL exit routine ENOTRM, to be invoked for the error indicated (*iusadr*)

ERRSET must be called for each error code you want to indicate as being recoverable. For ESSL, *ierno* should have a value of 2015, 2030 or 2200. If you want to eliminate error messages, you should indicate a negative number for *inomes*; otherwise, you should specify 0 for this argument. All the other ERRSET arguments should be specified as 0.

For a list of the default values set in the ESSL error option table, see “How Do You Control Error Handling by Setting Values in the ESSL Error Option Table?” on page 67. For a description of the input-argument errors, see “Input-Argument Error Messages(2001-2099)” on page 187. For a description of ERRSET, see Chapter 17, “Utilities subroutines,” on page 1187.

Step 5. Call ESSL

```
  CALL  name  (arg-1,...,arg-n,*yyy,*zzz,...)
```

This calls the ESSL subroutine and specifies a branch on one or more return code values, where:

- *name* specifies the ESSL subroutine.
- *arg-1,..., arg-n* are the input and output arguments.
- *yyy*, *zzz*, and any other statement numbers preceded by an "*" are the Fortran statement numbers indicating where you want to branch when you get a nonzero return code. Each corresponds to a different ESSL value. Control goes to the corresponding statement number when a nonzero return code value is returned for the CALL statement. Return code values are described under “Error Conditions” in each ESSL subroutine description.

Step 6. Perform the Desired Action

These are the statements at statement number *yyy* or *zzz*, shown in the CALL statement in Step 5, and preceded by an "*". The statement to which control is passed corresponds to the return code value for the error.

These statements perform whatever action is desired when the recoverable error occurs. These statements may check the new values set in the input arguments to determine whether adequate program storage is available, and then decide whether to continue or terminate the program. Otherwise, these statements may check that the size of the working storage arrays or the length of the transform agrees with other data in the program. The program may also store this corrected input argument value for future reference.

Step 7. Call ERRSTR

```
  CALL  ERRSTR (ierno,tabent)
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number *ierno* from an 8-byte storage area, *tabent*, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 3, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see “ How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?” on page 69. For an example, see “Computational Errors in Fortran Example 3” on page 141, as the use is the same as for computational errors.

Input-Argument Errors in Fortran Example

This example shows an error code 2015, which resets the size of the work area *aux*, specified in *naux*, if the value specified is too small. It also indicates that no error messages should be issued.

```
       .
       .
       .
C                    DECLARE ENOTRM AS EXTERNAL
       EXTERNAL ENOTRM
       .
       .
       .

C                    INITIALIZE THE ESSL ERROR
C                    OPTION TABLE
       CALL EINFO(0)
       .
       .
       .

C                    MAKE ERROR CODE 2015 A RECOVERABLE
C                    ERROR AND SUPPRESS PRINTING ALL
C                    ERROR MESSAGES FOR IT
       CALL ERRSET(2015,0,-1,0,ENOTRM,2015)
       .
       .
       .

C                    CALL ESSL ROUTINE SWLEV.
C                    IF THE NAUX INPUT
C                    ARGUMENT IS TOO SMALL, ERROR
C                    2015 OCCURS.  THE MINIMUM VALUE
C                    REQUIRED IS STORED IN THE NAUX
C                    INPUT ARGUMENT AND CONTROL GOES
C                    TO LABEL 400.
       CALL SWLEV(X,INCX,U,INCU,Y,INCY,N,AUX,NAUX,*400)
       .
       .
       .

C                    CHECK THE RESULTING INPUT ARGUMENT
C                    VALUE  IN NAUX AND TAKE THE
C                    DESIRED ACTION
400    .
       .
       .
```

Computational Errors in Fortran

To obtain information about an ESSL computational error in a Fortran program, add the statements in the following steps to your program. Steps 2 and 7 for ERRSAV and ERRSTR, respectively, are optional. Adding these steps makes the effect of the call to ERRSET temporary. For a list of those computational errors that return information and to which these steps apply, see “EINFO (ESSL Error Information-Handler Subroutine)” on page 1188.

Step 1. Call EINFO for Initialization

```
   CALL EINFO (0)
```

This calls the EINFO subroutine with one argument of value 0 to initialize the ESSL error option table. It is required only if you call ERRSET in your program. It is coded only once in the beginning of your program before any calls to ERRSET. For a description of EINFO, see "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188.

Step 2. Call ERRSAV

```
   CALL  ERRSAV (ierno,tabent)
```

(This is an optional step.) This calls the ERRSAV subroutine, which stores the error option table entry for error number *ierno* in an 8-byte storage area, *tabent*, which is accessible to your program. ERRSAV must be called for each entry you want to save. This step is used, along with step 7, for ERRSTR. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 69.

Step 3. Call ERRSET

```
CALL  ERRSET (ierno,inoal,inomes,itrace,iusadr,irange)
```

This calls the ERRSET subroutine, which allows you to dynamically modify the action taken when an error occurs. For ESSL computational errors, you need to call ERRSET only if you want to change the default values in the ESSL error option table. For one error (*ierno*) or a range of errors (*irange*), you can specify:

- How many times each error can occur before execution terminates (*inoal*)
- How many times each error message can be printed (*inomes*)

ERRSET must be called for each error code for which you want to change the default values. For ESSL, *ierno* should be set to one of the eligible values listed in "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188. To allow your program to continue after an error in the specified range occurs, *inoal* must be set to a value greater than 1. For ESSL, *iusadr* should be specified as either 0 or 1 in a 32-bit integer, 32-bit pointer environment (0_8 or 1_8 in a 32-bit integer, 64-bit pointer environment or a 64-bit integer, 64-bit pointer environment), so a user exit is not taken.

For a list of the default values set in the ESSL error option table, see "How Do You Control Error Handling by Setting Values in the ESSL Error Option Table?" on page 67. For a description of the computational errors, see "Computational Error Messages(2100-2199)" on page 191. For a description of ERRSET, see Chapter 17, "Utilities subroutines," on page 1187.

Step 4. Call ESSL

```
   CALL  name  (arg-1,...,arg-n,*yyy,*zzz,...)
```

This calls the ESSL subroutine and specifies a branch on one or more return code values, where:

- *name* specifies the ESSL subroutine.
- *arg-1*,..., *arg-n* are the input and output arguments.
- *yyy*, *zzz*, and any other statement numbers preceded by an "*" are the Fortran statement numbers indicating where you want to branch when you get a nonzero return code. Each corresponds to a

different ESSL value. Control goes to the corresponding statement number when a nonzero return code value is returned for the CALL statement. Return code values are described under “Error Conditions” in each ESSL subroutine description.

Step 5. Call EINFO for Information

```
nmbr  CALL EINFO (icode,inf1)
-or-
nmbr  CALL EINFO (icode,inf1,inf2)
```

This calls the EINFO subroutine, which returns information about certain computational errors, where:

- *nmbr* is the statement number *yyy*, *zzz*, or any of the other statement numbers preceded by an "*" in the CALL statement in Step 4, corresponding to the return code value for this error code.
- *icode* is the error code of interest.
- *inf1* and *inf2* are the integer variables used to receive the information, where *inf1* is assigned a value for all errors, and *inf2* is assigned a value for some errors. For a description of EINFO, see “EINFO (ESSL Error Information-Handler Subroutine)” on page 1188.

Step 6. Check the Values in the Information Receivers

These statements check the values returned in the output argument information receivers, *inf1* and *inf2*, which contain the information about the computational error.

Step 7. Call ERRSTR

```
CALL  ERRSTR (ierno,tabent)
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number *ierno* from an 8-byte storage area, *tabent*, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 2, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see “ How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?” on page 69.

Computational Errors in Fortran Example 1

This 32-bit integer, 64-bit pointer environment example shows an error code 2104, which returns one piece of information: the index of the last diagonal with nonpositive value (I1).

```
       .
       .
       .
C                    INITIALIZE THE ESSL ERROR
C                    OPTION TABLE
       CALL EINFO(0)
       .
       .
       .

C                    ALLOW 100 ERRORS FOR CODE 2104
       CALL ERRSET(2104,100,0,0,0_8,2104)
       .
       .
       .

C                    CALL ESSL ROUTINE DPPF.
C                    IF THE INPUT MATRIX IS NOT
C                    POSITIVE DEFINITE, CONTROL GOES TO
C                    LABEL 400
       IOPT=0
       CALL DPPF(APP,N,IOPT,*400)
       .
       .
       .

C                    CALL THE INFORMATION-HANDLER
C                    ROUTINE FOR ERROR CODE 2104 TO
C                    RETURN ONE PIECE OF INFORMATION
C                    IN VARIABLE I1, THE INDEX OF THE
C                    LAST NONPOSITIVE DIAGONAL FOUND
C                    BY ROUTINE DPPF
400    CALL EINFO (2104,I1)
       .
       .
       .
```

Computational Errors in Fortran Example 2

This 32-bit integer, 64-bit pointer environment example shows an error code 2103, which returns one piece of information: the index of the zero diagonal (I1) found by DGEF.

```
       .
       .
       .
C                    INITIALIZE THE ESSL ERROR
C                    OPTION TABLE
       CALL EINFO(0)
       .
       .
       .

C                    ALLOW 100 ERRORS FOR CODE 2103
       CALL ERRSET(2103,100,0,0,0_8,2103)
       .
       .
       .

C                    CALL ESSL SUBROUTINE DGEF.
C                    IF THE INPUT MATRIX IS
C                    SINGULAR, CONTROL GOES TO
C                    LABEL 400
       CALL DGEF(A,LDA,N,IPVT,*400)
       .
       .
       .

C                    CALL THE INFORMATION-HANDLER
C                    ROUTINE FOR ERROR CODE 2103 TO
C                    RETURN ONE PIECE OF INFORMATION
C                    IN VARIABLE I1, THE INDEX OF THE
C                    LAST ZERO DIAGONAL FOUND BY
C                    SUBROUTINE DGEF
400    CALL EINFO (2103,I1)
       .
       .
       .
```

Computational Errors in Fortran Example 3

This 32-bit integer, 64-bit pointer environment example shows an error code 2100, which returns two pieces of information: the lower range (I1) and the upper range (I2). It uses ERRSAV and ERRSTR to insulate the effects of the error handling for error 2100 by this program.

```
      .
      .
C                   DECLARE AN AREA TO SAVE THE
C                   ERROR OPTION TABLE INFORMATION
C                   FOR ERROR CODE 2100
      CHARACTER*8 SAV2100
      .
      .
C                   INITIALIZE THE ESSL ERROR
C                   OPTION TABLE
      CALL EINFO(0)
C                   SAVE THE EXISTING ERROR OPTION
C                   TABLE ENTRY FOR ERROR CODE 2100
      CALL ERRSAV(2100,SAV2100)
      .
      .
C                   ALLOW 255 ERRORS FOR CODE 2100
      CALL ERRSET(2100,255,0,0,0_8,2100)
      .
      .
C                   CALL ESSL SUBROUTINE DQINT.
C                   IF AN INVALID INDEX IS
C                   COMPUTED, CONTROL GOES TO LABEL 400
      CALL DQINT(S,G,OMEGA,X,INCX,N,T,INCT,Y,INCY,M,*400)
      .
      .
C                   CALL THE INFORMATION-HANDLER
C                   ROUTINE FOR ERROR CODE 2100 TO
C                   RETURN TWO PIECES OF INFORMATION.
C                   VARIABLE I1 CONTAINS THE LOWER RANGE
C                   FOR THE COMPUTED INDEX.
C                   VARIABLE I2 CONTAINS THE UPPER RANGE
C                   FOR THE COMPUTED INDEX.
400   CALL EINFO (2100,I1,I2)
      .
      .
C                   RESTORE THE PREVIOUS ERROR OPTION
C                   TABLE ENTRY FOR ERROR CODE 2100.
C                   ERROR PROCESSING RETURNS TO HOW IT
C                   WAS BEFORE IT WAS ALTERED BY THE ABOVE
C                   ERRSET STATEMENT.
      CALL ERRSTR(2100,SAV2100)
      .
      .
```

Example of Handling Errors in a Multithreaded Application Program

This 32-bit integer, 64-bit pointer environment example shows how to modify the MATINV_EXAMPLE program in “Creating Multiple Threads and Calling ESSL from Your Fortran Program” on page 133 with calls to the ESSL error handling subroutines. The ESSL error handling subroutines are called from each thread to: initialize the error option table, save the current error option table values for input-argument error 2015 and computational error 2105, change the default values for errors 2015 and 2105, and then restore the original default values for errors 2015 and 2105.

```
program matinv_example
       implicit none
!
!  program to invert m nxn random matrices
!
       real(8), allocatable :: A(:,:,:), det(:,:), rcond(:)
       real(8)              :: dummy_aux, seed=1998, sd
       integer              :: rc, i, m=8, n=500, iopt=3, naux=0
       integer              :: inf1(8)
       character(8)         :: sav2015(8)
       character(8)         :: sav2105(8)!
       external ENOTRM
!
! allocate storage
       allocate(A(n,n,m),stat=rc)
       call error_exit(rc,"Allocation of matrix A")
       allocate(det(2,m),stat=rc)
       call error_exit(rc,"Allocation of det")
       allocate(rcond(m),stat=rc)

       call error_exit(rc,"Allocation of rcond")
!
! Calculate inverses in parallel
!
!SMP$  parallel do private(i,sd), schedule(static),
!SMP$&    share(n,m,a,iopt,rcond,det,dummy_aux,naux,sav2015,sav2105,inf1)
       do i=1,m
!
!       initialize error handling
         call einfo(0)
!
!       Save existing option table values for error 2015
         call errsav(2015,sav2015(i))
!
!       Set Error 2015 to be non-recoverable so dgeicd will dynamically
!       allocate the work area.
         call errset(2015,100,100,0,1_8,2015)
!
!       Save existing option table values for error 2105
         call errsav(2105,sav2105(i))
!
!       Set Error 2105 to be recoverable
         call errset(2105,100,100,0,ENOTRM,2105)
!
         sd = seed + 100*i
         call durand(sd,n*n,A(1,1,i))
         call dgeicd(A(1,1,i),n,n,iopt,rcond(i),det(1,i),
     &              dummy_aux,naux,*10,*20)
10       goto 30
!
!       Catch singular matrix returned by dgeicd.
20       CALL EINFO(2105,inf1(i))
         WRITE(*,*) 'ERROR: Zero pivot found at location ',inf1(i)
!
!       Restore the error option table entries
30       continue
         call errstr(2015,SAV2015(i))
         call errstr(2105,SAV2105(i))

       enddo
```

```
       write(*,*)'Reciprocal condition numbers of the matrices are:'
       write(*,'(4E12.4)') rcond
!
       deallocate(A,stat=rc)
       call error_exit(rc,"Deallocation of matrix A")
       deallocate(det,stat=rc)
       call error_exit(rc,"Deallocation of det")

       deallocate(rcond,stat=rc)
       call error_exit(rc,"Deallocation of rcond")
       stop
       contains
         subroutine error_exit(error_code,string)
         character(*)    :: string
         integer         :: error_code
         if(error_code .eq. 0 ) return
         write(0,*)string,": failing return code was ",error_code
         stop 1
         end subroutine error_exit
       end
```

C Programs

This describes how to code your C program.

Calling ESSL Subroutines and Functions in C

This shows how to call ESSL subroutines and functions from your C program.

Before You Call ESSL

Before you can call the ESSL subroutines from your C program, you must have the appropriate ESSL header file installed on your system. The ESSL header file allows you to code your function calls as described here. It contains entries for all the ESSL subroutines. The ESSL header file is distributed with the ESSL package. The ESSL header file to be used with the C compiler is named `essl.h`. You should check with your system support group to verify that the appropriate ESSL header file is installed.

In the beginning of your program, before you call any of the ESSL subroutines, **you must code the following statement for the ESSL header file:**

```
   #include <essl.h>
```

If you are planning to create your own threads for the ESSL Thread-Safe or SMP Libraries, you must include the `pthread.h` header file as the first include file in your C program. For an example, see "Creating Multiple Threads and Calling ESSL from Your C Program" on page 146.

Coding the Calling Sequences

In C programs, the ESSL subroutines, not returning a function value, are invoked with the following type of statement:

```
    subroutine-name (argument-1, . . . , argument-n);
```

An example of a calling sequence for SAXPY might be:

```
   saxpy (5,a,x,incx,y,1);
```

The ESSL subroutines returning a function value are invoked with the following type of statement:

```
function-value-name=subroutine-name (argument-1, . . . , argument-n);
```

An example of invoking DASUM might be:

```
   sum = dasum (n,x,incx);
```

See the C publications for details about how to code the function calls.

Passing Arguments in C

This describes how to pass arguments in your C program.

About the Syntax Shown in this Documentation

The argument syntax shown assumes that you have installed and are using the ESSL header file. For further details, see "Calling ESSL Subroutines and Functions in C" on page 142.

No Optional Arguments

In the ESSL calling sequences for C, there are no optional arguments, as for some programming languages. You must code all the arguments listed in the syntax.

Arguments That Must Be Passed by Value

All scalar arguments that are not modified must be passed by value in the ESSL calling sequence. (This refers to input-only scalar arguments, such as *incx*, *m*, and *lda*.)

Arguments That Must Be Passed by Reference

Following are the instances in which you pass your arguments by reference (as a pointer) in the ESSL calling sequence:

Arrays

Arguments that are arrays are passed by reference, as usual.

Subroutine Names

Some ESSL subroutines call a user-supplied subroutine. The name is part of the ESSL calling sequence. It must be passed by reference.

Output Scalar Arguments

When an output argument is a scalar data item, it must be passed by reference. This is true for all scalar data types: real, complex, and so forth. **When this occurs, it is listed in the notes of each subroutine description.**

Character Arguments

Character arguments must be passed as strings, by reference. You specify the character, in upper- or lowercase, in the ESSL calling sequence with double quotation marks around it, as in "t". Following is an example of how you can call SGEADD, specifying the *transa* and *transb* arguments as strings *n* and *t*, respectively:

```
sgeadd (a,5,"n",b,3,"t",c,4,4,3);
```

Altered Arguments When Using Error Handling

If you use ESSL error handling in your C program, as described in "Handling Errors in Your C Program" on page 147, you must pass by reference all the arguments that can potentially be altered by ESSL error handling. This applies to all your ESSL call statements after the point where you code the #define statement, shown in step 1 in "Input-Argument Errors in C" on page 147 and in step 1 in "Computational Errors in C" on page 151. The two types of ESSL arguments are:

- *naux* arguments for auxiliary storage
- *n* arguments for transform lengths

Setting Up a User-Supplied Subroutine for ESSL in C

Some ESSL numerical quadrature subroutines call a user-supplied subroutine, *subf*, identified in the ESSL calling sequence. If your program that calls the numerical quadrature subroutines is coded in C, there are some coding rules you must follow for the *subf* subroutine:

- You can code the *subf* subroutine using only C or Fortran.
- You must declare *subf* as an external subroutine in your application program.
- You should code the *subf* subroutine to the specifications given in "Programming Considerations for the SUBF Subroutine" on page 1144. For an example of coding a *subf* subroutine in C, see Example 1.

Setting Up Scalar Data Types in C

Table 42 on page 144 lists the scalar data types in C that are used for ESSL. Only those types and lengths used by ESSL are listed.

Table 42: Scalar Data Types in C Programs.

Terminology Used by ESSL	**C Equivalent**
Character item[1] 'N', 'T', 'C' or 'n', 't', 'c'	char * "n", "t", "c"
32-bit logical item[5] .TRUE., .FALSE.	int For additional information, see "Using Logical Data in C" on page 145.[2]

Table 42: Scalar Data Types in C Programs. (continued)

Terminology Used by ESSL	**C Equivalent**
64-bit logical item[5] .TRUE., .FALSE.	long For additional information, see "Using Logical Data in C" on page 145.[2]
32-bit integer 12345, -12345	int
64-bit integer[5] 12345l, -12345l	long
Short-precision real number[4] 12.345	float
Long-precision real number[4] 12.345	double
Short-precision complex number[4] (123.45, -54321.0)	Specify it as described in "Setting Up Complex Data Types in C" on page 145.
Long-precision complex number[4] (123.45, -54321.0)	Specify it as described in "Setting Up Complex Data Types in C" on page 145.

Note:

1. ESSL accepts character data in either upper- or lowercase in its calling sequences.
2. There are no equivalent data types for logical data in C. These require special procedures. For details, see "Using Logical Data in C" on page 145.
3. For a 32-bit integer, 64-bit pointer environment, in accordance with the LP64 data model, all ESSL integer arguments remain 32-bits except for the `iusadr` argument for ERRSET.
4. Short- and long-precision numbers look the same in this documentation.
5. If you are using the ESSL hear file in a 64-bit integer, 64-bit pointer environment, add `-D_ESV6464` to your compiler command to define the integer and logical arguments as long.

Setting Up Complex Data Types in C

The ESSL header file supports C99 complex floating-point types for complex arithmetic (`<complex.h>`).

Assuming you are using the ESSL header file, if you declare data items to be of type `float_Complex` or `double_Complex`, you can pass them as short- and long-precision complex data to ESSL, respectively.

Using Logical Data in C

Logical data types are not part of the C language; however, some ESSL subroutines require arguments of these data types.

By coding the following simple macro definitions in your program, you can then use TRUE or FALSE in assigning values to or specifying any logical arguments passed to ESSL:

For 32-bit logical arguments

Use this macro definition:

```
#define FALSE 0
#define TRUE 1
```

For 64-bit logical arguments
Use this macro definition:

```
#define FALSE 0l
#define TRUE 1l
```

Setting Up Arrays in C

C arrays are arranged in storage in row-major order. This means that the last subscript expression increases most rapidly, the next-to-the-last subscript expression increases less rapidly, and so forth, with the first subscript expression increasing least rapidly. ESSL subroutines require that arrays passed as arguments be in column-major order. This is the array storage convention used by Fortran, described in "Setting Up Arrays in Fortran" on page 126. To pass an array from your C program to ESSL, to have ESSL process the data correctly, and to get a result that is in the proper form for your C program, you can do any of the following:

- Build and process the matrix, logically transposed from the outset, and transpose the results as necessary.
- Before the ESSL call, transpose the input arrays. Then, following the ESSL call, transpose any arrays updated as output.
- If there are arguments in the ESSL calling sequence indicating whether the arrays are to be processed in normal or transposed form, such as the *transa* and *transb* arguments in the _GEMM subroutines, use these arguments in combination with the matrix equivalence rules to avoid having to transpose your data in separate operations. For further detail, see "SGEMMS, DGEMMS, CGEMMS, and ZGEMMS (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm)" on page 408.

Creating Multiple Threads and Calling ESSL from Your C Program

The 32-bit integer, 64-bit pointer environment example shown below shows how to create two threads, where each thread calls the ISAMAX subroutine. To use the pthreads library, you must specify the `pthread.h` header file as the first include file in your program.

Note: Be sure to compile this program with the `cc_r` command.

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#include <essl.h>

/* Create structure for argument list  */
typedef struct {
  int        n;
  float     *x;
  int     incx;
} arg_list;

/* Define prototype for thread routine */
void *Thread(void *v);

int main()
{
 float  sx1[9] = { 1., 2., 7., -8., -5., -10., -9., 10., 6. };
 float  sx2[8] = { 1.,12., 7., -8., -5., -10., -9., 19.};
 pthread_t first_th;
 pthread_t second_th;
 int rc;
 arg_list a_l,b_l;

 /* Creating argument list for the first thread  */
 a_l.n = 9;
 a_l.incx = 1;
 a_l.x = sx1;

 /* Creating argument list for the second thread  */
 b_l.n = 8;
 b_l.incx = 1;
 b_l.x = sx2;

 /*  Creating first thread which calls the ESSL subroutine ISAMAX  */
 rc = pthread_create(&first_th, NULL, Thread, (void *) &a_l);
 if (rc) exit(-1);

 /*  Creating second thread which calls the ESSL subroutine ISAMAX */
 rc = pthread_create(&second_th, NULL, Thread, (void *) &b_l);
 if (rc) exit(-1);

 sleep(1);
 exit(0);
}

/* Thread routine which call ESSL routine ISAMAX */
void *Thread(void *v)
{
 arg_list *al;
 float *x;
 int n,incx;
 int i;

 al = (arg_list *)(v);
 x = al->x;
 n = al->n;
 incx = al->incx;

 /* Calling the ESSL subroutine ISAMAX  */
  i = isamax(n,x,incx);
 if ( i == 8)
   printf("max for sx2 should be 8 = %d\n",i);
 else
   printf("max for sx1 should be 6 = %d\n",i);
 return NULL;
}
```

Handling Errors in Your C Program

ESSL provides you with flexibilities in handling both input-argument errors and computational errors:

- For input-argument errors 2015, 2030, and 2200, which are optionally-recoverable errors, ESSL allows you to obtain corrected input-argument values and react at run time.

  Note: In the case where error 2015 is unrecoverable, you have the option of dynamic allocation for most of the *aux* arguments. For details see the subroutine descriptions.

- For computational errors, ESSL provides a return code and additional information to help you analyze the problem in your program and react at run time.

"Input-Argument Errors in C" on page 147 and "Computational Errors in C" on page 151 explain how to use these facilities by describing the additional statements you must code in your program.

For multithreaded application programs, if you want to initialize the error option table and change the default settings for input-argument and computational errors, you need to implement the steps shown in "Input-Argument Errors in C" on page 147 and "Computational Errors in C" on page 151 on each thread that calls ESSL.

Input-Argument Errors in C

To obtain corrected input-argument values in a C program and to avert program termination for the optionally-recoverable input-argument errors 2015, 2030, and 2200, add the statements in the following steps to your program. Steps 4 and 8 for ERRSAV and ERRSTR, respectively, are optional. Adding these steps makes the effect of the call to ERRSET temporary.

Step 1. Code the Global Statements for ESSL Error Handling

```
/* Code two underscores */
/* before the letters ESVERR */
#define __ESVERR
#include <essl.h>
      extern int enotrm();
```

These statements are coded with your global declares in the front of your program. The `#define` must be coded before the `#include` statement for the ESSL header file. The `extern` statement declares the ESSL error exit routine ENOTRM as an external reference in your program. **After the point where you code these statements in your program, you must pass by reference all ESSL calling sequence arguments that can potentially be altered by ESSL error handling.** This applies to all your ESSL call statements. The two types of arguments are:

- *naux* arguments for auxiliary storage
- *n* arguments for transform lengths

Step 2. Declare the Variables

```
  int (*iusadr) ();
  int ierno,inoal,inomes,itrace,irange,irc,dummy;
  char storarea[8];
```

This declares a pointer, *iusadr*, to be used for the ESSL error exit routine ENOTRM. Also included are declares for the variables used by the ESSL and Fortran error-handling subroutines. Note that *storarea* must be 8 characters long. These should be coded in the beginning of your program before any of the following statements.

Step 3. Do Initialization for ESSL

```
iusadr = enotrm;
einfo (0,&dummy,&dummy);
```

The first statement sets the function pointer, *iusadr*, to ENOTRM, the ESSL error exit routine. The last statement calls the EINFO subroutine to initialize the ESSL error option table, where *dummy* is a declared integer and is a placeholder. For a description of EINFO, see "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188. These statements should be coded only once in the beginning of your program before calls to ERRSET.

Step 4. Call ERRSAV

```
  errsav (&ierno,storarea);
```

(This is an optional step.) This calls the ERRSAV subroutine, which stores the error option table entry for error number *ierno* in an 8-byte storage area, *storarea*, which is accessible to your program. ERRSAV must be called for each entry you want to save. This step is used, along with step 8, for ERRSTR. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 69. For an example, see "Computational Errors in C Example" on page 155, as the use is the same as for computational errors.

Step 5. Call ERRSET

```
errset (&ierno,&inoal,&inomes,&itrace,&iusadr,&irange);
```

This calls the ERRSET subroutine, which allows you to dynamically modify the action taken when an error occurs. For optionally-recoverable ESSL input-argument errors, you need to call ERRSET only if you want to avoid terminating your program and you want the input arguments associated with this error to be assigned correct values in your program when the error occurs. For one error (*ierno*) or a range of errors (*irange*), you can specify:

- How many times each error can occur before execution terminates (*inoal*)
- How many times each error message can be printed (*inomes*)
- The ESSL exit routine ENOTRM, to be invoked for the error indicated (*iusadr*)

ERRSET must be called for each error code you want to indicate as being recoverable. For ESSL, *ierno* should have a value of 2015, 2030, or 2200. If you want to eliminate error messages, you should indicate a negative number for *inomes*; otherwise, you should specify 0 for this argument. All the other ERRSET arguments should be specified as 0.

For a list of the default values set in the ESSL error option table, see "How Do You Control Error Handling by Setting Values in the ESSL Error Option Table?" on page 67. For a description of the input-argument errors, see "Input-Argument Error Messages(2001-2099)" on page 187. For a description of ERRSET, see Chapter 17, "Utilities subroutines," on page 1187.

Step 6. Call ESSL

```
irc = name (arg1,...,argn);
if irc == rc1
        {
           .
           .
           .
              }
```

This calls the ESSL subroutine and specifies a branch on one or more return code values, where:

- *name* specifies the ESSL subroutine.
- *arg1*,...,*argn* are the input and output arguments. As explained in step 1, all arguments that can potentially be altered by error handling must be coded by reference.
- *irc* is the integer variable containing the return code resulting from the computation performed by the ESSL subroutine.
- *rc1*, *rc2*, and so forth are the possible return code values that can be passed back from the ESSL subroutine to C. The values can be 0, 1, 2, and so forth. Return code values are described under "Error Conditions" in each ESSL subroutine description.

Step 7. Perform the Desired Action

These are the statements following the test for each value of the return code, returned in *irc* in step 6. These statements perform whatever action is desired when the recoverable error occurs. These statements may check the new values set in the input arguments to determine whether adequate program storage is available, and then decide whether to continue or terminate the program. Otherwise, these statements may check that the size of the working storage arrays or the length of the transform agrees with other data in the program. The program may also store this corrected input argument value for future reference.

Step 8. Call ERRSTR

```
errstr (&ierno,storarea);
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number *ierno* from an 8-byte storage area, *storarea*, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 4, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 69. For an example, see "Computational Errors in C Example" on page 155, as the use is the same as for computational errors.

Input-Argument Errors in C Example

This 32-bit integer, 64-bit pointer environment example shows an error code 2015, which resets the size of the work area *aux*, specified in *naux*, if the value specified is too small. It also indicates that no error messages should be issued.

```
   .
   .
   .
       /*GLOBAL STATEMENTS FOR ESSL ERROR HANDLING*/
   #define __ESVERR
   #include <essl.h>
   extern int enotrm();
   .
   .
   .

       /*DECLARE THE VARIABLES*/
   main ()
   {
   int (*iusadr) ();
   int ierno,inoal,inomes,itrace,irange,irc,dummy;
   int naux;
   .
   .
   .

      /*INITIALIZE THE POINTER TO THE ENOTRM ROUTINE*/
   iusadr = enotrm;
   .
   .
   .

       /*INITIALIZE THE ESSL ERROR OPTION TABLE*/
   einfo (0,&dummy,&dummy);
   .
   .
   .

       /*MAKE ERROR CODE 2015 A RECOVERABLE ERROR AND
         SUPPRESS PRINTING ALL ERROR MESSAGES FOR IT*/
   ierno = 2015;
   inoal = 0;
   inomes = -1;
   itrace = 0;
   irange = 2015;
   errset (&ierno,&inoal,&inomes,&itrace,&iusadr,&irange);
   .
   .
   .

       /*CALL ESSL SUBROUTINE SWLEV.  NAUX IS PASSED BY
         REFERENCE.  IF THE NAUX INPUT IS TOO SMALL,
         ERROR 2015 OCCURS.  THE MINIMUM VALUE REQUIRED
         IS STORED IN THE NAUX INPUT ARGUMENT, AND THE
         RETURN CODE OF 1 IS SET IN IRC.*/
   irc = swlev (x,incx,u,incu,y,incy,n,aux,&naux);
   if irc == 1

 {
   .    /*CHECK THE RESULTING INPUT ARGUMENT VALUE
   .      IN NAUX AND TAKE THE DESIRED ACTION*/
   .
   }

   .
   .
   .
   }
```

Computational Errors in C

To obtain information about an ESSL computational error in a C program, add the statements in the following steps to your program. Steps 4 and 9 for ERRSAV and ERRSTR, respectively, are optional. Adding these steps makes the effect of the call to ERRSET temporary. For a list of those computational errors that return information and to which these steps apply, see "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188.

Step 1. Code the Global Statements for ESSL Error Handling

```
/* Code two underscores */
/* before the letters ESVERR */
#define __ESVERR
#include <essl.h>
```

These statements are coded with your global declares in the front of your program. The `#define` must be coded before the `#include` statement for the ESSL header file. **After the point where you code these statements in your program, you must pass by reference all ESSL calling sequence arguments that can potentially be altered by ESSL error handling.** This applies to all your ESSL call statements. The two types of arguments are:

- *naux* arguments for auxiliary storage
- *n* arguments for transform lengths

Step 2. Declare the Variables

```
  int ierno,inoal,inomes,itrace,iusadr,irange,irc;
  int inf1,inf2,dummy;
  char storarea[8];
```

These statements include declares for the variables used by the ESSL and Fortran error-handling subroutines. Note that *storarea* must be 8 characters long. These should be coded in the beginning of your program before any of the following statements.

Step 3. Do Initialization for ESSL

```
einfo (0,&dummy,&dummy);
```

This statement calls the EINFO subroutine to initialize the ESSL error option table, where *dummy* is a declared integer and is a placeholder. For a description of EINFO, see "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188. These statements should be coded only once in the beginning of your program before calls to ERRSET.

Step 4. Call ERRSAV

```
  errsav (&ierno,storarea);
```

(This is an optional step.) This calls the ERRSAV subroutine, which stores the error option table entry for error number *ierno* in an 8-byte storage area, *storarea*, which is accessible to your program. ERRSAV must be called for each entry you want to save. This step is used, along with step 8, for ERRSTR. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 69. For an example, see "Computational Errors in C Example" on page 155.

Step 5. Call ERRSET

```
errset (&ierno,&inoal,&inomes,&itrace,&iusadr,&irange);
```

This calls the ERRSET subroutine, which allows you to dynamically modify the action taken when an error occurs. For ESSL computational errors, you need to call ERRSET only if you want to change the default values in the ESSL error option table. For one error (*ierno*) or a range of errors (*irange*), you can specify:

- How many times each error can occur before execution terminates (*inoal*)
- How many times each error message can be printed (*inomes*)

ERRSET must be called for each error code for which you want to change the default values. For ESSL, *ierno* should be set to one of the eligible values listed in "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188. To allow your program to continue after an error in the specified range occurs, *inoal* must be set to a value greater than 1. For ESSL, *iusadr* should be specified as either 0 or 1 in a 32-bit integer, 32-bit pointer environment (0l or 1l in a 32-bit integer, 64-bit pointer environment or a 64-bit integer, 64-bit pointer environment), so a user exit is not taken.

For a list of the default values set in the ESSL error option table, see "How Do You Control Error Handling by Setting Values in the ESSL Error Option Table?" on page 67. For a description of the computational errors, see "Computational Error Messages(2100-2199)" on page 191. For a description of ERRSET, see Chapter 17, "Utilities subroutines," on page 1187.

Step 6. Call ESSL

```
irc = name (arg1,...,argn);
if irc == rc1
        {
          .
          .
          .
             }
if irc == rc2
        {
          .
          .
          .
            }
```

This calls the ESSL subroutine and specifies a branch on one or more return code values, where:

- *name* specifies the ESSL subroutine.
- *arg1*,...,*argn* are the input and output arguments. As explained in step 1, all arguments that can potentially be altered by error handling must be coded by reference.
- *irc* is the integer variable containing the return code resulting from the computation performed by the ESSL subroutine.
- *rc1*, *rc2*, and so forth are the possible return code values that can be passed back from the ESSL subroutine to C. The values can be 0, 1, 2, and so forth. Return code values are described under "Error Conditions" in each ESSL subroutine description.

The statements following each test of the return code can perform any desired action. This includes calling EINFO for more information about the error, as described in step 7.

Step 7. Call EINFO for Information

```
einfo (ierno,&inf1,&inf2);
```

This calls the EINFO subroutine, which returns information about certain computational errors, where:

- *ierno* is the error code of interest.
- *inf1* and *inf2* are the integer variables used to receive the information, where *inf1* is assigned a value for all errors, and *inf2* is assigned a value for some errors. You must specify both arguments, as there are no optional arguments for C. Both arguments must be passed by reference, because they are output scalar arguments. For a description of EINFO, see "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188.

Step 8. Check the Values in the Information Receivers

These statements check the values returned in the output argument information receivers, *inf1* and *inf2*, which contain the information about the computational error.

Step 9. Call ERRSTR

```
errstr (&ierno,storarea);
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number *ierno* from an 8-byte storage area, *storarea*, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 4, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 69. For an example, see "Computational Errors in C Example" on page 155.

Computational Errors in C Example

This 32-bit integer, 64-bit pointer environment example shows an error code 2105, which returns one piece of information: the index of the pivot element (*i*) near zero, causing factorization to fail. It uses ERRSAV and ERRSTR to insulate the effects of the error handling for error 2105 by this program.

```
        .
        .
                    /*GLOBAL STATEMENTS FOR ESSL ERROR HANDLING*/
        #define __ESVERR
        #include <essl.h>
        .
        .
                    /*DECLARE THE VARIABLES*/
        main ()
        {
        int ierno,inoal,inomes,itrace,irange,irc;
        long int iusadr;
        int inf1,inf2,dummy;
        char sav2105[8];
        .
        .
                    /*INITIALIZE THE ESSL ERROR OPTION TABLE*/
        einfo (0,&dummy,&dummy);
                    /*SAVE THE EXISTING ERROR OPTION TABLE ENTRY
                      FOR ERROR CODE 2105*/
        ierno = 2105;
        errsav (&ierno,sav2105);
        .
        .
                    /*MAKE ERROR CODES 2101 THROUGH 2105 RECOVERABLE
                      ERRORS AND SUPPRESS PRINTING ALL ERROR MESSAGES
                      FOR THEM.  THIS SHOWS HOW YOU CODE THE
                      ERRSET ARGUMENTS FOR A RANGE OF ERRORS. */
        ierno = 2101;
        inoal = 0;
        inomes = 0; /*A DUMMY ARGUMENT*/
        itrace = 0; /*A DUMMY ARGUMENT*/
        iusadr = 0l; /*A DUMMY ARGUMENT*/
        irange = 2105;
        errset (&ierno,&inoal,&inomes,&itrace, &iusadr,&irange);
        .
        .
                    /*CALL ESSL SUBROUTINE DGEICD. IF THE INPUT MATRIX
                      IS SINGULAR OR NEARLY SINGULAR, ERROR 2105
                      OCCURS.  A RETURN CODE OF 2 IS SET IN IRC.*/
        irc = dgeicd (a,lda,n,iopt,&rcond,det,aux,&naux);
        if irc == 2
        {
                    /*CALL THE INFORMATION-HANDLER ROUTINE FOR ERROR
                      CODE 2105 TO RETURN ONE PIECE OF INFORMATION
                      IN VARIABLE INF1, THE INDEX OF THE PIVOT ELEMENT
                      NEAR ZERO, CAUSING FACTORIZATION TO FAIL.
                      INF2 IS NOT USED, BUT MUST BE SPECIFIED.
                      BOTH INF1 AND INF2 ARE PASSED BY REFERENCE,
                      BECAUSE THEY ARE OUTPUT SCALAR ARGUMENTS.*/
           ierno = 2105;
           einfo (ierno,&inf1,&inf2);
                    /*CHECK THE VALUE IN VARIABLE INF1 AND TAKE THE
                      DESIRED ACTION*/
           .
           .
           }
        .
        .
                    /*RESTORE THE PREVIOUS ERROR OPTION TABLE ENTRY
                      FOR ERROR CODE 2105.  ERROR PROCESSING
                      RETURNS TO HOW IT WAS BEFORE IT WAS ALTERED BY
                      THE ABOVE ERRSAV STATEMENT*/
        ierno = 2105;
        errstr (&ierno,sav2105);
        .
        .
        }
```

C++ Programs

This describes how to code your C++ program.

Calling ESSL Subroutines and Functions in C++

This shows how to call ESSL subroutines and functions from your C++ program.

Before You Call ESSL

Before you can call the ESSL subroutines from your C++ program, you must have the appropriate ESSL header file installed on your system. The ESSL header file allows you to code your function calls as described here. It contains entries for all the ESSL subroutines. The ESSL header file is distributed with the ESSL package. The ESSL header file to be used with the C++ compiler is named `essl.h`.

In the beginning of your program, before you call any of the ESSL subroutines, **you must code the following statement for the ESSL header file:**

```
   #include <essl.h>
```

If you are creating your own threads for the ESSL Thread-Safe or SMP Libraries, you must include the `pthread.h` header file in your C++ program. For an example, see "Creating Multiple Threads and Calling ESSL from Your C++ Program" on page 160.

Coding the Calling Sequences

In C++ programs, the ESSL subroutines, not returning a function value, are invoked with the following type of statement:

```
   subroutine-name (argument-1, . . . , argument-n);
```

An example of a calling sequence for SAXPY might be:

```
   saxpy (5,a,x,incx,y,1);
```

The ESSL subroutines returning a function value are invoked with the following type of statement:

```
function-value-name=subroutine-name (argument-1, . . . , argument-n);
```

An example of invoking DASUM might be:

```
   sum = dasum (n,x,incx);
```

See the C++ publications for details about how to code the function calls.

Passing Arguments in C++

This describes how to pass arguments in your C++ program.

About the Syntax Shown in this Documentation

The argument syntax shown assumes that you have installed and are using the ESSL header file. For further details, see "Calling ESSL Subroutines and Functions in C++" on page 155.

No Optional Arguments

In the ESSL calling sequences for C++, there are no optional arguments, as for some programming languages. You must code all the arguments listed in the syntax.

Arguments That Must Be Passed by Value

All scalar arguments that are not modified must be passed by value in the ESSL calling sequence. (This refers to input-only scalar arguments, such as *incx*, *m*, and *lda*.)

Arguments That Must Be Passed by Reference

Following are the instances in which you pass your arguments by reference (as a pointer) in the ESSL calling sequence:

Arrays

Arguments that are arrays are passed by reference, as usual.

Subroutine Names

Some ESSL subroutines call a user-supplied subroutine. The name is part of the ESSL calling sequence. It must be passed by reference.

Output Scalar Arguments

When an output scalar argument is a scalar data item, it must be passed by reference as shown below. This is true for all scalar data types: real, complex, and so forth.

The ESSL header file supports two alternatives:

- The arguments are declared to be type reference in the function prototype. This is the default. Following is an example of how you can call DURAND using this alternative:

  ```
  durand (seed, n, x);
  ```

- The arguments are declared as pointers in the function prototype. If you wish to use this alternative, you must define _ESVCPTR using one of the following methods:
  - Define _ESVCPTR in your program using a `#define` statement, as shown below:

    ```
    #define _ESVCPTR
    ```

    This statement is coded with your global declares and must be coded before the `#include` statement for the ESSL header file.
  - Define _ESVCPTR at compile time by using the job processing procedure described in "C++ Program Procedures" on page 174.

  Following is an example of how you can call DURAND using this alternative:

  ```
  durand (&seed, n, x);
  ```

Character Arguments

Character arguments must be passed as strings, by reference. You specify the character, in upper- or lowercase, in the ESSL calling sequence with double quotation marks around it, as in "t". Following is an example of how you can call SGEADD, specifying the *transa* and *transb* arguments as strings *n* and *t*, respectively:

```
  sgeadd (a,5,"n",b,3,"t",c,4,4,3);
```

Setting Up a User-Supplied Subroutine for ESSL in C++

Some ESSL numerical quadrature subroutines call a user-supplied subroutine, *subf*, identified in the ESSL calling sequence. If your program that calls the numerical quadrature subroutines is coded in C++, there are some coding rules you must follow for the *subf* subroutine:

- You can code the *subf* subroutine using only C, C++, or Fortran.
- You must declare *subf* as an external subroutine in your application program.
- You should code the *subf* subroutine to the specifications given in "Programming Considerations for the SUBF Subroutine" on page 1144. For an example of coding a *subf* subroutine in C++, see Example 1.

Setting Up Scalar Data in C++

Table 43 on page 158 lists the scalar data types in C++ that are used for ESSL. Only those types and lengths used by ESSL are listed.

Table 43: Scalar Data Types in C++ Programs.

Terminology Used by ESSL	**C++ Equivalent**
Character item[1] 'N', 'T', 'C' or 'n', 't', 'c'	char * “n”, “t”, “c”
32-bit logical item[7] .TRUE., .FALSE.	int For additional information, see “Using Logical Data in C++” on page 159.[2]
64-bit logical item[7] .TRUE., .FALSE.	long For additional information, see “Using Logical Data in C++” on page 159.[2]
32-bit integer 12345, -12345	int
64-bit integer[7] 12345l, -12345l	long
Short-precision real number[4] 12.345	float
Long-precision real number[4] 12.345	double
Short-precision complex number[4] (123.45, -54321.0)	complex <float>“5” on page 158, float_Complex[6]
Long-precision complex number[4] (123.45, -54321.0)	complex <double>“5” on page 158, double_Complex[6]

Note:

1. ESSL accepts character data in either upper- or lowercase in its calling sequences.
2. There are no equivalent data types for logical data in C++. These require special procedures. For details, see “Using Logical Data in C++” on page 159.
3. For a 32-bit integer, 64-bit pointer environment, in accordance with the LP64 data model, all ESSL integer arguments remain 32-bits except for the `iusadr` argument for ERRSET.
4. Short- and long-precision numbers look the same in this documentation.
5. This data type is defined in file <complex>.
6. This data type is defined in <complex.h>.
7. If you are using the ESSL header file in a 64-bit integer, 64-bit pointer environment, add -D_ESV6464 to your compiler command to define the integer and logical arguments as long.

Using Complex Data in C++

The ESSL header file supports both the Standard Numerics Library and C99 floating-point types for complex arithmetic (<`complex.h`>).

Selecting the <complex> or <complex.h> Header File

Although the header files <complex> and <complex.h> are similar in purpose, they are mutually incompatible and cannot be simultaneously used.

If you wish to use the C99 complex floating-point types for complex arithmetic, you must perform one of the following steps:

- Code the #include statement for the C99 complex floating point types (#include <complex.h>) in your program prior to coding the #include statement for the ESSL header file.
- Define _ESV_COMPLEX99_, using one of the following methods:
  - Define _ESV_COMPLEX99_ in your program using a #define statement, as shown below:

    ```
    #define _ESV_COMPLEX99_
    ```

    This statement is coded with your global declares and must be coded before the #include statement for the ESSL header file.
  - Define _ESV_COMPLEX99_ at compile time by using the job processing procedures described in Chapter 5, "Processing Your Program," on page 169.

If you take none of the preceding steps, the ESSL header file will use the Standard Numerics Library. The ESSL header file will also use the Standard Numerics Library if you code the #include statement for the Standard Numerics Library (#include<complex.h>) in your program prior to coding the #include statement for the ESSL header file.

Using Logical Data in C++

Logical data types are not part of the C++ language; however, some ESSL subroutines require arguments of these data types.

By coding the following simple macro definitions in your program, you can then use TRUE or FALSE in assigning values to or specifying any logical arguments passed to ESSL:

For 32-bit logical arguments
Use this macro definition:

```
    #define FALSE 0
    #define TRUE 1
```

For 64-bit logical arguments
Use this macro definition:

```
    #define FALSE 0l
    #define TRUE 1l
```

Setting Up Arrays in C++

C++ arrays are arranged in storage in row-major order. This means that the last subscript expression increases most rapidly, the next-to-the-last subscript expression increases less rapidly, and so forth, with the first subscript expression increasing least rapidly. ESSL subroutines require that arrays passed as arguments be in column-major order. This is the array storage convention used by Fortran, described in "Setting Up Arrays in Fortran" on page 126. To pass an array from your C++ program to ESSL, to have ESSL process the data correctly, and to get a result that is in the proper form for your C++ program, you can do any of the following:

- Build and process the matrix, logically transposed from the outset, and transpose the results as necessary.
- Before the ESSL call, transpose the input arrays. Then, following the ESSL call, transpose any arrays updated as output.
- If there are arguments in the ESSL calling sequence indicating whether the arrays are to be processed in normal or transposed form, such as the *transa* and *transb* arguments in the _GEMM subroutines, use

these arguments in combination with the matrix equivalence rules to avoid having to transpose your data in separate operations. For further detail, see "SGEMMS, DGEMMS, CGEMMS, and ZGEMMS (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm)" on page 408.

Creating Multiple Threads and Calling ESSL from Your C++ Program

The 32-bit integer, 64-bit pointer environment example shown below shows how to create two threads, where each thread calls the ISAMAX subroutine. To use the pthreads library, you must remember to code the `pthread.h` header file in your C++ program.

Note: Be sure to compile this program with the `xlC_r` command.

```
#include "essl.h"
#ifdef __linux
#include <iostream>
std::cout;
#else
#include <iostream.h>
#endif

/* Define prototype for thread routine */
void *Thread(void *v);

/* Define prototype for thread library routine, which is in C */
extern "C" {
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
}
/* Create structure for argument list  */
struct arg_list {
  int      n;
  float   *x;
  int   incx;
};
int main()
{
 float  sx1[9] = { 1., 2., 7., -8., -5., -10., -9., 10., 6. };
 float  sx2[8] = { 1.,12., 7., -8., -5., -10., -9., 19.};
 pthread_t first_th;
 pthread_t second_th;
 int rc;
 struct arg_list a_l,b_l;

 a_l.n = 9;
 a_l.incx = 1;
 a_l.x = sx1;

 b_l.n = 8;
 b_l.incx = 1;
 b_l.x = sx2;

 /* Creating argument list for first thread  */
 rc = pthread_create(&first_th, NULL, Thread, (void *) &a_l);
 if (rc) exit(-1);

 /* Creating argument list for second thread  */
 rc = pthread_create(&second_th, NULL, Thread, (void *) &b_l);
 if (rc) exit(-1);

 sleep(20);
 exit(0);
}
/* Thread routine which calls the ESSL subroutine ISAMAX */
void* Thread(void *v)
{
 struct arg_list *al;
 float *t;
 int n,incx;
 int i;

 al = (struct arg_list *)(v);
 t = al->x;
 n = al->n;
 incx = al->incx;
```

```
 /* Calling the ESSL subroutine ISAMAX  */
 i = isamax(n,t,incx);
 if ( i == 8)
   cout << "max for sx2 should be 8 = " << i << "\n";
 else
   cout << "max for sx1 should be 6 = " << i << "\n";
 return NULL;
}
```

Handling Errors in Your C++ Program

ESSL provides you with flexibilities in handling both input-argument errors and computational errors:

- For input-argument errors 2015, 2030, and 2200 which are optionally-recoverable errors, ESSL allows you to obtain corrected input-argument values and react at run time.

  Note: In the case where error 2015 is unrecoverable, you have the option of dynamic allocation for most of the *aux* arguments. For details see the subroutine descriptions.
- For computational errors, ESSL provides a return code and additional information to help you analyze the problem in your program and react at run time.

"Input-Argument Errors in C++" on page 161 and "Computational Errors in C++" on page 165 explain how to use these facilities by describing the additional statements you must code in your program.

For multithreaded application programs, if you want to initialize the error option table and change the default settings for input-argument and computational errors, you need to implement the steps shown in "Input-Argument Errors in C++" on page 161 and "Computational Errors in C++" on page 165 on each thread that calls ESSL.

Input-Argument Errors in C++

To obtain corrected input-argument values in a C++ program and to avert program termination for the optionally-recoverable input-argument errors 2015, 2030, and 2200, add the statements in the following steps to your program. Steps 4 and 8 for ERRSAV and ERRSTR, respectively, are optional. Adding these steps makes the effect of the call to ERRSET temporary.

Step 1. Code the Global Statements for ESSL Error Handling

```
/* Code one underscore */
/* before the letters ESVERR */
#define _ESVERR
#ifdef __linux
#include <iostream>
#else
#include <iostream.h>
#endif
#include <stdio.h>
#include <essl.h>
extern "C" int enotrm(int &,int &);
extern "C" typedef int (*FN) (int &,int &);
```

These statements are coded with your global declares in the front of your program. The `#define` must be coded before the `#include` statements for the ESSL header file. The `extern` statements are required to call the ESSL error exit routine ENOTRM as an external reference in your program.

Step 2. Declare the Variables

```
FN iusadr;
int ierno,inoal,inomes,itrace,irange,irc,dummy;
char storarea[8];
```

This declares a pointer, *iusadr*, to be used for the ESSL error exit routine ENOTRM. Also included are declares for the variables used by the ESSL and Fortran error-handling subroutines. Note that *storarea* must be 8 characters long. These should be coded in the beginning of your program before any of the following statements.

Step 3. Do Initialization for ESSL

```
iusadr = enotrm;
dummy = 0;
einfo (0,dummy,dummy);
```

The first statement sets the function pointer, *iusadr*, to ENOTRM, the ESSL error exit routine. The last statement calls the EINFO subroutine to initialize the ESSL error option table, where *dummy* is a declared integer and is a placeholder. For a description of EINFO, see "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188. These statements should be coded only once in the beginning of your program before calls to ERRSET.

Step 4. Call ERRSAV

```
errsav (ierno,storarea);
```

(This is an optional step.) This calls the ERRSAV subroutine, which stores the error option table entry for error number *ierno* in an 8-byte storage area, *storarea*, which is accessible to your program. ERRSAV must be called for each entry you want to save. This step is used, along with step 8, for ERRSTR. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 69. For an example, see "Computational Errors in C++ Example" on page 168, as the use is the same as for computational errors.

Step 5. Call ERRSET

```
errset (ierno,inoal,inomes,itrace,&iusadr,irange);
```

This calls the ERRSET subroutine, which allows you to dynamically modify the action taken when an error occurs. For optionally-recoverable ESSL input-argument errors, you need to call ERRSET only if you want to avoid terminating your program and you want the input arguments associated with this error to be assigned correct values in your program when the error occurs. For one error (*ierno*) or a range of errors (*irange*), you can specify:

- How many times each error can occur before execution terminates (*inoal*)
- How many times each error message can be printed (*inomes*)
- The ESSL exit routine ENOTRM, to be invoked for the error indicated (*iusadr*)

ERRSET must be called for each error code you want to indicate as being recoverable. For ESSL, *ierno* should have a value of 2015, 2030, or 2200. If you want to eliminate error messages, you should indicate a negative number for *inomes*; otherwise, you should specify 0 for this argument. All the other ERRSET arguments should be specified as 0.

For a list of the default values set in the ESSL error option table, see "How Do You Control Error Handling by Setting Values in the ESSL Error Option Table?" on page 67. For a description of the input-argument errors, see "Input-Argument Error Messages(2001-2099)" on page 187. For a description of ERRSET, see Chapter 17, "Utilities subroutines," on page 1187.

Step 6. Call ESSL

```
irc = name (arg1,...,argn);
if irc == rc1
    {
      .
      .
      .
        }
if irc == rc2
    {
      .
      .
      .
        }
```

This calls the ESSL subroutine and specifies a branch on one or more return code values, where:

- *name* specifies the ESSL subroutine.
- *arg1*,...,*argn* are the input and output arguments.
- *irc* is the integer variable containing the return code resulting from the computation performed by the ESSL subroutine.
- *rc1*, *rc2*, and so forth are the possible return code values that can be passed back from the ESSL subroutine to C++. The values can be 0, 1, 2, and so forth. Return code values are described under “Error Conditions” in each ESSL subroutine description.

Step 7. Perform the Desired Action

These are the statements following the test for each value of the return code, returned in *irc* in step 6. These statements perform whatever action is desired when the recoverable error occurs. These statements may check the new values set in the input arguments to determine whether adequate program storage is available, and then decide whether to continue or terminate the program. Otherwise, these statements may check that the size of the working storage arrays or the length of the transform agrees with other data in the program. The program may also store this corrected input argument value for future reference.

Step 8. Call ERRSTR

```
errstr (ierno,storarea);
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number *ierno* from an 8-byte storage area, *storarea*, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 4, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see “ How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?” on page 69. For an example, see “Computational Errors in C++ Example” on page 168, as the use is the same as for computational errors.

Input-Argument Errors in C++ Example

This 32-bit integer, 64-bit pointer environment example shows an error code 2015, which resets the size of the work area *aux*, specified in *naux*, if the value specified is too small. It also indicates that no error messages should be issued.

```
  .
  .
  .
      /*GLOBAL STATEMENTS FOR ESSL ERROR HANDLING*/
  #define _ESVERR
  #include <essl.h>
  #ifdef __linux
  #include <iostream>
  #else
  #include <iostream.h>
  #endif
  #include <stdio.h>
  extern “C” int enotrm(int &,int &);
  extern “C” typedef int (*FN) (int &,int &);
  .
  .
  .
      /*DECLARE THE VARIABLES*/
  int main ()
  {
  FN iusadr;
  int ierno,inoal,inomes,itrace,irange,irc,dummy;
  int naux;
  .
  .
  .
     /*INITIALIZE THE POINTER TO THE ENOTRM ROUTINE*/
  iusadr = enotrm;
  .
  .
  .
/*INITIALIZE THE ESSL ERROR OPTION TABLE*/
  dummy = 0;
  einfo (0,dummy,dummy);
  .
  .
  .
      /*MAKE ERROR CODE 2015 A RECOVERABLE ERROR AND
        SUPPRESS PRINTING ALL ERROR MESSAGES FOR IT*/
  ierno = 2015;
  inoal = 0;
  inomes = -1;
  itrace = 0;
  irange = 2015;
  errset (ierno,inoal,inomes,itrace,&iusadr,irange);
  .
  .
  .
      /*CALL ESSL SUBROUTINE SWLEV.  NAUX IS PASSED BY
        REFERENCE.  IF THE NAUX INPUT IS TOO SMALL,
        ERROR 2015 OCCURS.  THE MINIMUM VALUE REQUIRED
        IS STORED IN THE NAUX INPUT ARGUMENT, AND THE
        RETURN CODE OF 1 IS SET IN IRC.*/
  irc = swlev (x,incx,u,incu,y,incy,n,aux,naux);
  if irc == 1

{
  .    /*CHECK THE RESULTING INPUT ARGUMENT VALUE
  .      IN NAUX AND TAKE THE DESIRED ACTION*/
  .
  }

  .
  .
  .
  }
```

Computational Errors in C++

To obtain information about an ESSL computational error in a C++ program, add the statements in the following steps to your program. Steps 4 and 9 for ERRSAV and ERRSTR, respectively, are optional. Adding these steps makes the effect of the call to ERRSET temporary. For a list of those computational errors that return information and to which these steps apply, see "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188.

Step 1. Code the Global Statements for ESSL Error Handling

```
/* Code one underscore */
/* before the letters ESVERR */
#define _ESVERR
#ifdef __linux
#include <iostream>
#else
#include <iostream.h>
#endif
#include <stdio.h>
#include <essl.h>
```

These statements are coded with your global declares in the front of your program. The `#define` must be coded before the `#include` statement for the ESSL header file.

Step 2. Declare the Variables

```
int ierno,inoal,inomes,itrace,iusadr,irange,irc;
int inf1,inf2,dummy;
char storarea[8];
```

These statements include declares for the variables used by the ESSL and Fortran error-handling subroutines. Note that *storarea* must be 8 characters long. These should be coded in the beginning of your program before any of the following statements.

Step 3. Do Initialization for ESSL

```
dummy = 0;
einfo (0,dummy,dummy);
```

The last statement calls the EINFO subroutine to initialize the ESSL error option table, where *dummy* is a declared integer and is a placeholder. For a description of EINFO, see "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188. These statements should be coded only once in the beginning of your program before calls to ERRSET.

Step 4. Call ERRSAV

```
errsav (ierno,storarea);
```

(This is an optional step.) This calls the ERRSAV subroutine, which stores the error option table entry for error number *ierno* in an 8-byte storage area, *storarea*, which is accessible to your program. ERRSAV must be called for each entry you want to save. This step is used, along with step 8, for ERRSTR. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 69. For an example, see "Computational Errors in C++ Example" on page 168.

Step 5. Call ERRSET

```
errset (ierno,inoal,inomes,itrace,&iusadr,irange);
```

This calls the ERRSET subroutine, which allows you to dynamically modify the action taken when an error occurs. For ESSL computational errors, you need to call ERRSET only if you want to change the default values in the ESSL error option table. For one error (*ierno*) or a range of errors (*irange*), you can specify:

- How many times each error can occur before execution terminates (*inoal*)
- How many times each error message can be printed (*inomes*)

ERRSET must be called for each error code for which you want to change the default values. For ESSL, *ierno* should be set to one of the eligible values listed in "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188. To allow your program to continue after an error in the specified range occurs, *inoal* must be set to a value greater than 1. For ESSL, *iusadr* should be specified as either 0 or 1 in a 32-bit environment (0l or 1l in a 32-bit integer, 64-bit pointer environment or a 64-bit integer, 64-bit pointer environment), so a user exit is not taken.

For a list of the default values set in the ESSL error option table, see "How Do You Control Error Handling by Setting Values in the ESSL Error Option Table?" on page 67. For a description of the computational errors, see "Computational Error Messages(2100-2199)" on page 191. For a description of ERRSET, see Chapter 17, "Utilities subroutines," on page 1187.

Step 6. Call ESSL

```
irc = name (arg1,...,argn);
if irc == rc1
    {
      .
      .
      .
        }
if irc == rc2
    {
      .
      .
      .
        }
```

This calls the ESSL subroutine and specifies a branch on one or more return code values, where:

- *name* specifies the ESSL subroutine.
- *arg1*,...,*argn* are the input and output arguments.
- *irc* is the integer variable containing the return code resulting from the computation performed by the ESSL subroutine.
- *rc1*, *rc2*, and so forth are the possible return code values that can be passed back from the ESSL subroutine to C++. The values can be 0, 1, 2, and so forth. Return code values are described under "Error Conditions" in each ESSL subroutine description.

The statements following each test of the return code can perform any desired action. This includes calling EINFO for more information about the error, as described in step 7.

Step 7. Call EINFO for Information

```
einfo (ierno,inf1,inf2);
```

This calls the EINFO subroutine, which returns information about certain computational errors, where:

- *ierno* is the error code of interest.
- *inf1* and *inf2* are the integer variables used to receive the information, where *inf1* is assigned a value for all errors, and *inf2* is assigned a value for some errors. You must specify both arguments, as there are no optional arguments for C. For a description of EINFO, see "EINFO (ESSL Error Information-Handler Subroutine)" on page 1188.

Step 8. Check the Values in the Information Receivers

These statements check the values returned in the output argument information receivers, *inf1* and *inf2*, which contain the information about the computational error.

Step 9. Call ERRSTR

```
errstr (ierno,storarea);
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number *ierno* from an 8-byte storage area, *storarea*, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 4, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 69. For an example, see "Computational Errors in C++ Example" on page 168.

Computational Errors in C++ Example

This 32-bit integer, 64-bit pointer environment example shows an error code 2105, which returns one piece of information: the index of the pivot element (*i*) near zero, causing factorization to fail. It uses ERRSAV and ERRSTR to insulate the effects of the error handling for error 2105 by this program.

```
.
.
            /*GLOBAL STATEMENTS FOR ESSL ERROR HANDLING*/
#define _ESVERR
#include <essl.h>
#if defined(__linux)
#include <iostream>
#else
#include <iostream.h>
#endif
#include <stdio.h>
.
.
            /*DECLARE THE VARIABLES*/
int main ()
{
int ierno,inoal,inomes,itrace,irange,irc;
long int iusadr;
int inf1,inf2,dummy;
char sav2105[8];
.
.
             /*INITIALIZE THE ESSL ERROR OPTION TABLE*/
dummy = 0;
       einfo (0,dummy,dummy);
            /*SAVE THE EXISTING ERROR OPTION TABLE ENTRY
              FOR ERROR CODE 2105*/
ierno = 2105;
errsav (ierno,sav2105);
.
.
             /*MAKE ERROR CODES 2101 THROUGH 2105 RECOVERABLE
              ERRORS AND SUPPRESS PRINTING ALL ERROR MESSAGES
              FOR THEM.  THIS SHOWS HOW YOU CODE THE
              ERRSET ARGUMENTS FOR A RANGE OF ERRORS. */
ierno = 2101;
inoal = 0;
inomes = 0; /*A DUMMY ARGUMENT*/
itrace = 0; /*A DUMMY ARGUMENT*/
iusadr = 0l; /*A DUMMY ARGUMENT*/
irange = 2105;
errset (ierno,inoal,inomes,itrace,&iusadr,irange);
.
.
            /*CALL ESSL SUBROUTINE DGEICD. IF THE INPUT MATRIX
              IS SINGULAR OR NEARLY SINGULAR, ERROR 2105
              OCCURS.  A RETURN CODE OF 2 IS SET IN IRC.*/
irc = dgeicd (a,lda,n,iopt,rcond,det,aux,naux);
if irc == 2
   {
            /*CALL THE INFORMATION-HANDLER ROUTINE FOR ERROR
              CODE 2105 TO RETURN ONE PIECE OF INFORMATION
              IN VARIABLE INF1, THE INDEX OF THE PIVOT ELEMENT
              NEAR ZERO, CAUSING FACTORIZATION TO FAIL.
              INF2 IS NOT USED, BUT MUST BE SPECIFIED.
              BOTH INF1 AND INF2 ARE PASSED BY REFERENCE,
              BECAUSE THEY ARE OUTPUT SCALAR ARGUMENTS.*/
   ierno = 2105;
   einfo (ierno,inf1,inf2);
            /*CHECK THE VALUE IN VARIABLE INF1 AND TAKE THE
              DESIRED ACTION*/
   .
   .
   }
```

```
   .
   .
            /*RESTORE THE PREVIOUS ERROR OPTION TABLE ENTRY
              FOR ERROR CODE 2105.  ERROR PROCESSING
              RETURNS TO HOW IT WAS BEFORE IT WAS ALTERED BY
              THE ABOVE ERRSAV STATEMENT*/
   ierno = 2105;
   errstr (ierno,sav2105);
   .
   .
   }
```

Chapter 5. Processing Your Program

You must make ESSL-specific changes to your job procedures for compiling, linking, and running your program. You can use any procedures you are currently using to compile, link, and run your Fortran, C, and C++ programs, as long as you make the necessary modifications required by ESSL.

Consider the following notes when you process your program:

Notes:

1. The default search paths for the ESSL shared libraries are as follows:

Environment	Shared Library Default Search Path
32-bit integer, 64-bit pointer	`/usr/lib64`
64-bit integer, 64-bit pointer	`/usr/lib64`

   If the shared libraries are in another location, you must set the link-time and run-time library search paths. There are two ways to set these search paths:

   - Use one of the following compile/link options:

     `-R` (or `-rpath`)
     Writes the specified run-time library search paths into the executable program.

     `-L`
     Searches the library search paths at link time, but does not write them into the executable as run-time library search paths.

     —or—

   - Use one of the following environment variables:

     `LD_LIBRARY_PATH`
     Specifies the directories that are to be searched for libraries at run time.

     `LD_RUN_PATH`
     Specifies the directories that are to be searched for libraries at both link and run time.

     For example, if you copied the ESSL 32-bit/64-bit pointer libraries in /home/me/lib64, you would issue commands similar to the following in order to compile and link a program:

```
LD_LIBRARY_PATH=/home/me/lib64:$LD_LIBRARY_PATH
LD_RUN_PATH=/home/me/lib64:$LD_RUN_PATH
export LD_LIBRARY_PATH
export LD_RUN_PATH
xlf_r -o myprog myprog.f -lessl
```

     The result would be that the /home/me/lib64 directory is the directory that gets searched at link time and run time.

   For more information on link options and environment variables, see the man page for the `ld` command

2. The default search path for the ESSL header files is `/usr/include`. If the header files are in another location, you must set the compile-time search path by setting the search path with -I.
3. If you changed Makefile or Makefile.gcc to install the FFTW Wrapper libraries in /usr/local/lib instead of /usr/local/lib64 (see Appendix C, "FFTW Version 3.1.2 to ESSL Wrapper Libraries," on page 1239), then you must specify -L/usr/local/lib instead of -L/usr/local/lib64 in the commands in Table 45 on page 171, Table 47 on page 172, Table 48 on page 173, and Table 51 on page 175.
4. For the ESSL SMP and SMP CUDA Libraries, you can use the `XLSMPOPTS` or `OMP_NUM_THREADS` environment variable to specify options which affect SMP execution. For details, see the IBM Compiler publications.

5. If you are accessing ESSL from a 64-bit integer, 64-bit pointer environment program, you may want to use the `-qintsize=8` compiler option.
6. ESSL supports the XL Fortran compile-time option `-qextname`. For details, see the Fortran publications.
7. Fortran 90 programmers may be interested in the `-qessl` compiler option which allows the use of ESSL routines in place of Fortran 90 intrinsic procedures. For details, see the Fortran manuals.
8. The commands in the table below assume that you installed:
   - The IBM compilers in the default directory, `/opt/ibm`. If you used different directories, you need to make the appropriate changes to the `-L` and `-R` options.
   - ESSL in the default directory `/opt/ibmmath`. If you used different directories, you need to make the appropriate changes to the `-I`, `-L`, and `-R` options.
   - The CUDA SDK in the default directory, /usr/local/cuda. If you used different directories, you need to make the appropriate changes to the -L and -R options.
9. In your job procedures, you must use only the required software products listed in Software requirements.

Fortran Program Procedures

You do not need to modify your existing Fortran compilation procedures when using ESSL.

When linking and running your program, you must modify your existing job procedures for ESSL in order to set up the necessary libraries.

If you are accessing ESSL from a Fortran program, you can compile and link using the commands shown in the table below.

Note: ESSL supports the XL Fortran compile-time option **-qextname**. For details, see the Fortran manuals.

Table 44: Fortran Compile Commands.

ESSL Library	Environment	Fortran Compile Command
Serial	32-bit integer, 64-bit pointer	`xlf_r  -O  -qnosave  xyz.f  -lessl`
Serial	64-bit integer, 64-bit pointer	`xlf_r  -O  -qnosave  xyz.f  -lessl6464`

SMP	32-bit integer, 64-bit pointer	`xlf_r  -O  -qnosave  -qsmp xyz.f  -lesslsmp`
		`xlf_r  -O  -qnosave  xyz.f  -lesslsmp  -lxlsmp`

SMP	64-bit integer, 64-bit pointer	`xlf_r  -O  -qnosave  -qsmp xyz.f  -lesslsmp6464`
		`xlf_r  -O  -qnosave  xyz.f  -lesslsmp6464  -lxlsmp`

SMP CUDA	32-bit integer, 64-bit pointer	`xlf_r  -O  -qnosave  -qsmp xyz.f  -lesslsmpcuda -lcublas -lcudart`
		`-L/usr/local/cuda/lib64`
		`-R/usr/local/cuda/lib64`
		`xlf_r  -O  -qnosave  xyz.f  -lesslsmpcuda  -lxlsmp -lcublas -lcudart`
		`-L/usr/local/cuda/lib64`
		`-R/usr/local/cuda/lib64`

where *xyz*.f is the name of your Fortran program.

If you want to use the FFTW Wrapper libraries with your Fortran program, the header file fftw3.f contains the constant definitions used by the FFTW wrappers. To use these defintions, you can do one of the following:

- Add the following line to your Fortran application

  ```
  include "fftw3.f"
  ```
- Imbed the fftw3.f header file in your application.

You can compile and link with the FFTW Wrapper libraries using the command shown in the table below (assuming that the FFTW Wrapper header files were installed in /usr/local/include).

Table 45: Fortran Compile Commands on Linux for Use with FFTW Wrapper Libraries.

ESSL Library	Environment	Fortran Compile Command
Serial	32-bit integer, 64-bit pointer	`xlf_r -O -qnosave xyz.f -lessl -I/usr/local/include -lfftw3_essl -L/usr/local/lib64`
SMP	32-bit integer, 64-bit pointer	`xlf_r -O -qnosave xyz.f -lesslsmp -I/usr/local/include -lfftw3_essl -L/usr/local/lib64`

For additional information on the FFTW Wrapper libraries, see Appendix C, "FFTW Version 3.1.2 to ESSL Wrapper Libraries," on page 1239.

C Program Procedures

If you are using the ESSL header file in a 64-bit integer, 64-bit pointer environment, add -D_ESV6464 to your compile and link command.

When linking and running your program, you must modify your existing job procedures for ESSL in order to set up the necessary libraries.

Table 46: C Compile and Link Commands.

ESSL Library	Environment	C Compile Command
Serial	32-bit integer, 64-bit pointer	`cc_r -O xyz.c -lessl -lxlf90_r -lxlfmath -L/opt/ibm/xlsmp/xlsmp_version.release/lib -L/opt/ibm/xlf/xlf_version.release/lib -R/opt/ibm/lib`
	64-bit integer, 64-bit pointer	`cc_r -O -D_ESV6464 xyz.c -lessl6464 -lxlf90_r -lxlfmath -L/opt/ibm/xlsmp/xlsmp_version.release/lib -L/opt/ibm/xlf/xlf_version.release/lib -R/opt/ibm/lib`
SMP	32-bit integer, 64-bit pointer	`cc_r -O xyz.c -lesslsmp -lxlf90_r -lxlsmp -lxlfmath -L/opt/ibm/xlsmp/xlsmp_version.release/lib -L/opt/ibm/xlf/xlf_version.release/lib -R/opt/ibm/lib`
	64-bit integer, 64-bit pointer	`cc_r -O -D_ESV6464 xyz.c -lesslsmp6464 -lxlf90_r -lxlsmp -lxlfmath -L/opt/ibm/xlsmp/xlsmp_version.release/lib -L/opt/ibm/xlf/xlf_version.release/lib -R/opt/ibm/lib`

Table 46: C Compile and Link Commands. (continued)

ESSL Library	Environment	C Compile Command
SMP CUDA	32-bit integer, 64-bit pointer	`cc_r -O xyz.c -lesslsmpcuda -lxlf90_r -lxlsmp -lxlfmath -lcublas -lcudart -L/usr/local/cuda/lib64 -R/usr/local/cuda/lib64 -L/opt/ibm/xlsmp/xlsmp_version.release/lib -L/opt/ibm/xlf/xlf_version.release/lib -R/opt/ibm/lib`

Note: In the commands listed in the table above, you must specify the following values:

xlf_version.release
: >|`15.1.6` or `16.1.0`|<

xlsmp_version.release
: >|`4.1.6` or `5.1.0`|<

If you want to use the FFTW Wrapper libraries with your C program, you must use header file fftw3_essl.h instead of fftw3.h. You can compile and link with the FFTW Wrapper libraries using the command shown in the table below (assuming that the FFTW Wrapper header files were installed in /usr/local/include).

Table 47: C Compile and Link Commands for Use with FFTW Wrapper Libraries.

ESSL Library	Environment	C Compile Command
Serial	32-bit integer, 64-bit pointer	`cc_r -O xyz.c -lessl -lxlf90_r -lxlfmath -L/opt/ibm/xlsmp/xlsmp_version.release/lib -L/opt/ibm/xlf/xlf_version.release/lib -R/opt/ibm/lib -I/usr/local/include -lfftw3_essl -L/usr/local/lib64`
SMP	32-bit integer, 64-bit pointer	`cc_r -O xyz.c -lesslsmp -lxlf90_r -lxlsmp -lxlfmath -L/opt/ibm/xlsmp/xlsmp_version.release/lib -L/opt/ibm/xlf/xlf_version.release/lib -R/opt/ibm/lib -I/usr/local/include -lfftw3_essl -L/usr/local/lib64`

Note: In the commands listed in the table above, you must specify the following values:

xlf_version.release
: >|`15.1.6` or `16.1.0`|<

xlsmp_version.release
: >|`4.1.6` or `5.1.0`|<

For additional information on the FFTW Wrapper libraries, see Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1239.

If you want to use gcc compile and link commands, use the commands shown in Table 48 on page 173

Table 48: gcc Compile and Link Commands.

ESSL Library	Environment	C Compile Command

Serial	32-bit integer, 64-bit pointer	`gcc   xyz.c`
		`      -lessl -lxlf90_r -lxlfmath -lm`
		`      -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`      -L/opt/ibm/xlf/xlf_version.release/lib`
		`      -R/opt/ibm/lib`

	64-bit integer, 64-bit pointer	`gcc   -D_ESV6464  xyz.c`
		`      -lessl6464 -lxlf90_r -lxlfmath -lm`
		`      -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`      -L/opt/ibm/xlf/xlf_version.release/lib`
		`      -R/opt/ibm/lib`

SMP*	32-bit integer, 64-bit pointer	`gcc   xyz.c`
		`      -lesslsmp -lxlf90_r -lxlsmp -lxlfmath -lm`
		`      -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`      -L/opt/ibm/xlf/xlf_version.release/lib`
		`      -R/opt/ibm/lib`

	64-bit integer, 64-bit pointer	`gcc   -D_ESV6464 xyz.c`
		`      -lesslsmp6464 -lxlf90_r -lxlsmp -lxlfmath -lm`
		`      -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`      -L/opt/ibm/xlf/xlf_version.release/lib`
		`      -R/opt/ibm/lib`

SMP CUDA*	32-bit integer, 64-bit pointer	`gcc   xyz.c`
		`      -lesslsmpcuda -lxlf90_r -lxlsmp -lxlfmath -lm -lcublas -lcudart`
		`      -L/usr/local/cuda/lib64`
		`      -R/usr/local/cuda/lib64`
		`      -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`      -L/opt/ibm/xlf/xlf_version.release/lib`
		`      -R/opt/ibm/lib`

* The ESSL SMP libraries require XL OpenMP runtime. The gcc OpenMP runtime is not compatible with XL OpenMP runtime.

Note: In the commands listed in Table 48 on page 173, you must specify the following values:

xlf_version.release
: `15.1.6` or `16.1.0`

xlsmp_version.release
: `4.1.6` or `5.1.0`

If you want to use the FFTW Wrapper libraries with your C program, you must use header file fftw3_essl.h instead of fftw3.h. You can compile and link with the FFTW Wrapper libraries using the command shown in the table below (assuming that the FFTW Wrapper header files were installed in /usr/local/include).

Table 49: gcc Compile and Link Commands for Use with FFTW Wrapper Libraries.

ESSL Library	Environment	C Compile Command

Serial	32-bit integer, 64-bit pointer	`gcc   xyz.c`
		`     -lessl  -lxlf90_r -lxlfmath -lm`
		`     -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`     -L/opt/ibm/xlf/xlf_version.release/lib`
		`     -R/opt/ibm/lib`
		`     -I/usr/local/include -lfftw3_essl -L/usr/local/lib64`

SMP*	32-bit integer, 64-bit pointer	`gcc   xyz.c`
		`     -lesslsmp  -lxlf90_r -lxlsmp -lxlfmath -lm`
		`     -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`     -L/opt/ibm/xlf/xlf_version.release/lib`
		`     -R/opt/ibm/lib`
		`     -I/usr/local/include -lfftw3_essl -L/usr/local/lib64`

* The ESSL SMP libraries require XL OpenMP runtime. The gcc OpenMP runtime is not compatible with XL OpenMP runtime.		

Note: In the commands listed in Table 49 on page 174, you must specify the following values:

xlf_version.release
: `15.1.6` or `16.1.0`

xlsmp_version.release
: `4.1.6` or `5.1.0`

C++ Program Procedures

The ESSL header file supports two alternatives for handling complex floating-point arguments. By default the Standard Numerics Library complex floating-point types are used. If you prefer to use the C99 complex floating-point types, add `-D_ESV_COMPLEX99_` to your compile and link commands.

The ESSL header file supports two alternatives for declaring scalar output arguments. By default, the arguments are declared to be type reference. If you prefer for them to be declared as pointers, add `-D_ESVCPTR` to your compile and link commands.

If you are using the ESSL header file in a 64-bit integer, 64-bit pointer environment, add `-D_ESV6464` to your compile and link command.

When linking and running your program, you must modify your existing job procedures for ESSL, to set up the necessary libraries.

Table 50: C++ Compile and Link Commands.

ESSL Library	Environment	C++ Compile Command

Serial	32-bit integer, 64-bit pointer	`xlC_r  -O  xyz.C`
		`       -lessl -lxlf90_r  -lxlfmath`
		`       -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`       -L/opt/ibm/xlf/xlf_version.release/lib`
		`       -R/opt/ibm/lib`

	64-bit integer, 64-bit pointer	`xlC_r  -O  -D_ESV6464 xyz.C`
		`       -lessl6464 -lxlf90_r  -lxlfmath`
		`       -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`       -L/opt/ibm/xlf/xlf_version.release/lib`
		`       -R/opt/ibm/lib`

Table 50: C++ Compile and Link Commands. (continued)

ESSL Library	Environment	C++ Compile Command

SMP	32-bit integer, 64-bit pointer	`xlC_r  -O  xyz.C`
		`       -lesslsmp -lxlf90_r -lxlsmp -lxlfmath`
		`       -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`       -L/opt/ibm/xlf/xlf_version.release/lib`
		`       -R/opt/ibm/lib`

	64-bit integer, 64-bit pointer	`xlC_r  -O  -D_ESV6464  xyz.C`
		`       -lesslsmp6464 -lxlf90_r -lxlsmp -lxlfmath`
		`       -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`       -L/opt/ibm/xlf/xlf_version.release/lib`
		`       -R/opt/ibm/lib`

SMP CUDA	32-bit integer, 64-bit pointer	`xlC_r  -O  xyz.C`
		`       -lesslsmpcuda -lxlf90_r -lxlsmp -lxlfmath -lcublas -`
		`lcudart`
		`       -L/usr/local/cuda/lib64`
		`       -R/usr/local/cuda/lib64`
		`       -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`       -L/opt/ibm/xlf/xlf_version.release/lib`
		`       -R/opt/ibm/lib`

Note: In the commands listed in the table above, you must specify the following values:

xlf_version.release
: `15.1.6` or `16.1.0`

xlsmp_version.release
: `4.1.6` or `5.1.0`

If you want to use the FFTW Wrapper libraries with your C++ program, you must use header file fftw3_essl.h instead of fftw3.h. You can compile and link with the FFTW Wrapper libraries using the command shown in the table below (assuming that the FFTW Wrapper header files were installed in /usr/local/include).

Table 51: C++ Compile and Link Commands for Use with FFTW Wrapper Libraries.

ESSL Library	Environment	C++ Compile Command

Serial	32-bit integer, 64-bit pointer	`xlC_r -O xyz.C`
		`     -lessl -lxlf90_r -lxlfmath -lm`
		`     -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`     -L/opt/ibm/xlf/xlf_version.release/lib`
		`     -R/opt/ibm/lib`
		`     -I/usr/local/include -lfftw3_essl -L/usr/local/lib64`

SMP	32-bit integer, 64-bit pointer	`xlC_r -O xyz.C`
		`     -lesslsmp -lxlf90_r -lxlsmp -lxlfmath -lm`
		`     -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`     -L/opt/ibm/xlf/xlf_version.release/lib`
		`     -R/opt/ibm/lib`
		`     -I/usr/local/include -lfftw3_essl -L/usr/local/lib64`

Note: In the commands listed in the table above, you must specify the following values:

xlf_version.release
: `15.1.6` or `16.1.0`

xlsmp_version.release
: `4.1.6` or `5.1.0`

For additional information on the FFTW Wrapper libraries, see Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1239.

If you want to use g++ compile and link commands, use the commands shown in Table 52 on page 176

Table 52: g++ Compile and Link Commands.

ESSL Library	Environment	C++ Compile Command

Serial	32-bit integer, 64-bit pointer	`g++   xyz.C`
		`      -lessl -lxlf90_r -lxlfmath -lm`
		`      -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`      -L/opt/ibm/xlf/xlf_version.release/lib`
		`      -R/opt/ibm/lib`

	64-bit integer, 64-bit pointer	`g++   -D_ESV6464  xyz.C`
		`      -lessl6464 -lxlf90_r -lxlfmath -lm`
		`      -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`      -L/opt/ibm/xlf/xlf_version.release/lib`
		`      -R/opt/ibm/lib`

SMP*	32-bit integer, 64-bit pointer	`g++   xyz.C`
		`      -lesslsmp -lxlf90_r -lxlsmp -lxlfmath -lm`
		`      -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`      -L/opt/ibm/xlf/xlf_version.release/lib`
		`      -R/opt/ibm/lib`

	64-bit integer, 64-bit pointer	`g++   -D_ESV6464 xyz.C`
		`      -lesslsmp6464 -lxlf90_r -lxlsmp -lxlfmath -lm`
		`      -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`      -L/opt/ibm/xlf/xlf_version.release/lib`
		`      -R/opt/ibm/lib`

SMP CUDA*	32-bit integer, 64-bit pointer	`g++   xyz.C`
		`      -lesslsmpcuda -lxlf90_r -lxlsmp -lxlfmath -lm -`
		`lcublas -lcudart`
		`      -L/usr/local/cuda/lib64`
		`      -R/usr/local/cuda/lib64`
		`      -L/opt/ibm/xlsmp/xlsmp_version.release/lib`
		`      -L/opt/ibm/xlf/xlf_version.release/lib`
		`      -R/opt/ibm/lib`

* The ESSL SMP libraries require XL OpenMP runtime. The gcc OpenMP runtime is not compatible with XL OpenMP runtime.		

Note: In the commands listed in Table 52 on page 176, you must specify the following values:

xlf_version.release
: `15.1.6` or `16.1.0`

xlsmp_version.release
: `4.1.6` or `5.1.0`

If you want to use the FFTW Wrapper libraries with your C program, you must use header file fftw3_essl.h instead of fftw3.h. You can compile and link with the FFTW Wrapper libraries using the command shown in the table below (assuming that the FFTW Wrapper header files were installed in /usr/local/include).

Table 53: g++ Compile and Link Commands for Use with FFTW Wrapper Libraries.		
ESSL Library	**Environment**	**C++ Compile Command**
Serial	32-bit integer, 64-bit pointer	`g++   xyz.C -lessl  -lxlf90_r -lxlfmath -lm -L/opt/ibm/xlsmp/`*`xlsmp_version.release`*`/lib -L/opt/ibm/xlf/`*`xlf_version.release`*`/lib -R/opt/ibm/lib -I/usr/local/include -lfftw3_essl -L/usr/local/lib64`
SMP*	32-bit integer, 64-bit pointer	`g++   xyz.C -lesslsmp  -lxlf90_r -lxlsmp -lxlfmath -lm -L/opt/ibm/xlsmp/`*`xlsmp_version.release`*`/lib -L/opt/ibm/xlf/`*`xlf_version.release`*`/lib -R/opt/ibm/lib -I/usr/local/include -lfftw3_essl -L/usr/local/lib64`
* The ESSL SMP libraries require XL OpenMP runtime. The gcc OpenMP runtime is not compatible with XL OpenMP runtime.		

Note: In the commands listed in Table 53 on page 177, you must specify the following values:

xlf_version.release
: >|`15.1.6` or `16.1.0`|<

xlsmp_version.release
: >|`4.1.6` or `5.1.0`|<

Chapter 6. Migrating programs to ESSL

This explains what is required to migrate your application programs to the current release of ESSL.

>|Migrating Programs from ESSL for Linux on Power Version 5 Release 5 to Version 6 Release 1

The calling sequences for the subroutines in ESSL Version 5 Release 5 and ESSL Version 6 Release 1 are identical. Therefore, your application programs do not require any changes.

|<

Migrating Programs from ESSL for Linux on Power Version 5 Release 4 to Version 5 Release 5

The calling sequence for all subroutines except the CBLAS subroutines in ESSL Version 5 Release 4 and Version 5 Release 5 are identical, and therefore no changes to your application programs for these subroutines are required. However, if your programs contain calls to CBLAS functions using the arguments in Table 54 on page 179, changes are required. As shown in Table 54 on page 179, CBLAS enumerated types have changed to the following type definitions:

Table 54: CBLAS enumerated types changing to type definitions for ESSL Version 5 Release 5.

ESSL V5R4 enumerated type	**ESSL V5R5 definition type**
enum CBLAS_ORDER {CblasRowMajor=101, CblasColMajor=102};	typedef enum {CblasRowMajor=101, CblasColMajor=102} CBLAS_LAYOUT; typedef CBLAS_LAYOUT CBLAS_ORDER;[1]
enum CBLAS_TRANSPOSE {CblasNoTrans=111, CblasTrans=112, CblasConjTrans=113};	typedef enum {CblasNoTrans=111, CblasTrans=112, CblasConjTrans=113} CBLAS_TRANSPOSE;
enum CBLAS_UPLO {CblasUpper=121, CblasLower=122};	typedef enum {CblasUpper=121, CblasLower=122} CBLAS_UPLO;
enum CBLAS_DIAG {CblasNonUnit=131, CblasUnit=132};	typedef enum {CblasNonUnit=131, CblasUnit=132} CBLAS_DIAG;
enum CBLAS_SIDE {CblasLeft=141, CblasRight=142};	typedef enum {CblasLeft=141, CblasRight=142} CBLAS_SIDE;

[1] You may continue to use the name CLAS_ORDER, but it is recommended that you change to CBLAS_LAYOUT.

For example, in ESSL Version 5 Release 5 the following type of variable definition is no longer supported:

```
enum CBLAS_UPLO variable;
```

Instead, use something like the following:

```
CBLAS_UPLO variable;
```

Migrating Programs from ESSL for Linux on Power Version 5 Release 3.2 to Version 5 Release 4

The calling sequences for the subroutines in ESSL Version 5 Release 3.2 and ESSL Version 5 Release 4 are identical; therefore, no changes to your application programs are required.

Planning for Future Migration

With respect to planning for the future, if working storage does not need to persist after the subroutine call, you should use dynamic allocation. Otherwise, you should use the processor-independent formulas or simple formulas for calculating the values for the *naux* arguments in the ESSL calling sequences. Two things may occur that could cause the minimum values of *naux*, returned by ESSL error handling, to increase in the future:

- If changes are made to the ESSL subroutines to improve performance
- If changes are necessary to support future processors

The formulas allow you to specify your auxiliary storage large enough to accommodate any future improvements to ESSL and any future processors. If you do not provide, at least, these amounts of storage, your program may not run in the future.

You should use the following rule of thumb: To protect your application from having to be recoded in the future because of possible increased requirements for auxiliary storage, use dynamic allocation if possible. If the working storage must persists after the subroutine call, then you should provide as much storage as possible in your current application. In determining the right amount to specify, you should weigh your storage constraints against the inconvenience of making future changes, then specify what you think is best. If possible, you should provide this larger amount of storage to prevent future migration problems.

Migrating From One Hardware Platform to Another

This describes all the aspects of migrating your ESSL application programs from one hardware platform to another.

Auxiliary Storage

The minimum amount of auxiliary storage returned by ESSL error handling may vary from one hardware platform to another for the following subroutines:

- all the Fourier transform subroutines
- SCONF
- SCORF
- SACORF

Therefore, to guarantee that your application programs always migrate from any platform to any other platform, you should use the processor-independent formulas to determine the amount of auxiliary storage to use.

Bitwise-Identical Results

Because of hardware and ESSL design differences, the results you obtain when migrating from one ESSL service level to another, one ESSL library to another, or one hardware platform to another may not be bitwise-identical. The results, however, are mathematically equivalent.

Migrating from Other Libraries to ESSL

This describes some general aspects of moving from an IBM or non-IBM engineering and scientific library to ESSL.

Migrating from LAPACK

ESSL contains some subroutines that conform to the LAPACK interface. If you are using these subroutines, no coding changes are needed to migrate to ESSL.

Migrating from FFTW Version 3.1.2

ESSL includes header files and C and Fortran wrappers in source form for a subset of the FFTW Version 3.1.2 subroutines. If you want to use these wrappers, you must include the header file fftw3_essl.h instead of fftw3.h. For additional information on the FFTW Wrapper libraries, see Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1239.

Migrating from a Non-IBM Subroutine Library

If you are using a non-IBM library, ESSL may provide subroutines corresponding to those you are currently using. You may choose to migrate your program to benefit from the increased performance offered by the ESSL subroutines. In this case, you may have to recode your program to use the ESSL calling sequences, because the names and arguments used by ESSL may be different from those used by the non-IBM library. On the other hand, if you are using any of the standard Level 1, 2, and 3 BLAS or LAPACK routines that correspond to ESSL subroutines, you do not need to recode the calling sequences. The ESSL calling sequences are the same as the public domain code.

Chapter 7. Troubleshooting ESSL

This provides the following information for your use when dealing with errors.

Where to Find More Information About Errors

Specific errors associated with each ESSL subroutine are listed under "Error Conditions" in each subroutine description.

Getting Help from IBM Support

Should you require help from IBM in resolving an ESSL problem, report it and provide the following information, if available and appropriate.

1. Your customer number
2. The ESSL program number:

   ESSL for Linux

   >|5765-L61|<

   This is important information that speeds up the correct routing of your call.
3. The version and release of the operating system that you are running on. Enter the following command:

   ```
   uname -a
   ```

   This is important information that speeds up the correct routing of your call.
4. The names and versions of key products being run. Enter the following command:

   ```
   rpm -q packagewhere the appropriate values of package are listed in Table 55 on page 183.
   ```

Table 55: Product File Set and Package Names.

Descriptive Name	**Product Packages**
ESSL	essl.rte essl.3264.rte essl.3264.rtecuda essl.6464.rte
XL Fortran Runtime Environment	libxlf
SMP Runtime Environment	libxlsmp
XL Fortran compiler	>|xlf.15.1.6, or 16.1.0|<
XL C compiler	>|xlc.13.1.6, or 16.1.0|<
XL C++ compiler	>|xlc.13.1.6, or 16.1.0|<

5. The message that is returned when an error is detected.
6. Any error message relating to core dumps.

7. The compiler listings, including compiler options in effect, and any run-time listings produced
8. Program changes made in comparison with a previous successful run
9. A small test case demonstrating the problem using the minimum number of statements and variables, including input data

Consult your IBM Service representative for more assistance.

Globalization

For globalization, all ESSL subroutines display messages located in externalized message catalogs. English versions of the message catalogs are shipped with the product, but your site may be using its own translated message catalogs. The environment variable **NLSPATH** is used by the various ESSL subroutines to find the appropriate message catalog. **NLSPATH** specifies a list of directories to search for message catalogs. The directories are searched, in the order listed, to locate the message catalog. In resolving the path to the message catalog, **NLSPATH** is affected by the value of the environment variables **LC_MESSAGES** and **LANG**.

The ESSL message catalogs are in English, and are located in the following directories:

```
>|/opt/ibmmath/essl/6.1/msg/en_US/essl.cat|<
/usr/share/locale/en_US.UTF-8/essl.cat
/usr/share/locale/en_US/essl.cat
/usr/share/locale/en/essl.cat
/usr/share/locale/C/essl.cat
```

If your site is using its own translations of the message catalogs, consult your system administrator for the appropriate value of **NLSPATH** or **LANG**.

Dealing with Errors

At run time, you can encounter a number of different types of errors that are specifically related to the use of the ESSL subroutines:

- Program exceptions
- Input-argument errors (2001-2099) and (2200-2299)
- Computational errors (2100-2199)
- Resource errors (2401-2499)
- Informational and Attention messages (2600-2699)
- Miscellaneous errors (2700-2799)

Program Exceptions

The program exceptions you can encounter in ESSL are described in *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754–1985*.

ESSL Input-Argument Error Messages

If you receive an error message in the form 2538-20*nn* or 2538–22*nn*, you have an input-argument error in the calling sequence for an ESSL subroutine. Your program terminated at this point unless you did one of the following:

- Specified the ESSL user exit routine, ENOTRM, with ERRSET to determine the correct input argument values in your program for the optionally-recoverable ESSL errors 2015, 2030 or 2200. For details on how to do this, see Chapter 4, “Coding programs,” on page 125.
- Reset the number of allowable errors (2099) during ESSL installation or using ERRSET in your program. **This is not recommended for input-argument errors.**

Note: For many of the ESSL subroutines requiring auxiliary storage, you can avoid program termination due to error 2015 by allowing ESSL to dynamically allocate auxiliary storage for you. You do this by setting *naux* = 0 and making error 2015 unrecoverable. For details on which *aux* arguments allow dynamic allocation and how to specify them, see the subroutine descriptions.

The name of the ESSL subroutine detecting the error is listed as part of the message. The argument number(s) involved in the error appears in the message text. See "Input-Argument Error Messages(2001-2099)" on page 187 for a complete description of the information contained in each message and for an indication of which messages correspond to optionally-recoverable errors. Regardless of whether the name in the message is a user-callable ESSL subroutine or an internal ESSL routine, the message-text and its unique parts apply to the user-callable ESSL subroutine. Return code values are described under "Error Conditions" for each ESSL subroutine.

You may get more than one error message, because most of the arguments are checked by ESSL for possible errors during each call to the subroutine. The ESSL subroutine returns as many messages as there are errors detected. As a result, fewer runs are necessary to diagnose your program.

Fix the error(s), recompile, relink, and rerun your program.

ESSL Computational Error Messages

If you receive an error message in the form 2538-21*nn*, you have a computational error in the ESSL subroutine. A computational error is any error occurring in the ESSL subroutine while using the computational data (that is, scalar and array data). The name of the ESSL subroutine detecting the error is listed as part of the message. Regardless of whether the name in the message is a user-callable ESSL subroutine or an internal ESSL routine, the message-text and its unique parts apply to the user-callable ESSL subroutine. A nonzero return code is returned when the ESSL subroutine encounters a computational error. See "Computational Error Messages(2100-2199)" on page 191 for a complete description of the information in each message. Return code values are described under "Error Conditions" for each ESSL subroutine.

Your program terminates for some computational errors unless you have called ERRSET to reset the number of allowable errors for that particular error, and the number has not been exceeded. A message is issued for each computational error. You should use the message to determine where the error occurred in your program.

If you called ERRSET and you have not reached the limit of errors you had set, you can check the return code. If it is not 0, you should call the EINFO subroutine to obtain information about the data involved in the error. EINFO provides the same information provided in the messages; however, it is provided to your program so your program can check the information during run time. Depending on what you want to do, you may choose to continue processing or terminate your program after the error occurs. For information on how to make these changes in your program to reset the number of allowable errors, how to diagnose the error, and how to decide whether to continue or terminate your program, see Chapter 4, "Coding programs," on page 125.

If you are unable to solve the problem, report it and provide the following information, if available and appropriate:

- The message number and the module that detected an error
- The system dump, system error code, and system log of this job
- The compiler listings, including compiler options in effect, and any run-time listings produced
- Program changes made in comparison with a previous successful run
- A small test case demonstrating the problem using the minimum number of statements and variables, including input data
- A brief description of the problem

ESSL Resource Error Messages

If you receive a message in the form 2538-24*nn*, it means that ESSL issued a resource error message.

A resource error occurs when a buffer storage allocation request fails in a ESSL subroutine. In general, the ESSL subroutines allocate internal auxiliary storage dynamically as needed. Without sufficient storage, the subroutine cannot complete the computation.

When a buffer storage allocation request fails, a resource error message is issued, and the application program is terminated. You need to reduce the memory constraint on the system or increase the amount of memory available before rerunning the application program.

The following ways may reduce memory constraints:

- Investigate the load of your process and run in a more dedicated environment.
- Increase your processor's paging space.
- Select a machine with more memory.
- Check the setting of your user ID's user limit (`ulimit`).

ESSL Informational and Attention Messages

If you receive a message in the form 2538-26*nn*, it means that ESSL issued an informational or attention message.

Informational Messages

When you receive an informational message, check your application to determine why the condition was detected.

ESSL Attention Messages

An attention message is issued to describe a condition that occurred. ESSL is able to continue processing, but performance may be degraded.

One condition that may produce an attention message is when enough work area was available to continue processing, but was not the amount initially requested. ESSL does not terminate your application program, but performance may be degraded. If you want to reduce the memory constraint on the system or increase the amount of memory available to eliminate the attention message, see the suggestions in “ESSL Resource Error Messages” on page 185.

Miscellaneous Error Messages

If you receive a message in the form 2538-27*nn*, it means that ESSL issued a miscellaneous error message.

A miscellaneous error is an error that does not fall under any other categories.

When ESSL detects a miscellaneous error, you receive an error message with information on how to proceed and your application program is terminated.

Messages

This explains the conventions used for the ESSL messages and lists all the ESSL messages. For a description of each of the four types of ESSL messages, see “Dealing with Errors” on page 184.

Message Conventions

This describes the message conventions for the ESSL product.

About Upper- and Lowercase

Literals, such as, 'N', 'T', 'U', and so forth, appear in the messages in this documentation in uppercase; however, they may be specified in your ESSL calling sequence in either upper- or lowercase, for example, 'n', 't', and 'u'.

Message Format

The ESSL messages are issued in your output in the following format:

rtn-name : 2538-*mmnn*
message-text

Figure 10: Message Format

The parts of the ESSL message are as follows:

rtn-name
gives the name of the ESSL subroutine that encountered the error. If *rtn-name* is ESSL, this indicates that at least one ESSL subroutine encountered this error.

2538
is the ESSL component identification number.

mm
indicates the type of ESSL error message:

20—Input-argument error message
21—Computational error message
22—Input-argument error message
24—Resource error message
26—Information and attention message
27—Miscellaneous error message

nn
is the message identification number.

message-text
describes the nature of the error. Where one of several possible message-texts can be issued for a particular ESSL error, they are listed with an "or" between them. The possible unique parts are:

- The argument number of each argument involved in the error is included in the message description as (ARG NO. _)
- Additional information about the error is included in the message. The placement of this information is shown in the messages as (_)

Input-Argument Error Messages(2001-2099)

Note: There are more input-argument error messages listed in "Input-Argument Error Messages(2200-2299)" on page 193

2538-2001	**The number of elements (ARG NO. _) in a vector must be greater than or equal to zero.**
2538-2002	**The stride (ARG NO. _) for a vector must be nonzero.**
2538-2003	**The number of rows (ARG NO. _) in a matrix must be greater than or equal to zero.**
2538-2004	**The number of columns (ARG NO. _) in a matrix must be greater than or equal to zero.**
2538-2005	**The size of the leading dimension (ARG NO. _) of an array must be greater than zero.**
2538-2006	**The number of rows (ARG NO. _) of a matrix must be less than or equal to the size of the leading dimension (ARG NO. _) of its array.**
2538-2007	**The degree of a polynomial (ARG NO. _) must be greater than or equal to zero.**
2538-2008	**The number of elements (ARG NO. _) to be scanned must be greater than or equal to 2.**
2538-2009	**The number of elements (ARG NO. _) in a vector to be processed must be greater than or equal to 3.**
2538-2010	**The transform length (ARG NO. _) must be a power of 2.**

2538-2011	The number of points used in the interpolation (ARG NO. _) must be greater than or equal to zero and less than or equal to the number of data points (ARG NO. _).
2538-2012	The transform length (ARG NO. _) must be less than or equal to (_).
2538-2013	The transform length (ARG NO. _) must be greater than or equal to (_).
2538-2014	The routine must be initialized with the present value of (ARG NO. _).
2538-2015	The number of elements (ARG NO. _) in a work array must be greater than or equal to (_).
2538-2016	The form (ARG NO. _) of a matrix must be 'N' or 'T'. or The form (ARG NO. _) of a matrix must be 'N', 'T', or 'C'. or The form (ARG NO. _) of a matrix must be 'N' or 'C'.
2538-2017	The dimension (ARG NO. _) of the matrices must be greater than or equal to zero.
2538-2018	The matrix form is specified by (ARG NO. _); therefore, the leading dimension (ARG NO. _) of its array must be greater than or equal to the number of its rows (ARG NO. _).
2538-2019	The number of sequences (ARG NO. _) must be greater than zero.
2538-2020	(ARG NO. _) must be nonzero.
2538-2021	The storage control switch (ARG NO. _) must be 1, 2, 3, or 4.
2538-2022	(ARG NO. _) must be less than (_).
2538-2023	The outer loop increment (ARG NO. _) must be greater than or equal to zero.
2538-2024	The stride (ARG NO. _) for a vector must be greater than or equal to zero.
2538-2025	The stride (ARG NO. _) for a vector must be greater than zero.
2538-2026	The stride (ARG NO. _) for a vector must be greater than or equal to (_).
2538-2027	The order (ARG NO. _) of a matrix must be greater than or equal to zero.
2538-2028	The job option argument (ARG NO. _) must be [one of the following: 0, 1, or 2; 0, 1, 2, or 3; 0, 1, 2, 10, 11, or 12; 0, 1, 10, or 11; 0, 1, 20, or 21; 0, 1, 10, 11, 20, 21, 30, or 31; 0, 1, 2, 3, or 4].
2538-2029	The job option argument (ARG NO. _) must be 0 or 1.
2538-2030	The transform length (ARG NO. _) is not an allowed value. The next higher allowed value is (_).
2538-2031	The resulting convolution length obtained from ARG NO. 10 = (_), ARG NO. 11 = (_), ARG NO. 13 = (_), and ARG NO. 14 = (_) must be less than (_).
2538-2032	The size of the leading dimension (ARG NO. _) of the matrix must be greater than or equal to (_), the bandwidth constraint.
2538-2033	The lower bandwidth (ARG NO. _) must be greater than or equal to zero.
2538-2034	The upper bandwidth (ARG NO. _) must be greater than or equal to zero.
2538-2035	The half-band bandwidth (ARG NO. _) must be greater than or equal to zero.
2538-2036	The lower bandwidth (ARG NO. _) must be less than the order (ARG NO. _) of the matrix.
2538-2037	The upper bandwidth (ARG NO. _) must be less than the order (ARG NO. _) of the matrix.
2538-2038	The half-band bandwidth (ARG NO. _) must be less than the order (ARG NO. _) of the matrix.
2538-2039	(ARG NO. _) must be greater than zero.
2538-2040	Insufficient storage allocated for positive definite solve. (_) additional bytes required.
2538-2041	The resulting correlation length obtained from ARG NO. 8 = (_) and ARG NO. 10 = (_) must be less than (_).

2538-2042	**(ARG NO. _) must be greater than or equal to zero.**
2538-2043	**(ARG NO. _) must be greater than (_).**
2538-2044	**The number of initialized coefficients (ARG NO. _) cannot exceed the size of the coefficient vector (ARG NO. _).**
2538-2045	**The order specified (ARG NO. _) is not supported for this quadrature method. The nearest supported order is (_).**
2538-2046	**The scaling parameter (ARG NO. _) must be greater than zero for this quadrature method.**
2538-2047	**The scaling parameter (ARG NO. _) must be nonzero for this quadrature method.**
2538-2048	**The sum of (ARG NO. _) and (ARG NO. _) must be nonzero for this quadrature method.**
2538-2049	**The number of data points (ARG NO. _) must be greater than one in order to perform numerical quadrature.**
2538-2050	**The number of columns specified for the arrays to store the matrix in compressed matrix mode (ARG NO. _) must be greater than or equal to (_).**
2538-2051	**The number of columns (ARG NO. _) specified for the matrix used to store the sparse matrix in compressed mode must be greater than zero.**
2538-2052	**The total number of non-zero elements of the input sparse matrix stored by rows, obtained from element (_) of the row pointers array (ARG NO. _), must be greater than or equal to zero.**
2538-2053	**The number of non-zero elements in row (_) obtained from the row pointer array (ARG NO. _) is less than zero.**
2538-2054	**The number of diagonals (ARG NO. _) specified for the matrix used to store the sparse matrix in compressed diagonal mode must be greater than zero.**
2538-2055	**Element (_) of the vector used to store the diagonal numbers (ARG NO. _) is incompatible with the order of the sparse matrix (ARG NO. _).**
2538-2056	**The matrix is singular because the number of non-zero entries (ARG NO. _) is zero.**
2538-2057	**Element (_) in the integer parameter vector (ARG NO. _) must be greater than or equal to zero.**
2538-2058	**Element (_) in the integer parameter vector (ARG NO. _) must be (_),(_), or (_).**
2538-2059	**Element (_) in the real parameter vector (ARG NO. _) must be greater than zero.**
2538-2060	**The size of the leading dimension (ARG NO. _) of an array must be greater than or equal to the maximum of (ARG NO. _) and (ARG NO. _).**
2538-2061	**Parameter (ARG NO. _), which specifies the number of columns of the input sparse matrix (ARG NO. _ and ARG NO. _), must be greater than or equal to (_).**
2538-2062	**The number of random numbers generated (ARG NO. _) must be even and greater than or equal to zero.**
2538-2063	**SIDE (ARG NO. _), which specifies whether the triangular input matrix (ARG NO. _) appears on the left or right of the other input matrix, must be 'L' or 'R'.**
2538-2064	**UPLO (ARG NO. _), which specifies whether an input matrix (ARG NO. _) is upper or lower triangular, must be 'U' or 'L'.**
2538-2065	**DIAG (ARG NO. _), which specifies whether an input matrix (ARG NO. _) is unit triangular, must be 'U' or 'N'.**
2538-2066	**Given the value which has been assigned to SIDE (ARG NO. _), the leading dimension (ARG NO. _) for the triangular input matrix must be greater than or equal to (ARG NO. _).**

2538-2067	**TRANSA (ARG NO. _) specifies whether an input matrix (ARG NO. _), its transpose, or its conjugate transpose should be used. TRANSA must be 'N', 'T', or 'C'.**
2538-2068	**The size of the leading dimension (ARG NO. _) of an array must be greater than or equal to zero.**
2538-2070	**Element (_) in (ARG NO. _) must be [one of the following: 0 or 1; greater than zero; greater than or equal to zero; greater than or equal to zero and less than or equal to 1; greater than the preceding element; greater than or equal to 1 and less than or equal to n; -1 or 1; nonzero; 0, 1, 2, 10, or 11; 0, 1, 2, 10, 11, 100, 102, or 110; 0; 1; 0, 1, 2, 10, 11, 100, 101, 102, 110, or 111; 1, 2, 3, or 4; 1, 2, 3, 4, or 5].**
2538-2071	**The number of eigenvalues (ARG NO. _) must be less than or equal to the order of the matrix (ARG NO. _).**
2538-2072	**The work area (ARG NO. _) does not contain a valid vector seed. The routine must be called with a nonzero value of ISEED (ARG NO. _).**
2538-2073	**(ARG NO. _) must be a double precision whole number greater than or equal to 1.0 and less than 2147483647.0.**
2538-2074	**Performance can be improved by using a larger work array. For best performance, specify the number of elements (ARG NO. _) in the work array to be greater than or equal to (_).**
2538-2075	**The data type parameter (ARG NO. _) must be 'S', 'D', 'C', or 'Z'.**
2538-2076	**(ARG NO. _) must be greater than or equal to (_) and smaller than (_).**
2538-2077	**The matrix is singular. Column (_) is empty in the matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _).**
2538-2078	**The matrix is singular. Row (_) is empty in the matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _).**
2538-2079	**The matrix, specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _), contains at least one duplicate column index in row (_).**
2538-2080	**Element (_) in (ARG NO. _) must be [one of the following: greater than or equal to (_) and less than or equal to (_); greater than or equal to (_) and less than or equal to (ARG NO. _); greater than or equal to element (_) and less than or equal to (_); zero or must be greater than or equal to (_)].**
2538-2081	**Element (_) in (ARG NO. _) must be less than or equal to (_).**
2538-2082	**Element (_) in (ARG NO. _) may cause incorrect or misleading results. [One of the following: A nonzero number with absolute value less than or equal to 1; a positive number less than or equal to 1] is recommended.**
2538-2083	**The pivot tolerance (element (_) in (ARG NO. _)) may cause incorrect or misleading results. A number greater than or equal to 0 and less than or equal to 1 is recommended.**
2538-2084	**The dimension (ARG NO. _) of the array (ARG NO. _) must be greater than or equal to (_).**
2538-2085	**The number of steps after which the generalized minimum residual method is restarted, element (_) in (ARG NO. _), must be greater than 0.**
2538-2086	**The acceleration parameter, element (_) in (ARG NO. _), must be greater than 0 when using the SSOR preconditioner.**
2538-2087	**STOR (ARG NO. _), which specifies the storage variation used to represent the input sparse matrix, must be 'G', 'L', or 'U'.**
2538-2088	**INIT (ARG NO. _), which specifies the type of computation to be performed, must be 'I' or 'S'.**
2538-2089	**Element (_) in (ARG NO. _) must be [one of the following: greater than**

	or equal to (_); greater than or equal to element (_)].
2538-2090	**For level (_), the number of grid points for dimension (_) must be an odd number greater than 1.**
2538-2091	**Since the mesh spacing (ARG NO. _) here is not constant, the second order prolongation method must be used. That is, element (_) of (ARG NO. _) must be (_).**
2538-2092	**The index into (ARG NO. _) is out of range. This index is element (_,_) of (ARG NO. _).**
2538-2093	**The index into (ARG NO. _) is out of range. This index is element (_,_,_) of (ARG NO. _).**
2538-2094	**For dimension (_) on level (_), the mesh spacing must be changed to a positive value.**
2538-2095	**Excess space in (ARG NO. _) has been decreased and may be inadequate. To avoid this, specify the coarse level matrix as the final item in this argument.**
2538-2096	**For level (_), the matrix type, solver, and preconditioner are incompatible.**
2538-2097	**The solver requested for level (_) requires a square matrix. Elements (_,_,_) and (_,_,_) in (ARG NO. _) must be equal.**
2538-2098	**Element (_,_) of (ARG NO. _) must be greater than or equal to (_).**
2538-2099	**End of input argument error reporting. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.**

Computational Error Messages(2100-2199)

2538-2100	**The computed index of a vector is out of the range (_) to (_).**
2538-2101	**Eigenvalue (_) failed to converge after (_) iterations.**
2538-2102	**Eigenvector (_) failed to converge after (_) iterations.**
2538-2103	**The matrix (ARG NO. _) is singular. Zero diagonal element (_) has been detected.**
2538-2104	**The matrix (ARG NO. _) is not positive definite. The last diagonal element with nonpositive value is (_).**
2538-2105	**Factorization failed due to near zero pivot number (_).**
2538-2106	**Vector boundary misalignment detected in ESSL scalar library.**
2538-2107	**Singular value (_) failed to converge after (_) iterations.**
2538-2108	**The matrix specified by (ARG NO. _) and (ARG NO. _) is not definite because the diagonal is not of constant sign.**
2538-2109	**The matrix specified by (ARG NO. _) and (ARG NO. _) is not definite and the iterative process is stopped at iteration number (_).**
2538-2110	**The maximum allowed number of iterations, element number (_) of (ARG NO. _), were performed but the iterative process did not converge to a solution according to the stopping procedure.**
2538-2111	**The factorization matrix (ARG NO. _) is not consistent with the sparse matrix specified by (ARG NO. _) and (ARG NO. _).**
2538-2112	**The incomplete factorization of the sparse matrix specified by (ARG NO. _) and (ARG NO. _) is not stable.**
2538-2113	**Unexpected nonzero vector mask detected in ESSL scalar routine. Contact your IBM Service Representative.**
2538-2114	**Eigenvalue (_) failed to converge after (_) iterations.**
2538-2115	**The matrix (ARG NO. _) is not positive definite. The leading minor of order (_) has a nonpositive determinant.**
2538-2116	**The matrix specified by (ARG NO. _) and (ARG NO. _) is singular.**
2538-2117	**The pivot element in column (_) is smaller than the first element in (ARG NO. _).**
2538-2118	**The pivot element in row (_) is smaller than the first element in (ARG NO. _).**

2538-2119	The storage space, specified by (ARG NO. _), is insufficient.
2538-2120	The matrix is singular. The last row processed in the matrix was row (_).
2538-2121	The matrix is singular. the last column processed was column (_).
2538-2122	The factorization failed. No pivot element was found in the active submatrix.
2538-2123	Performance can be improved by specifying a larger value for (ARG NO. _). (_) compressions were performed.
2538-2124	The data contained in AUX1, (ARG NO. _), was computed for a different algorithm.
2538-2126	The pivot value at row (_) is not acceptable based on pivot criteria ((ARG NO. _) and (ARG NO. _)). No fixup was applicable to this pivot. The matrix (ARG NO. _) may be singular or not definite.
2538-2127	The pivot value at row (_) was replaced with element (_) in (ARG NO. _). The matrix (ARG NO. _) may be singular or not definite.
2538-2128	Internal ESSL error. contact your IBM service representative.
2538-2129	The matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _) is not definite because the diagonal is not of constant sign or some diagonal element is zero.
2538-2130	The incomplete factorization of the sparse matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _) is not stable.
2538-2131	The matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _) is singular.
2538-2132	Element (_) in (ARG NO. _) indicates that factorization was done on a previous call. The data passed is not the result of a prior valid factorization.
2538-2133	An error occurred on level (_) in the user-supplied subroutine specified by (ARG NO. _).
2538-2134	The data contained in (ARG NO. _) is not consistent with the sparse matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _).
2538-2135	For level (_), loss of orthogonality occurred in a minimum residual solver because the input matrix (element (_,_) of (ARG NO. _)) is inappropriate. Choose one of the other non-symmetric solvers.
2538-2136	For level (_), the main diagonal element for row (_) of a matrix is 0.
2538-2145	The input matrix (ARG NO. _) is singular. The first diagonal element found to be exactly zero was in column (_).
2538-2146	The input matrix (ARG NO. _) is singular. The first diagonal element found to be exactly zero was in column (_).
2538-2147	The matrix (ARG NO. _) is singular. Zero diagonal element (_) has been detected.
2538-2148	The matrix (ARG NO. _) is not positive definite. The leading minor of order (_) has a nonpositive determinant.
2538-2149	Factorization failed due to near zero pivot number (_).
2538-2150	The inverse of matrix (ARG NO. _) could not be computed. The first diagonal element of the factored matrix found to be exactly zero was in column (_).
2538-2151	The inverse of matrix (ARG NO. _) could not be computed. The first diagonal element of the factored matrix found to be exactly zero was in column (_).
2538-2152	The alignment of (ARG NO. _) changed after initialization. Performance may be significantly degraded.
2538-2153	Eigenvalue (_) failed to converge. Arrays WR (ARG NO. _) and WI (ARG NO. _) contain the eigenvalues successfully computed. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.

2538-2154	Bisection failed to converge for some eigenvalues. The eigenvalues may not be as accurate as the absolute and relative tolerances.
2538-2155	The number of eigenvalues computed (ARG NO. _) does not match the number of eigenvalues requested.
2538-2156	No eigenvalues were computed since the Gershgorin interval initially used was incorrect.
2538-2157	(_) eigenvectors failed to converge after (_) iterations. The indices are stored in IFAIL (ARG NO. _).
2538-2158	Eigenvalue (_) failed to converge. Array W (ARG NO. _) contains the eigenvalues successfully computed. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.
2538-2159	Eigenvalue (_) failed to converge in the QZ iteration. Arrays ALPHAR (ARG NO. _), ALPHAI (ARG NO. _) and BETA (ARG NO. _) contain the eigenvalues successfully computed. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.
2538-2160	Eigenvalue (_) failed to converge in the computation of shifts. Arrays ALPHAR (ARG NO. _), ALPHAI (ARG NO. _) and BETA (ARG NO. _) contain the eigenvalues successfully computed. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.
2538-2161	An eigenvector failed to converge because the 2-by-2 block (_ : _) did not have a complex eigenvalue.
2538-2162	The algorithm failed to converge because (_) off-diagonal elements of an intermediate tridiagonal form did not converge to zero.
2538-2163	An eigenvalue failed to converge in the submatrix starting at row and column (_) and ending at row and column (_).
2538-2164	Eigenvalue (_) failed to converge in the QZ iteration. Arrays ALPHA (ARG NO. _) and BETA (ARG NO. _) contain the eigenvalues successfully computed. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.
2538-2165	Eigenvalue (_) failed to converge in the computation of shifts. Arrays ALPHA (ARG NO. _) and BETA (ARG NO. _) contain the eigenvalues successfully computed. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.
2538-2166	The matrix specified by (ARG NO. _) and (ARG NO. _) is not positive definite. The leading minor of order (_) has a nonpositive determinant.
2538-2167	(_) superdiagonals of an intermediate bidiagonal form B did not converge to zero. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.
2538-2168	The matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _) is singular. The first diagonal element found to be exactly zero was in column (_).
2538-2169	Singular value decomposition failed to converge.
2538-2199	End of computational error reporting. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.

Input-Argument Error Messages(2200-2299)

2538-2200	The dimension (ARG NO. _) of the array (ARG NO. _) must be greater than or equal to (_).
2538-2201	The number of elements (ARG NO. _) in a work array (ARG NO. _) must be zero, to indicate dynamic allocation, minus one, to indicate workspace query, or greater than

	or equal to (_) if a work array is being supplied.
2538-2207	The number of elements in the array (ARG NO. _) must be less than or equal to (_).
2538-2208	ANORM (ARG NO. _) must be equal to zero or greater than or equal to (_) and less than or equal to (_).
2538-2209	NORM (ARG NO. _), which specifies the computation to be performed, must be 'M', '1', 'O', 'I', 'F', or 'E'.
2538-2210	NORM (ARG NO. _), which specifies whether to calculate the 1-norm condition number or the infinity-norm condition number, must be '1', 'O', or 'I'.
2538-2211	The alignment of (ARG NO. _) changed after initialization.
2538-2212	JOBZ (ARG NO. _), which specifies whether or not to compute eigenvectors, must be 'N' or 'V'.
2538-2213	RANGE (ARG NO. _), which specifies which eigenvalues to find, must be 'A', 'V', or 'I'.
2538-2214	VU (ARG NO. _), which specifies the upper bound of the interval to be searched for eigenvalues, must be greater than VL (ARG NO. _), which specifies the lower bound of the interval to be searched for eigenvalues.
2538-2215	IL (ARG NO. _), which specifies the index of the smallest eigenvalue to be returned, must be greater than or equal to 1 and less than or equal to the larger of 1 and the order (ARG NO. _) of the matrix (ARG NO. _).
2538-2216	IU (ARG NO. _), which specifies the index of the largest eigenvalue to be returned, must be greater than or equal to the smaller of the order (ARG NO. _) of the matrix (ARG NO. _) and IL (ARG NO. _) and less than or equal to the order of the matrix.
2538-2217	BALANC (ARG NO. _), which specifies whether or not to diagonally scale the input matrix (ARG NO. _) and whether or not to permute the input matrix, must be 'N', 'P', 'S', or 'B'.
2538-2218	JOBVL (ARG NO. _), which specifies whether or not to compute left eigenvectors, must be 'N' or 'V'.
2538-2219	JOBVR (ARG NO. _), which specifies whether or not to compute right eigenvectors, must be 'N' or 'V'.
2538-2220	SENSE (ARG NO. _), which specifies which reciprocal condition numbers are to be computed, must be 'N', 'E', 'V', or 'B'.
2538-2221	JOBVL (ARG NO. _) and JOBVR (ARG NO. _) must be 'V' if SENSE (ARG. NO _) is 'E' or 'B'.
2538-2222	ITYPE (ARG NO. _), which specifies the problem type, must be 1, 2, or 3.
2538-2223	The routine must be initialized with the present value of element (_) of (ARG NO. _).
2538-2224	UPLO (ARG NO. _), which specifies whether off-diagonal E (ARG NO. _) is the superdiagonal or the subdiagonal of the bidiagonal factorization, must be 'U' or 'L'.
2538-2225	The lower bandwidth (ARG NO. _) must be less than the number of rows (ARG NO. _) of the matrix.
2538-2226	The upper bandwidth (ARG NO. _) must be less than the number of columns (ARG NO. _) of the matrix.
2538-2227	JOBU (ARG NO. _), which specifies whether or not to compute left singular vectors, must be 'N', 'A', 'S', or 'O'.
2538-2228	JOBVT (ARG NO. _), which specifies whether or not to compute left singular vectors, must be 'N', 'A', 'S', or 'O'.
2538-2229	JOBU (ARG NO. _) and JOBVT (ARG NO. _) cannot both be 'O'.
2538-2230	The size of the leading dimension (ARG NO. _) of an array must be greater than or equal to the smaller of (ARG NO. _) and (ARG NO. _).
2538-2231	IOPT (ARG NO. _) must be 1 or 2.

2538-2232	**IREPEAT (ARG NO. _) must be 0 or 1.**
2538-2233	**LISEED (ARG NO. _), which depends on IOPT (ARG NO. _), must be greater than or equal to (_).**
2538-2234	**LISTATE (ARG NO. _), which depends on IOPT (ARG NO. _), must be minus one to indicate an ISTATE (ARG NO. _) size query, or greater than or equal to (_) if the state vector has been supplied.**
2538-2235	**ISTATE (ARG NO. _) is not initialized.**
2538-2236	**(ARG NO. _) must be less than (ARG NO. _).**
2538-2237	**ISTATE (ARG NO. _) must be initialized with IOPT equal to (_).**
2538-2238	**ESSL_CUDA_HYBRID must be "yes", "no", or unset.**
2538-2239	**ESSL_CUDA_PIN must be "yes", "no", "pinned", or unset.**
2538-2240	**Element (_) of array IDS (ARG NO. _) must be greater than or equal to zero or less than the number of CUDA devices (_).**
2538-2241	**This subroutine may be called only once, and it must be called before any ESSL GPU enabled subroutines.**
2538-2242	**The CUDA device corresponding to element (_) in array IDS (ARG NO. _) must be in NVIDIA compute mode 0 (DEFAULT) or 3 (EXCLUSIVE_PROCESS).**
2538-2243	**cblas_layout (ARG NO. _), which specifies whether matrices are stored in row major or column major order, must be CblasRowMajor or CblasColMajor.**
2538-2244	**The form (ARG NO. _) of a matrix must be CblasNoTrans or CblasTrans.**
2538-2245	**The form (ARG NO. _) of a matrix must be CblasNoTrans, CblasTrans, or CblasConjTrans.**
2538-2246	**The form (ARG NO. _) of a matrix must be CblasNoTrans or CblasConjTrans.**
2538-2247	**cblas_diag (ARG NO. _), which specifies whether an input matrix (ARG NO. _) is unit triangular, must be CblasUnit or CblasNonUnit.**
2538-2248	**cblas_side (ARG NO. _), which specifies whether the triangular input matrix (ARG NO. _) appears on the left or right of the other input matrix, must be CblasLeft or CblasRight.**
2538-2249	**cblas_uplo (ARG NO. _), which specifies whether an input matrix (ARG NO. _) is upper or lower triangular, must be CblasUpper or CblasLower.**
2538-2250	**matrix_layout (ARG NO. _), which specifies whether matrices are stored in row major or column major order, must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.**
2538-2251	**JOBZ (ARG NO. _), which specifies options for computing all or part of the matrix U, must be 'N', 'A', 'S', or 'O'.**
2538-2252	**The size of a work array (ARG NO. _), specified by (ARG NO. _) or implicitly computed by ESSL, must be greater than zero and less than or equal to 2147483647 when 32-bit integers are used.**
2538-2253	**BALANC (ARG NO. _), which specifies whether or not to diagonally scale the input matrices (ARG NO. _ and ARG NO. _) and whether or not to permute the input matrices, must be 'N', 'P', 'S', or 'B'.**
2538-2254	**The size of the leading dimension (ARG NO. _) of an array must be greater than or equal to (_), computed as (ARG NO. _) minus (ARG NO. _) plus 1.**
2538-2255	**The number of columns (ARG NO. _) of a matrix must be less than or equal to the size of the leading dimension (ARG NO. _) of its array.**

Resource Error Messages(2400-2499)

2538-2400	An internal buffer allocation has failed due to insufficient memory.
2538-2401	(_) failed with CUDA message: (_)
2538-2402	All CUDA visible devices, except those in mode 2 (PROHIBITED), must be in the same compute mode.

Informational and Attention Error Messages(2600-2699)

2538-2600	Performance may be degraded due to limited buffer space availability.
2538-2601	Execution terminating due to error count for error number (_) Message summary: Message number - Count
2538-2602	User error corrective routine entered. User corrective action taken. Execution continuing.
2538-2603	Standard corrective action taken. Execution continuing.
2538-2604	Execution terminating due to error count for error number _.
2538-2605	Message summary: _ - _
2538-2606	Serial execution is taking place since the input array is equal to the output array and either: INC2X (ARG NO. _) is not equal to 2 times INC2Y (ARG NO. _) or INC3X (ARG NO. _) is not equal to 2 times INC3Y (ARG NO. _).
2538-2607	Serial execution is taking place since the input array is equal to the output array and either: INC2X (ARG NO. _) is not equal to INC2Y (ARG NO. _) or INC3X (ARG NO. _) is not equal to INC3Y (ARG NO. _).
2538-2608	Performance may be improved by using a larger work array. For best performance, specify the number of elements (ARG NO. _) in the work array to be greater than or equal to (_).
2538-2609	Performance may be improved by specifying a larger value for (ARG NO. _). (_) compressions were performed.
2538-2610	Performance may be degraded due to the alignment of (ARG NO. _).
2538-2611	Performance may be improved by specifying an even value for (ARG NO. _).
2538-2612	Performance may be improved by specifying a multiple of four for (ARG NO. _).
2538-2613	ESSL computed the eigenvalues using multiple algorithms. Performance may be degraded.
2538-2614	Performance may be degraded because the number of available GPUs is zero.
2538-2615	Performance may be improved by specifying the number of threads (_) greater than or equal to the number of available GPUs (_).
2538-2616	_returned with CUDA message: _

Miscellaneous Error Messages(2700-2799)

2538-2700	Internal ESSL error number (_). Contact your IBM service representative.
2538-2701	ESSL chose to use GPUs during the previous initialization call, but GPUs may not be used by the current computational call because it was made from an OpenMP parallel region.
2538-2703	Internal ESSL error: message number requested (_) is outside of the valid range. Contact your IBM service representative.
2538-2799	Unable to locate message number (_). Please refer to 'Using Error Handling' in the ESSL Guide and Reference for the full message text.

Chapter 8. Linear algebra subprograms

The linear algebra subprograms, provided in four areas, are described here.

Overview of the Linear Algebra Subprograms

This describes the subprograms in each of the four linear algebra subprogram areas:

- Vector-scalar linear algebra subprograms ("Vector-Scalar Linear Algebra Subprograms" on page 197)
- Sparse vector-scalar linear algebra subprograms ("Sparse Vector-Scalar Linear Algebra Subprograms" on page 200)
- Matrix-vector linear algebra subprograms ("Matrix-Vector Linear Algebra Subprograms" on page 200)
- Sparse matrix-vector linear algebra subprograms ("Sparse Matrix-Vector Linear Algebra Subprograms" on page 202)

Note:

1. The term **subprograms** is used to be consistent with the Basic Linear Algebra Subprograms (BLAS), because many of these subprograms correspond to the BLAS.
2. Some of the linear algebra subprograms were designed in accordance with the Level 1 and Level 2 BLAS de facto standard. If these subprograms do not comply with the standard as approved, IBM will consider updating them to do so. If IBM updates these subprograms, the updates could require modifications of the calling application program.

Vector-Scalar Linear Algebra Subprograms

The vector-scalar linear algebra subprograms include a subset of the standard set of Level 1 BLAS. For details on the BLAS, see reference ["95" on page 1258]. The remainder of the vector-scalar linear algebra subprograms are commonly used computations provided for your applications. Both real and complex versions of the subprograms are provided.

Table 56: List of Vector-Scalar Linear Algebra Subprograms.

Short-Precision Subprogram	Long-Precision Subprogram	Descriptive Name and Location
ISAMAX†• ICAMAX†• cblas_isamax• cblas_icamax•	IDAMAX†• IZAMAX†• cblas_idamax• cblas_izamax•	"ISAMAX, IDAMAX, ICAMAX, and IZAMAX (Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude)" on page 204
ISAMIN† cblas_isamin	IDAMIN† cblas_idamin	"ISAMIN and IDAMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Absolute Value)" on page 206
ISMAX† cblas_ismax	IDMAX† cblas_idmax	"ISMAX and IDMAX (Position of the First or Last Occurrence of the Vector Element Having the Maximum Value)" on page 209
ISMIN† cblas_ismin	IDMIN† cblas_idmin	"ISMIN and IDMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Value)" on page 211

Table 56: List of Vector-Scalar Linear Algebra Subprograms. (continued)

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
SASUM†• SCASUM†• cblas_sasum• cblas_scasum•	DASUM†• DZASUM†• cblas_dasum• cblas_dzasum•	“SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector)” on page 214
>| SAXPBY• cblas_saxpby•	DAXPBY• cblas_daxpby•	“SAXPBY and DAXPBY (Scales Vector X and Vector Y by two Scalars, Adds them to one another, and Stores in the Vector Y)” on page 219|<
SAXPY• CAXPY• cblas_saxpy• cblas_caxpy•	DAXPY• ZAXPY• cblas_daxpy• cblas_zaxpy•	“SAXPY, DAXPY, CAXPY, and ZAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y)” on page 216
SCOPY• CCOPY• cblas_scopy• cblas_ccopy•	DCOPY• ZCOPY• cblas_dcopy• cblas_zcopy•	“SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)” on page 223
SDOT†• CDOTU†• CDOTC†• cblas_sdot• cblas_cdotu_sub• cblas_cdotc_sub•	DDOT†• ZDOTU†• ZDOTC†• cblas_ddot• cblas_zdotu_sub• cblas_zdotc_sub•	“SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDOTC (Dot Product of Two Vectors)” on page 225
SNAXPY	DNAXPY	“SNAXPY and DNAXPY (Compute SAXPY or DAXPY N Times)” on page 229
SNDOT	DNDOT	“SNDOT and DNDOT (Compute Special Dot Products N Times)” on page 233
SNRM2†• SCNRM2†• cblas_snrm2• cblas_scnrm2•	DNRM2†• DZNRM2†• cblas_dnrm2• cblas_dznrm2•	“SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow)” on page 238
SNORM2† CNORM2†	DNORM2† ZNORM2†	“SNORM2, DNORM2, CNORM2, and ZNORM2 (Euclidean Length of a Vector with No Scaling of Input)” on page 241
SROTG• CROTG• cblas_srotg• cblas_crotg•	DROTG• ZROTG• cblas_drotg• cblas_zrotg•	“SROTG, DROTG, CROTG, and ZROTG (Construct a Givens Plane Rotation)” on page 244

Table 56: List of Vector-Scalar Linear Algebra Subprograms. (continued)

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
SROT• CROT• CSROT• cblas_srot• cblas_crot• cblas_csrot•	DROT• ZROT• ZDROT• cblas_drot• cblas_zrot• cblas_zdrot•	"SROT, DROT, CROT, ZROT, CSROT, and ZDROT (Apply a Plane Rotation)" on page 250
SROTMG• cblas_srotmg•	DROTMG• cblas_drotmg•	"SROTMG and DROTMG (Construct a modified Givens Transformation)" on page 254
SROTM• cblas_srotm•	DROTM• cblas_drotm•	"SROTM and DROTM (Apply a modified Givens Transformation)" on page 256
SSCAL• CSCAL• CSSCAL• cblas_sscal• cblas_cscal• cblas_csscal•	DSCAL• ZSCAL• ZDSCAL• cblas_dscal• cblas_zscal• cblas_zdscal•	"SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, and ZDSCAL (Multiply a Vector X by a Scalar and Store in the Vector X)" on page 259
SSWAP• CSWAP• cblas_sswap• cblas_cswap•	DSWAP• ZSWAP• cblas_dswap• cblas_zswap•	"SSWAP, DSWAP, CSWAP, and ZSWAP (Interchange the Elements of Two Vectors)" on page 262
SVEA CVEA cblas_svea cblas_cvea	DVEA ZVEA cblas_dvea cblas_zvea	"SVEA, DVEA, CVEA, and ZVEA (Add a Vector X to a Vector Y and Store in a Vector Z)" on page 265
SVES CVES cblas_sves cblas_cves	DVES ZVES cblas_dves cblas_zves	"SVES, DVES, CVES, and ZVES (Subtract a Vector Y from a Vector X and Store in a Vector Z)" on page 269
SVEM CVEM cblas_svem cblas_cvem	DVEM ZVEM cblas_dvem cblas_zvem	"SVEM, DVEM, CVEM, and ZVEM (Multiply a Vector X by a Vector Y and Store in a Vector Z)" on page 273
SYAX CYAX CSYAX cblas_syax cblas_cyax cblas_csyax	DYAX ZYAX ZDYAX cblas_dyax cblas_zyax cblas_zdyax	"SYAX, DYAX, CYAX, ZYAX, CSYAX, and ZDYAX (Multiply a Vector X by a Scalar and Store in a Vector Y)" on page 276

Table 56: List of Vector-Scalar Linear Algebra Subprograms. (continued)

Short-Precision Subprogram	Long-Precision Subprogram	Descriptive Name and Location
SZAXPY CZAXPY cblas_szaxpy cblas_czaxpy	DZAXPY ZZAXPY cblas_dzaxpy cblas_zzaxpy	"SZAXPY, DZAXPY, CZAXPY, and ZZAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in a Vector Z)" on page 280
† This subprogram is invoked as a function in a Fortran program. · Level 1 BLAS		

Sparse Vector-Scalar Linear Algebra Subprograms

The sparse vector-scalar linear algebra subprograms operate on sparse vectors using optimized storage techniques; that is, only the nonzero elements of the vector are stored. These subprograms provide similar functions to the vector-scalar subprograms. These subprograms represent a subset of the sparse extensions to the Level 1 BLAS described in reference ["39" on page 1255]. Both real and complex versions of the subprograms are provided.

Table 57: List of Sparse Vector-Scalar Linear Algebra Subprograms.

Short-Precision Subprogram	Long-Precision Subprogram	Descriptive Name and Location
SSCTR CSCTR	DSCTR ZSCTR	"SSCTR, DSCTR, CSCTR, ZSCTR (Scatter the Elements of a Sparse Vector X in Compressed-Vector Storage Mode into Specified Elements of a Sparse Vector Y in Full-Vector Storage Mode)" on page 283
SGTHR CGTHR	DGTHR ZGTHR	"SGTHR, DGTHR, CGTHR, and ZGTHR (Gather Specified Elements of a Sparse Vector Y in Full-Vector Storage Mode into a Sparse Vector X in Compressed-Vector Storage Mode)" on page 285
SGTHRZ CGTHRZ	DGTHRZ ZGTHRZ	"SGTHRZ, DGTHRZ, CGTHRZ, and ZGTHRZ (Gather Specified Elements of a Sparse Vector Y in Full-Vector Mode into a Sparse Vector X in Compressed-Vector Mode, and Zero the Same Specified Elements of Y)" on page 288
SAXPYI CAXPYI	DAXPYI ZAXPYI	"SAXPYI, DAXPYI, CAXPYI, and ZAXPYI (Multiply a Sparse Vector X in Compressed-Vector Storage Mode by a Scalar, Add to a Sparse Vector Y in Full-Vector Storage Mode, and Store in the Vector Y)" on page 290
SDOTI† CDOTCI† CDOTUI†	DDOTI† ZDOTCI† ZDOTUI†	"SDOTI, DDOTI, CDOTUI, ZDOTUI, CDOTCI, and ZDOTCI (Dot Product of a Sparse Vector X in Compressed-Vector Storage Mode and a Sparse Vector Y in Full-Vector Storage Mode)" on page 292
† This subprogram is invoked as a function in a Fortran program.		

Matrix-Vector Linear Algebra Subprograms

The matrix-vector linear algebra subprograms operate on a higher-level data structure - matrix-vector rather than vector-scalar - using optimized algorithms to improve performance. These subprograms include a subset of the standard set of Level 2 BLAS. For details on the Level 2 BLAS, see ["44" on page 1255] and ["45" on page 1255]. Both real and complex versions of the subprograms are provided.

Table 58: List of Matrix-Vector Linear Algebra Subprograms.

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
SGEMV◄ CGEMV◄ SGEMX§ SGEMTX§ cblas_sgemv◄ cblas_cgemv◄	DGEMV◄ ZGEMV◄ DGEMX§ DGEMTX§ cblas_dgemv◄ cblas_zgemv◄	"SGEMV, DGEMV, CGEMV, ZGEMV, SGEMX, DGEMX, SGEMTX, and DGEMTX (Matrix-Vector Product for a General Matrix, Its Transpose, or Its Conjugate Transpose)" on page 295
SGER◄ CGERU◄ CGERC◄ cblas_sger◄ cblas_cgeru◄ cblas_cgerc◄	DGER◄ ZGERU◄ ZGERC◄ cblas_dger◄ cblas_zgeru◄ cblas_zgerc◄	"SGER, DGER, CGERU, ZGERU, CGERC, and ZGERC (Rank-One Update of a General Matrix)" on page 306
SSPMV◄ CHPMV◄ SSYMV◄ CHEMV◄ SSLMX§ cblas_sspmv◄ cblas_chpmv◄ cblas_ssymv◄ cblas_chemv◄	DSPMV◄ ZHPMV◄ DSYMV◄ ZHEMV◄ DSLMX§ cblas_dspmv◄ cblas_zhpmv◄ cblas_dsymv◄ cblas_zhemv◄	"SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)" on page 313
SSPR◄ CHPR◄ SSYR◄ CHER◄ SSLR1§ cblas_sspr◄ cblas_chpr◄ cblas_ssyr◄ cblas_cher◄	DSPR◄ ZHPR◄ DSYR◄ ZHER◄ DSLR1§ cblas_dspr◄ cblas_zhpr◄ cblas_dsyr◄ cblas_zher◄	"SSPR, DSPR, CHPR, ZHPR, SSYR, DSYR, CHER, ZHER, SSLR1, and DSLR1 (Rank-One Update of a Real Symmetric or Complex Hermitian Matrix)" on page 321
SSPR2◄ CHPR2◄ SSYR2◄ CHER2◄ SSLR2§ cblas_sspr2◄ cblas_chpr2◄ cblas_ssyr2◄ cblas_cher2◄	DSPR2◄ ZHPR2◄ DSYR2◄ ZHER2◄ DSLR2§ cblas_dspr2◄ cblas_zhpr2◄ cblas_dsyr2◄ cblas_zher2◄	"SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, ZHER2, SSLR2, and DSLR2 (Rank-Two Update of a Real Symmetric or Complex Hermitian Matrix) " on page 328
SGBMV◄ CGBMV◄ cblas_sgbmv◄ cblas_cgbmv◄	DGBMV◄ ZGBMV◄ cblas_dgbmv◄ cblas_zgbmv◄	"SGBMV, DGBMV, CGBMV, and ZGBMV (Matrix-Vector Product for a General Band Matrix, Its Transpose, or Its Conjugate Transpose)" on page 336

Table 58: List of Matrix-Vector Linear Algebra Subprograms. (continued)

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
SSBMV◂ CHBMV◂ cblas_ssbmv◂ cblas_chbmv◂	DSBMV◂ ZHBMV◂ cblas_dsbmv◂ cblas_zhbmv◂	"SSBMV, DSBMV, CHBMV, and ZHBMV (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Band Matrix)" on page 343
STRMV◂ CTRMV◂ STPMV◂ CTPMV◂ cblas_strmv◂ cblas_ctrmv◂ cblas_stpmv◂ cblas_ctpmv◂	DTRMV◂ ZTRMV◂ DTPMV◂ ZTPMV◂ cblas_dtrmv◂ cblas_ztrmv◂ cblas_dtpmv◂ cblas_ztpmv◂	"STRMV, DTRMV, CTRMV, ZTRMV, STPMV, DTPMV, CTPMV, and ZTPMV (Matrix-Vector Product for a Triangular Matrix, Its Transpose, or Its Conjugate Transpose)" on page 347
STRSV◂ CTRSV◂ STPSV◂ CTPSV◂ cblas_strsv◂ cblas_ctrsv◂ cblas_stpsv◂ cblas_ctps◂	DTRSV◂ ZTRSV◂ DTPSV◂ ZTPSV◂ cblas_dtrsv◂ cblas_ztrsv◂ cblas_dtpsv◂ cblas_ztps◂	"STRSV, DTRSV, CTRSV, ZTRSV, STPSV, DTPSV, CTPSV, and ZTPSV (Solution of a Triangular System of Equations with a Single Right-Hand Side)" on page 354
STBMV◂ CTBMV◂ cblas_stbmv◂ cblas_ctbmv◂	DTBMV◂ ZTBMV◂ cblas_dtbmv◂ cblas_ztbmv◂	"STBMV, DTBMV, CTBMV, and ZTBMV (Matrix-Vector Product for a Triangular Band Matrix, Its Transpose, or Its Conjugate Transpose)" on page 360
STBSV◂ CTBSV◂ cblas_stbsv◂ cblas_ctbsv◂	DTBSV◂ ZTBSV◂ cblas_dtbsv◂ cblas_ztbsv◂	"STBSV, DTBSV, CTBSV, and ZTBSV (Triangular Band Equation Solve)" on page 365

◂ Level 2 BLAS

§ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs.

Sparse Matrix-Vector Linear Algebra Subprograms

The sparse matrix-vector linear algebra subprograms operate on sparse matrices using optimized storage techniques; that is, only the nonzero elements of the vector are stored. These subprograms provide similar functions to the matrix-vector subprograms.

Table 59: List of Sparse Matrix-Vector Linear Algebra Subprograms.

Long-Precision Subprogram	**Descriptive Name and Location**
DSMMX	"DSMMX (Matrix-Vector Product for a Sparse Matrix in Compressed-Matrix Storage Mode)" on page 371

Table 59: List of Sparse Matrix-Vector Linear Algebra Subprograms. (continued)

Long-Precision Subprogram	Descriptive Name and Location
DSMTM	"DSMTM (Transpose a Sparse Matrix in Compressed-Matrix Storage Mode)" on page 378
DSDMX	"DSDMX (Matrix-Vector Product for a Sparse Matrix or Its Transpose in Compressed-Diagonal Storage Mode)" on page 374

Use Considerations

If your program uses a sparse matrix stored by rows, as defined in "Storage-by-Rows" on page 115, you should first convert your sparse matrix to compressed-matrix storage mode by using the subroutine DSRSM (see "DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)" on page 1212). DSRSM converts a matrix to compressed-matrix storage mode. To convert your sparse matrix to compressed-diagonal storage mode, you need to perform this conversion in your application program before calling the ESSL subroutine.

Performance and Accuracy Considerations

1. In ESSL, the SSCAL and DSCAL subroutines provide the fastest way to zero out contiguous (stride 1) arrays, by specifying *incx* = 1 and α = 0.
2. Where possible, use the matrix-vector linear algebra subprograms, rather than the vector-scalar, to optimize performance. Because data is presented in matrices rather than vectors, multiple operations can be performed by a single ESSL subprogram.
3. Where possible, use subprograms that do multiple computations, such as SNDOT and SNAXPY, rather than individual computations, such as SDOT and SAXPY. You get better performance.
4. Many of the short-precision subprograms provide increased accuracy by accumulating results in long precision. However, when short-precision subroutines use the VSX unit to improve performance, they do not accumulate intermediate results in long precision. This is noted in the functional description of each subprogram.
5. In some of the subprograms, because implementation techniques vary to optimize performance, accuracy of the results may vary for different array sizes. In the subprograms in which this occurs, a general description of the implementation techniques is given in the functional description for each subprogram.
6. To select the sparse matrix subroutine that gives you the best performance, you must consider the layout of the data in your matrix. From this, you can determine the most efficient storage mode for your sparse matrix. ESSL provides two versions of each of its sparse matrix-vector subroutines that you can use. One operates on sparse matrices stored in compressed-matrix storage mode, and the other operates on sparse matrices stored in compressed-diagonal storage mode. These two storage modes are described in "Sparse Matrix" on page 110.

   Compressed-matrix storage mode is generally applicable. It should be used when each row of the matrix contains approximately the same number of nonzero elements. However, if the matrix has a special form—that is, where the nonzero elements are concentrated along a few diagonals—compressed-diagonal storage mode gives improved performance.
7. There are some ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 61.

Vector-Scalar Subprograms

This contains the vector-scalar subprogram descriptions.

ISAMAX, IDAMAX, ICAMAX, and IZAMAX (Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude)

Purpose

ISAMAX and IDAMAX find the position *i* of the first or last occurrence of a vector element having the maximum absolute value. ICAMAX and IZAMAX find the position *i* of the first or last occurrence of a vector element having the largest sum of the absolute values of the real and imaginary parts of the vector elements.

You get the position of the first or last occurrence of an element by specifying positive or negative stride, respectively, for vector ***x***. Regardless of the stride, the position *i* is always relative to the location specified in the calling sequence for vector ***x*** (in argument *x*).

Table 60: Data Types.

Data Types	
x	**Subprogram**
Short-precision real	ISAMAX
Long-precision real	IDAMAX
Short-precision complex	ICAMAX
Long-precision complex	IZAMAX

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	ISAMAX | IDAMAX | ICAMAX | IZAMAX (*n*, *x*, *incx*)
C and C++	isamax | idamax | icamax | izamax (*n*, *x*, *incx*);
CBLAS	cblas_isamax | cblas_idamax | cblas_icamax | cblas_izamax (*n*, *x*, *incx*);

On Entry

n
is the number of elements in vector ***x***. Specified as: an integer; $n \geq 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 60 on page 204.

incx
is the stride for vector ***x***.

Specified as: an integer. It can have any value.

On Return

Function value

is the position i of the element in the array, where:

If $incx \geq 0$, i is the position of the first occurrence.

If $incx < 0$, i is the position of the last occurrence.

Returned as:

- an integer; $0 \leq i \leq n$ (for Fortran, C, and C++)
- a CBLAS_INDEX; $0 \leq i \leq n$-1

Notes

Declare the ISAMAX, IDAMAX, ICAMAX, and IZAMAX functions in your program as returning an integer value.

Function

ICAMAX and IZAMAX find the first element x_k, where k is defined as the smallest index k, such that:

$|a_k|+|b_k| = \max\{|a_j|+|b_j| \text{ for } j = 1, n\}$
where $x_k = (a_k, b_k)$

By specifying a positive or negative stride for vector ***x***, the first or last occurrence, respectively, is found in the array. The position i, returned as the value of the function, is always figured relative to the location specified in the calling sequence for vector ***x*** (in argument x). Therefore, depending on the stride specified for $incx$, i has the following values:

For $incx \geq 0$, $i = k$
For $incx < 0$, $i = n-k+1$

See reference [“95” on page 1258]. The result is returned as a function value. If n is 0, then 0 is returned as the value of the function.

Error conditions

Computational Errors

None

Input-Argument Errors

$n < 0$

Examples

Example 1

This example shows a vector, ***x***, with a stride of 1.

Function Reference and Input:

```
                N   X   INCX
                |   |    |
IMAX = ISAMAX( 9 , X ,  1   )

X        =  (1.0, 2.0, 7.0, -8.0, -5.0, -10.0, -9.0, 10.0, 6.0)
```

Output:

```
IMAX     =  6
```

Example 2

This example shows a vector, ***x***, with a stride greater than 1.

Function Reference and Input:

```
                N   X   INCX
                |   |    |
IMAX = ISAMAX( 5 , X ,  2   )

X        =  (1.0, . , 7.0, . , -5.0, . , -9.0, . , 6.0)
```

Output:

```
IMAX     =  4
```

Example 3
This example shows a vector, ***x***, with a stride of 0.

Function Reference and Input:

```
                N   X   INCX
                |   |    |
IMAX = ISAMAX( 9 , X ,  0   )

X        =  (1.0, . , . , . , . , . , . , . , .)
```

Output:

```
IMAX     =  1
```

Example 4
This example shows a vector, ***x***, with a negative stride. Processing begins at element `X(15)`, which is 2.0.

Function Reference and Input:

```
                N   X   INCX
                |   |    |
IMAX = ISAMAX( 8 , X , -2   )

X        =  (3.0, . , 5.0, . , -8.0, . , 6.0, . , 8.0, . ,
             4.0, . , 8.0, . , 2.0)
```

Output:

```
IMAX     =  7
```

Example 5
This example shows a vector, ***x***, containing complex numbers and having a stride of 1.

Function Reference and Input:

```
                N   X   INCX
                |   |    |
IMAX = ICAMAX( 5 , X ,  1   )

X        =  ((9.0 , 2.0) , (7.0 , -8.0) , (-5.0 , -10.0) , (-4.0 , 10.0),
             (6.0 , 3.0))
```

Output:

```
IMAX     =  2
```

ISAMIN and IDAMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Absolute Value)

Purpose

These subprograms find the position *i* of the first or last occurrence of a vector element having the minimum absolute value.

You get the position of the first or last occurrence of an element by specifying positive or negative stride, respectively, for vector ***x***. Regardless of the stride, the position *i* is always relative to the location specified in the calling sequence for vector ***x*** (in argument *x*).

Table 61: Data Types.

Data Types	
x	**Subprogram**
Short-precision real	ISAMIN
Long-precision real	IDAMIN

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	ISAMIN | IDAMIN (*n*, *x*, *incx*)
C and C++	isamin | idamin (*n*, *x*, *incx*);
CBLAS	cblas_isamin | cblas_idamin (*n*, *x*, *incx*);

On Entry

n
is the number of elements in vector ***x***. Specified as: an integer; $n \geq 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 61 on page 207.

incx
is the stride for vector ***x***.

Specified as: an integer. It can have any value.

On Return

Function value
is the position *i* of the element in the array, where:

If $incx \geq 0$, *i* is the position of the first occurrence.

If $incx < 0$, *i* is the position of the last occurrence.

Returned as:

- an integer; $0 \leq i \leq n$ (for Fortran, C, and C++)
- a CBLAS_INDEX; $0 \leq i \leq n-1$

Notes

Declare the ISAMIN and IDAMIN functions in your program as returning an integer value.

Function

These subprograms find the first element x_k, where *k* is defined as the smallest index *k*, such that:

$|x_k| = \min\{|x_j| \text{ for } j = 1, n\}$

By specifying a positive or negative stride for vector ***x***, the first or last occurrence, respectively, is found in the array. The position *i*, returned as the value of the function, is always figured relative to the location specified in the calling sequence for vector ***x*** (in argument *x*). Therefore, depending on the stride specified for *incx*, *i* has the following values:

For *incx* ≥ 0, *i* = *k*
For *incx* < 0, *i* = *n*-*k*+1

See reference [“95” on page 1258]. The result is returned as a function value. If *n* is 0, then 0 is returned as the value of the function.

Error conditions

Computational Errors
None

Input-Argument Errors
n < 0

Examples

Example 1
This example shows a vector, ***x***, with a stride of 1.

Function Reference and Input:

```
               N   X   INCX
               |   |    |
IMIN = ISAMIN( 6 , X ,  1  )

X        =  (3.0, 4.0, 1.0, 8.0, 1.0, 3.0)
```

Output

```
IMIN     =  3
```

Example 2
This example shows a vector, ***x***, with a stride greater than 1.

Function Reference and Input:

```
               N   X   INCX
               |   |    |
IMIN = ISAMIN( 4 , X ,  2  )

X        =  (-3.0, . , -9.0, . , -8.0, . , 3.0)
```

Output:

```
IMIN     =  1
```

Example 3
This example shows a vector, ***x***, with a positive stride and two elements with the minimum absolute value. The position of the first occurrence is returned.

Function Reference and Input:

```
               N   X   INCX
               |   |    |
IMIN = ISAMIN( 4 , X ,  2  )

X        =  (2.0, . , -1.0, . , 4.0, . , 1.0)
```

Output:

```
IMIN     =  2
```

Example 4

This example shows a vector, ***x***, with a negative stride and two elements with the minimum absolute value. The position of the last occurrence is returned. Processing begins at element `X(7)`, which is 1.0.

Function Reference and Input:

```
                N   X   INCX
                |   |    |
IMIN = ISAMIN( 4 , X , -2  )

X        =  (2.0, . , -1.0, . , 4.0, . , 1.0)
```

Output:

```
IMIN     =  4
```

ISMAX and IDMAX (Position of the First or Last Occurrence of the Vector Element Having the Maximum Value)

Purpose

These subprograms find the position *i* of the first or last occurrence of a vector element having the maximum value.

You get the position of the first or last occurrence of an element by specifying positive or negative stride, respectively, for vector ***x***. Regardless of the stride, the position *i* is always relative to the location specified in the calling sequence for vector ***x*** (in argument *x*).

Table 62: Data Types.

Data Types	
x	**Subprogram**
Short-precision real	ISMAX
Long-precision real	IDMAX

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	ISMAX | IDMAX (*n*, *x*, *incx*)
C and C++	ismax | idmax (*n*, *x*, *incx*);
CBLAS	cblas_ismax | cblas_idmax (*n*, *x*, *incx*);

On Entry

n
is the number of elements in vector ***x***. Specified as: an integer; $n \geq 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 62 on page 209.

incx
is the stride for vector ***x***.

Specified as: an integer. It can have any value.

On Return

Function value

is the position *i* of the element in the array, where:

If *incx* ≥ 0, *i* is the position of the first occurrence.

If *incx* < 0, *i* is the position of the last occurrence.

Returned as:

- an integer; $0 \le i \le n$ (for Fortran, C, and C++)
- a CBLAS_INDEX; $0 \le i \le n\text{-}1$

Notes

Declare the ISMAX and IDMAX functions in your program as returning an integer value.

Function

These subprograms find the first element x_k, where *k* is defined as the smallest index *k*, such that:

$x_k = \max\{x_j \text{ for } j = 1, n\}$

By specifying a positive or negative stride for vector ***x***, the first or last occurrence, respectively, is found in the array. The position *i*, returned as the value of the function, is always figured relative to the location specified in the calling sequence for vector ***x*** (in argument *x*). Therefore, depending on the stride specified for *incx*, *i* has the following values:

For *incx* ≥ 0, $i = k$
For *incx* < 0, $i = n\text{-}k\text{+}1$

See reference [“95” on page 1258]. The result is returned as a function value. If *n* is 0, then 0 is returned as the value of the function.

Error conditions

Computational Errors

None

Input-Argument Errors

$n < 0$

Examples

Example 1

This example shows a vector, ***x***, with a stride of 1.

Function Reference and Input:

```
              N   X  INCX
              |   |   |
IMAX = ISMAX( 6 , X , 1  )

X        =  (3.0, 4.0, 1.0, 8.0, 1.0, 8.0)
```

Output:

```
IMAX     =  4
```

Example 2

This example shows a vector, ***x***, with a stride greater than 1.

Function Reference and Input:

```
               N   X  INCX
               |   |   |
IMAX = ISMAX( 4 , X , 2  )

X        =  (-3.0, . , 9.0, . , -8.0, . , 3.0)
```

Output:

```
IMAX     =  2
```

Example 3

This example shows a vector, ***x***, with a positive stride and two elements with the maximum value. The position of the first occurrence is returned.

Function Reference and Input:

```
               N   X  INCX
               |   |   |
IMAX = ISMAX( 4 , X , 2  )

X        =  (2.0, . , 4.0, . , 4.0, . , 1.0)
```

Output:

```
IMAX     =  2
```

Example 4

This example shows a vector, ***x***, with a negative stride and two elements with the maximum value. The position of the last occurrence is returned. Processing begins at element `X(7)`, which is 1.0.

Function Reference and Input:

```
               N   X   INCX
               |   |    |
IMAX = ISMAX( 4 , X , -2  )

X        =  (2.0, . , 4.0, . , 4.0, . , 1.0)
```

Output:

```
IMAX     =  3
```

ISMIN and IDMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Value)

Purpose

These subprograms find the position *i* of the first or last occurrence of a vector element having the minimum value.

You get the position of the first or last occurrence of an element by specifying positive or negative stride, respectively, for vector ***x***. Regardless of the stride, the position *i* is always relative to the location specified in the calling sequence for vector ***x*** (in argument *x*).

Table 63: Data Types.

Data Types	
x	**Subprogram**
Short-precision real	ISMIN

Table 63: Data Types. (continued)

Data Types	
Long-precision real	IDMIN

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	ISMIN | IDMIN (*n*, *x*, *incx*)
C and C++	ismin | idmin (*n*, *x*, *incx*);
CBLAS	cblas_ismin | cblas_idmin (*n*, *x*, *incx*);

On Entry

n

is the number of elements in vector $\boldsymbol{x}$. Specified as: an integer; $n \geq 0$.

x

is the vector $\boldsymbol{x}$ of length n. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 63 on page 211.

incx

is the stride for vector $\boldsymbol{x}$.

Specified as: an integer. It can have any value.

On Return

Function value

is the position i of the element in the array, where:

If $incx \geq 0$, i is the position of the first occurrence.

If $incx < 0$, i is the position of the last occurrence.

Returned as:

- an integer; $0 \leq i \leq n$ (for Fortran, C, and C++)
- a CBLAS_INDEX; $0 \leq i \leq n-1$

Notes

Declare the ISMIN and IDMIN functions in your program as returning an integer value.

Function

These subprograms find the first element x_k, where k is defined as the smallest index k, such that:

$x_k = \min\{x_j \text{ for } j = 1, n\}$

By specifying a positive or negative stride for vector $\boldsymbol{x}$, the first or last occurrence, respectively, is found in the array. The position i, returned as the value of the function, is always figured relative to the location specified in the calling sequence for vector $\boldsymbol{x}$ (in argument x). Therefore, depending on the stride specified for $incx$, i has the following values:

For $incx \geq 0$, $i = k$
For $incx < 0$, $i = n-k+1$

See reference [“95” on page 1258]. The result is returned as a function value. If *n* is 0, then 0 is returned as the value of the function.

Error conditions

Computational Errors
None

Input-Argument Errors
n < 0

Examples

Example 1
This example shows a vector, ***x***, with a stride of 1.

Function Reference and Input:

```
               N   X  INCX
               |   |   |
IMIN = ISMIN( 6 , X , 1  )

X        =  (3.0, 4.0, 1.0, 8.0, 1.0, 3.0)
```

Output:

```
IMIN     =  3
```

Example 2
This example shows a vector, ***x***, with a stride greater than 1.

Function Reference and Input:

```
               N   X  INCX
               |   |   |
IMIN = ISMIN( 4 , X , 2  )

X        =  (-3.0, . , -9.0, . , -8.0, . , 3.0)
```

Output:

```
IMIN     =  2
```

Example 3

This example shows a vector, ***x***, with a positive stride and two elements with the minimum value. The position of the first occurrence is returned. Processing begins at element X(7), which is 1.0.

Function Reference and Input:

```
               N   X  INCX
               |   |   |
IMIN = ISMIN( 4 , X , 2  )

X        =  (2.0, . , 1.0, . , 4.0, . , 1.0)
```

Output:

```
IMIN     =  2
```

Example 4
This example shows a vector, ***x***, with a negative stride and two elements with the minimum value. The position of the last occurrence is returned. Processing begins at element X(7), which is 1.0.

Function Reference and Input:

```
               N   X   INCX
               |   |    |
IMIN = ISMIN( 4 , X , -2  )

X        =  (2.0, . , 1.0, . , 4.0, . , 1.0)
```

Output:

```
IMIN     =  4
```

SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector)

Purpose

SASUM and DASUM compute the sum of the absolute values of the elements in vector ***x***. SCASUM and DZASUM compute the sum of the absolute values of the real and imaginary parts of the elements in vector ***x***.

Table 64: Data Types.

Data Types		
x	**Result**	**Subprogram**
Short-precision real	Short-precision real	SASUM
Long-precision real	Long-precision real	DASUM
Short-precision complex	Short-precision real	SCASUM
Long-precision complex	Long-precision real	DZASUM

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	**Syntax**
Fortran	SASUM | DASUM | SCASUM | DZASUM (*n*, *x*, *incx*)
C and C++	sasum | dasum | scasum | dzasum (*n*, *x*, *incx*);
CBLAS	cblas_sasum | cblas_dasum | cblas_scasum | cblas_dzasum (*n*, *x*, *incx*);

On Entry

n
is the number of elements in vector ***x***. Specified as: an integer; $n \ge 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 64 on page 214.

incx
is the stride for vector ***x***.

Specified as: an integer. It can have any value.

On Return

Function value
is the result of the summation. Returned as: a number of the data type indicated in Table 64 on page 214.

Note: Declare this function in your program as returning a value of the type indicated in Table 64 on page 214.

Function

SASUM and DASUM compute the sum of the absolute values of the elements of ***x***, which is expressed as follows:

$$\sum_{i=1}^{n} |x_i| = |x_1| + |x_2| + \dots + |x_n|$$

SCASUM and DZASUM compute the sum of the absolute values of the real and imaginary parts of the elements of ***x***, which is expressed as follows:

$$\sum_{i=1}^{n} \left(|a_i| + |b_i|\right) = \left(|a_1| + |b_1|\right) + \left(|a_2| + |b_2|\right) + \dots + \left(|a_n| + |b_n|\right)$$

$$\text{where } x_i = (a_i, b_i)$$

See reference [“95” on page 1258]. The result is returned as a function value. If *n* is 0, then 0.0 is returned as the value of the function. For SASUM and SCASUM, intermediate results are accumulated in long precision when the VSX unit is not used.

Error conditions

Computational Errors
None

Input-Argument Errors
$n < 0$

Examples

Example 1
This example shows a vector, ***x***, with a stride of 1.

Function Reference and Input:

```
               N   X  INCX
               |   |   |
SUMM = SASUM( 7 , X , 1  )

X        =  (1.0, -3.0, -6.0, 7.0, 5.0, 2.0, -4.0)
```

Output:

```
SUMM     =  28.0
```

Example 2
This example shows a vector, ***x***, with a stride greater than 1.

Function Reference and Input:

```
               N   X  INCX
               |   |   |
SUMM = SASUM( 4 , X , 2  )
```

```
X        =  (1.0, . , -6.0, . , 5.0, . , -4.0)
```

Output:

```
SUMM     =  16.0
```

Example 3

This example shows a vector, ***x***, with negative stride. Processing begins at element `X(7)`, which is -4.0.

Function Reference and Input:

```
              N   X   INCX
              |   |    |
SUMM = SASUM( 4 , X , -2  )

X        =  (1.0, . , -6.0, . , 5.0, . , -4.0)
```

Output:

```
SUMM     =  16.0
```

Example 4

This example shows a vector, ***x***, with a stride of 0. The result in `SUMM` is nx_1.

Function Reference and Input:

```
              N   X  INCX
              |   |   |
SUMM = SASUM( 7 , X , 0  )

X        =  (-2.0, . , . , . , . , . , .)
```

Output:

```
SUMM     =  14.0
```

Example 5

This example shows a vector, ***x***, containing complex numbers and having a stride of 1.

Function Reference and Input:

```
               N   X  INCX
               |   |   |
SUMM = SCASUM( 5 , X , 1  )

X        =  ((1.0, 2.0), (-3.0, 4.0), (5.0, -6.0 ), (-7.0, -8.0),
             (9.0, 10.0))
```

Output:

```
SUMM     =  55.0
```

SAXPY, DAXPY, CAXPY, and ZAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y)

Purpose

These subprograms perform the following computation, using the scalar α and vectors ***x*** and ***y***:

$\boldsymbol{y} \leftarrow \boldsymbol{y}+\alpha\boldsymbol{x}$

Table 65: Data Types.	
Data Types	
alpha, ***x***, ***y***	**Subprogram**
Short-precision real	SAXPY
Long-precision real	DAXPY
Short-precision complex	CAXPY
Long-precision complex	ZAXPY

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SAXPY | DAXPY | CAXPY | ZAXPY (*n*, *alpha*, *x*, *incx*, *y*, *incy*)
C and C++	saxpy | daxpy | caxpy | zaxpy (*n*, *alpha*, *x*, *incx*, *y*, *incy*);
CBLAS	cblas_saxpy | cblas_daxpy | cblas_caxpy | cblas_zaxpy (*n*, *alpha*, *x*, *incx*, *y*, *incy*);

On Entry

n

is the number of elements in vectors ***x*** and ***y***.

Specified as: an integer; $n \ge 0$.

alpha

is the scalar *alpha*.

Specified as: a number of the data type indicated in Table 65 on page 217.

x

is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 65 on page 217.

incx

is the stride for vector ***x***.

Specified as: an integer. It can have any value.

y

is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 65 on page 217.

incy

is the stride for vector ***y***.

Specified as: an integer. It can have any value.

On Return

y

is the vector ***y***, containing the results of the computation $\mathbf{y}+\alpha\mathbf{x}$. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 65 on page 217.

Notes

1. If you specify the same vector for ***x*** and ***y***, *incx* and *incy* must be equal; otherwise, results are unpredictable.

2. If you specify different vectors for ***x*** and ***y***, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

The computation is expressed as follows:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} \leftarrow \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

See reference ["95" on page 1258]. If *alpha* or *n* is zero, no computation is performed. For CAXPY, intermediate results are accumulated in long precision when the VSX unit is not used.

Error conditions

Computational Errors
None

Input-Argument Errors
$n < 0$

Examples

Example 1

This example shows vectors ***x*** and ***y*** with positive strides.

Call Statement and Input:

```
             N  ALPHA  X  INCX  Y  INCY
             |    |    |   |    |   |
CALL SAXPY( 5 , 2.0 , X , 1  , Y , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0)
```

Output:

```
Y        =  (3.0, . , 5.0, . , 7.0, . , 9.0, . , 11.0)
```

Example 2

This example shows vectors ***x*** and ***y*** having strides of opposite signs. For ***y***, which has negative stride, processing begins at element Y(5), which is 1.0.

Call Statement and Input:

```
             N  ALPHA  X  INCX  Y   INCY
             |    |    |   |    |    |
CALL SAXPY( 5 , 2.0 , X , 1  , Y , -1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
Y        =  (15.0, 12.0, 9.0, 6.0, 3.0)
```

Example 3

This example shows a vector, ***x***, with 0 stride. Vector ***x*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***x***.

Call Statement and Input:

```
            N  ALPHA  X  INCX  Y  INCY
            |    |    |   |    |   |
CALL SAXPY( 5 , 2.0 , X , 0  , Y , 1  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
Y        =  (7.0, 6.0, 5.0, 4.0, 3.0)
```

Example 4

This example shows how SAXPY can be used to compute a scalar value. In this case, vectors ***x*** and ***y*** contain scalar values and the strides for both vectors are 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input:

```
            N  ALPHA  X  INCX  Y  INCY
            |    |    |   |    |   |
CALL SAXPY( 1 , 2.0 , X , 0  , Y , 0  )

X        =  (1.0)
Y        =  (5.0)
```

Output:

```
Y        =  (7.0)
```

Example 5

This example shows how to use CAXPY, where vectors ***x*** and ***y*** contain complex numbers. In this case, vectors ***x*** and ***y*** have positive strides.

Call Statement and Input:

```
            N  ALPHA  X  INCX  Y  INCY
            |    |    |   |    |   |
CALL CAXPY( 3 ,ALPHA, X , 1  , Y , 2  )

ALPHA    =  (2.0, 3.0)
X        =  ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
Y        =  ((1.0, 1.0 ), . , (0.0, 2.0), . , (5.0, 4.0))
```

```
Y        =  ((-3.0, 8.0), . , (4.0, 8.0), . , (-4.0, 23.0))
```

>|SAXPBY and DAXPBY (Scales Vector X and Vector Y by two Scalars, Adds them to one another, and Stores in the Vector Y)

Purpose

These subprograms perform the following computation, by using the scalars α and β, and vectors ***x*** and ***y***:

$\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \beta \boldsymbol{y}$

Table 66: Data Types.	
alpha*, *beta*, *x*, *y	**Subprogram**
Short-precision real	SAXPBY
Long-precision real	DAXPBY

Note: On certain processors, SIMD algorithms can be used if alignment requirements are met. For more information, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SAXPBY | DAXPBY (*n*, *alpha*, *x*, *incx*, *beta*, *y*, *incy*)
C and C++	saxpby | daxpby (*n*, *alpha*, *x*, *incx*, *beta*, *y*, *incy*);
CBLAS	cblas_saxpby | cblas_daxpby (*n*, *alpha*, *x*, *incx*, *beta*, *y*, *incy*);

On Entry

n

is the number of elements in vectors ***x*** and ***y***.

Specified as: an integer; $n \geq 0$.

alpha

is the scalar *alpha*.

Specified as: a number of the data type indicated in Table 66 on page 220.

x

is the vector ***x*** of length *n*.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, which contains numbers of the data type indicated in Table 66 on page 220.

incx

is the stride for vector ***x***.

Specified as: an integer. The *incx* argument can have any value.

beta

is the scalar *beta*.

Specified as: a number of the data type indicated in Table 66 on page 220.

y

is the vector ***y*** of length *n*.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incy*|, which contains numbers of the data type indicated in Table 66 on page 220.

incy

is the stride for vector ***y***.

Specified as: an integer. The *incy* argument can have any value.

On Return

y

is the vector ***y***, which contains the results of the computation $\alpha \boldsymbol{x} + \beta \boldsymbol{y}$.

Returned as: a one-dimensional array, which contains numbers of the data type indicated in Table 66 on page 220.

Notes

1. If you specify same values for vectors ***x*** and ***y***, the *incx* and *incy* arguments must be equal; otherwise, results are unpredictable.
2. If you specify different values for vectors ***x*** and ***y***, the vectors must not have common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

The computation is expressed as follows:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} \leftarrow \alpha \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} + \beta \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix}$$

See reference ["95" on page 1258]. Computation is not performed in the following cases:

- *n* is zero.
- *alpha* is zero and *beta* is one.

Error conditions

Computational Errors
None

Input-Argument Errors
$n < 0$

Examples

Example 1
This example shows vectors ***x*** and ***y*** with positive strides.

Call Statement and Input:

```
              N  ALPHA  X  INCX BETA Y  INCY
              |    |    |   |    |   |   |
CALL SAXPBY ( 5 , 2.0 , X , 1 , 1.0, Y , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0)
```

Output:

```
Y        =  (3.0, . , 5.0, . , 7.0, . , 9.0, . , 11.0)
```

Example 2
This example shows vectors ***x*** and ***y*** with strides of opposite signs. For vector ***y***, which has negative stride, processing begins at element `Y(5)`, which is 1.0.

Call Statement and Input:

```
              N  ALPHA  X  INCX BETA  Y   INCY
              |    |    |   |    |    |    |
CALL SAXPBY ( 5 , 2.0 , X , 1 , 1.0,  Y , -1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
Y        =  (15.0, 12.0, 9.0, 6.0, 3.0)
```

Example 3

This example shows a vector, ***x***, with 0 stride. Vector ***x*** is treated like a vector of length *n*, all of whose elements are the same as the single element in vector ***x***.

Call Statement and Input:

```
              N  ALPHA  X  INCX BETA Y  INCY
              |    |    |   |    |   |   |
CALL SAXPBY ( 5 , 2.0 , X , 0 , 1.0, Y , 0  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
Y        =  (7.0, 6.0, 5.0, 4.0, 3.0)
```

Example 4

This example shows a vector, ***x*** that has a positive stride, a vector, ***y*** that has a zero stride, and a vector count that is greater than 1.

Call Statement and Input:

```
              N  ALPHA  X  INCX BETA Y  INCY
              |    |    |   |    |   |   |
CALL SAXPBY ( 5 , 2.0 , X , 1 , 1.0, Y , 0  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output:

```
Y        =  (15.0)
```

Example 5

This example shows a vector, ***x*** that has a negative stride, a vector, ***y*** that has a zero stride, and a vector count that is greater than 1.

Call Statement and Input:

```
              N  ALPHA  X  INCX BETA Y  INCY
              |    |    |   |    |   |   |
CALL SAXPBY ( 5 , 2.0 , X, -1 , 1.0, Y , 0  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output:

```
Y        =  (7.0)
```

Example 6

This example shows how SAXPBY can be used to compute a scalar value. In this case, vectors ***x*** and ***y*** contain scalar values and the strides for both vectors are 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input:

```
              N  ALPHA  X  INCX BETA Y  INCY
              |    |    |   |    |   |   |
CALL SAXPBY ( 5 , 2.0 , X,  0 , 1.0, Y , 0  )

X        =  (1.0)
Y        =  (5.0)
```

Output:

```
Y        =  (7.0)
```

|<

SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)

Purpose

These subprograms copy vector ***x*** to another vector, ***y***:

y←***x***

Table 67: Data Types.

Data Types	
x, ***y***	**Subprogram**
Short-precision real	SCOPY
Long-precision real	DCOPY
Short-precision complex	CCOPY
Long-precision complex	ZCOPY

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SCOPY | DCOPY | CCOPY | ZCOPY (*n*, *x*, *incx*, *y*, *incy*)
C and C++	scopy | dcopy | ccopy | zcopy (*n*, *x*, *incx*, *y*, *incy*);
CBLAS	cblas_scopy | cblas_dcopy | cblas_ccopy | cblas_zcopy (*n*, *x*, *incx*, *y*, *incy*);

On Entry

n

is the number of elements in vectors ***x*** and ***y***.

Specified as: an integer; $n \geq 0$.

x

is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 67 on page 223.

incx

is the stride for vector ***x***.

Specified as: an integer. It can have any value.

y

See On Return.

incy

is the stride for vector ***y.*** Specified as: an integer. It can have any value.

On Return

y

is the vector ***y*** of length *n*. Returned as: a one-dimensional array of (at least) length 1+(*n*-1)|*incy*|, containing numbers of the data type indicated in Table 67 on page 223.

Notes

1. If you specify the same vector for ***x*** and ***y***, *incx* and *incy* must be equal; otherwise, results are unpredictable.
2. If you specify different vectors for ***x*** and ***y***, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

The copy is expressed as follows:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} \leftarrow \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

See reference ["95" on page 1258]. If *n* is 0, no copy is performed.

Error conditions

Computational Errors

None

Input-Argument Errors

n < 0

Examples

Example 1

This example shows input vector ***x*** and output vector ***y*** with positive strides.

Call Statement and Input:

```
             N   X  INCX  Y  INCY
             |   |   |    |   |
CALL SCOPY( 5 , X , 1  , Y , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output:

```
Y        =  (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0)
```

Example 2

This example shows how to obtain a reverse copy of the input vector ***x*** by specifying strides with the same absolute value, but with opposite signs, for input vector ***x*** and output vector ***y***. For ***y***, which has a negative stride, results are stored beginning at element `Y(5)`.

Call Statement and Input:

```
             N   X  INCX  Y   INCY
             |   |   |    |    |
CALL SCOPY( 5 , X , 1  , Y , -1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output:

```
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Example 3

This example shows an input vector, ***x***, with 0 stride. Vector ***x*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***x***. This is a technique for replicating an element of a vector.

Call Statement and Input:

```
             N   X  INCX  Y  INCY
             |   |   |    |   |
CALL SCOPY( 5 , X , 0  , Y , 1  )

X        =  (13.0)
```

Output:

```
Y        =  (13.0, 13.0, 13.0, 13.0, 13.0)
```

Example 4

This example shows input vector ***x*** and output vector ***y***, containing complex numbers and having positive strides.

Call Statement and Input:

```
             N   X  INCX  Y  INCY
             |   |   |    |   |
CALL CCOPY( 4 , X , 1  , Y , 2  )

X        =  ((1.0, 1.0), (2.0, 2.0), (3.0, 3.0), (4.0, 4.0))
```

Output:

```
Y        =  ((1.0, 1.0), . , (2.0, 2.0), . , (3.0, 3.0), . ,
             (4.0, 4.0))
```

SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDOTC (Dot Product of Two Vectors)

Purpose

SDOT, DDOT, CDOTU, and ZDOTU compute the dot product of vectors ***x*** and ***y***:

$$x \bullet y$$

CDOTC and ZDOTC compute the dot product of the complex conjugate of vector ***x*** with vector ***y***:

$$\bar{x} \bullet y$$

Table 68: Data Types.

Data Types	
x, ***y***, ***dotu***, ***dotc***, **Result**	**Subprogram**
Short-precision real	SDOT
Long-precision real	DDOT
Short-precision complex	CDOTU and CDOTC
Long-precision complex	ZDOTU and ZDOTC

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	SDOT | DDOT | CDOTU | ZDOTU | CDOTC | ZDOTC (*n*, *x*, *incx*, *y*, *incy*)
C and C++	sdot | ddot | cdotu | zdotu | cdotc | zdotc (*n*, *x*, *incx*, *y*, *incy*);

CBLAS	cblas_sdot | cblas_ddot (*n*, *x*, *incx*, *y*, *incy*);
	cblas_cdotu_sub | cblas_zdotu_sub (*n*, *x*, *incx*, *y*, *incy*, *dotu*);
	cblas_cdotc_sub | cblas_zdotc_sub (*n*, *x*, *incx*, *y*, *incy*, *dotc*);

On Entry

n

is the number of elements in vectors ***x*** and ***y***.

Specified as: an integer; $n \geq 0$.

x

is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 68 on page 226.

incx

is the stride for vector ***x***.

Specified as: an integer. It can have any value.

y

is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 68 on page 226.

incy

is the stride for vector ***y***.

Specified as: an integer. It can have any value.

On Return

Function value

is the result of the dot product computation. Returned as: a number of the data type indicated in Table 68 on page 226.

dotu

is the result of the dot product computation.

Returned as: a number of the data type indicated in Table 68 on page 226.

dotc

is the result of the dot product computation.

Returned as: a number of the data type indicated in Table 68 on page 226.

Notes

Declare this function in your program as returning a value of the data type indicated in Table 68 on page 226.

Function

SDOT, DDOT, CDOTU, and ZDOTU compute the dot product of the vectors ***x*** and ***y***, which is expressed as follows:

$$x \bullet y = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n$$

CDOTC and ZDOTC compute the dot product of the complex conjugate of vector ***x*** with vector ***y***, which is expressed as follows:

$$\bar{x} \bullet y = \bar{x}_1 y_1 + \bar{x}_2 y_2 + \ldots + \bar{x}_n y_n$$

See reference [“95” on page 1258]. The result is returned as a function value. If *n* is 0, then zero is returned as the value of the function.

For SDOT, CDOTU, and CDOTC, intermediate results are accumulated in long precision when the VSX unit is not used.

Error conditions

Computational Errors
None

Input-Argument Errors
$n < 0$

Examples

Example 1

This example shows how to compute the dot product of two vectors, ***x*** and ***y***, having strides of 1.

Function Reference and Input:

```
              N   X  INCX  Y  INCY
              |   |   |    |   |
DOTT = SDOT( 5 , X , 1  , Y , 1  )

X        =  (1.0, 2.0, -3.0, 4.0, 5.0)
Y        =  (9.0, 8.0, 7.0, -6.0, 5.0)
```

Output:

```
DOTT     =  (9.0 + 16.0 - 21.0 - 24.0 + 25.0) = 5.0
```

Example 2

This example shows how to compute the dot product of a vector, ***x***, with a stride of 1, and a vector, ***y***, with a stride greater than 1.

Function Reference and Input:

```
              N   X  INCX  Y  INCY
              |   |   |    |   |
DOTT = SDOT( 5 , X , 1  , Y , 2  )

X        =  (1.0, 2.0, -3.0, 4.0, 5.0)
Y        =  (9.0, . , 7.0, . , 5.0, . , -3.0, . , 1.0)
```

Output:

```
DOTT     =  (9.0 + 14.0 - 15.0 - 12.0 + 5.0) = 1.0
```

Example 3

This example shows how to compute the dot product of a vector, ***x***, with a negative stride, and a vector, ***y***, with a stride greater than 1. For ***x***, processing begins at element X(5), which is 5.0.

Function Reference and Input:

```
             N   X   INCX  Y  INCY
             |   |    |    |   |
DOTT = SDOT( 5 , X , -1  , Y , 2  )

X        =  (1.0, 2.0, -3.0, 4.0, 5.0)
Y        =  (9.0, . , 7.0, . , 5.0, . , -3.0, . , 1.0)
```

Output:

```
DOTT     =  (45.0 + 28.0 - 15.0 - 6.0 + 1.0) = 53.0
```

Example 4

This example shows how to compute the dot product of a vector, ***x***, with a stride of 0, and a vector, ***y***, with a stride of 1. The result in DOTT is $x_1(y_1+...+y_n)$.

Function Reference and Input:

```
             N   X  INCX  Y  INCY
             |   |   |    |   |
DOTT = SDOT( 5 , X , 0  , Y , 1  )

X        =  (1.0, . , . , . , .)
Y        =  (9.0, 8.0, 7.0, -6.0, 5.0)
```

Output:

```
DOTT     =  (1.0) × (9.0 + 8.0 + 7.0 - 6.0 + 5.0) = 23.0
```

Example 5

This example shows how to compute the dot product of two vectors, ***x*** and ***y***, with strides of 0. The result in DOTT is nx_1y_1.

Function Reference and Input:

```
             N   X  INCX  Y  INCY
             |   |   |    |   |
DOTT = SDOT( 5 , X , 0  , Y , 0  )

X        =  (1.0, . , . , . , .)
Y        =  (9.0, . , . , . , .)
```

Output:

```
DOTT     =  (5) × (1.0) × (9.0) = 45.0
```

Example 6

This example shows how to compute the dot product of two vectors, ***x*** and ***y***, containing complex numbers, where ***x*** has a stride of 1, and ***y*** has a stride greater than 1.

Function Reference and Input:

```
              N   X  INCX  Y  INCY
              |   |   |    |   |
DOTT = CDOTU( 3 , X , 1  , Y , 2  )

X        =  ((1.0, 2.0), (3.0, -4.0), (-5.0, 6.0))
Y        =  ((10.0, 9.0), . , (-6.0, 5.0), . , (2.0, 1.0))
```

Output:

```
DOTT     =  ((10.0 - 18.0 - 10.0) - (18.0 - 20.0 + 6.0),
             (9.0 + 15.0 - 5.0) + (20.0 + 24.0 + 12.0))
         =  (-22.0, 75.0)
```

Example 7

This example shows how to compute the dot product of the conjugate of a vector, ***x***, with vector ***y***, both containing complex numbers, where ***x*** has a stride of 1, and ***y*** has a stride greater than 1.

Function Reference and Input:

```
              N   X  INCX  Y  INCY
              |   |   |    |   |
DOTT = CDOTC( 3 , X , 1  , Y , 2  )

X        =  ((1.0, 2.0), (3.0, -4.0), (-5.0, 6.0))
Y        =  ((10.0, 9.0), . , (-6.0, 5.0), . , (2.0, 1.0))
```

Output:

```
DOTT     =  ((10.0 - 18.0 - 10.0) + (18.0 - 20.0 + 6.0),
             (9.0  +  15.0  -  5.0)  -  (20.0  +  24.0  +  12.0))
         =  (-14.0, -37.0)
```

SNAXPY and DNAXPY (Compute SAXPY or DAXPY N Times)

Purpose

These subprograms compute SAXPY or DAXPY, respectively, *n* times:

$\boldsymbol{y}_i \leftarrow \boldsymbol{y}_i + \alpha_i \boldsymbol{x}_i$ for $i = 1, n$

where each α_i is a scalar value, contained in the vector $\boldsymbol{\alpha}$, and each $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ are vectors, contained in vectors (or matrices) ***x*** and ***y***, respectively. For an explanation of the SAXPY and DAXPY computations, see "SAXPY, DAXPY, CAXPY, and ZAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y)" on page 216.

Table 69: Data Types.

Data Types	
α, x, y	**Subprogram**
Short-precision real	SNAXPY
Long-precision real	DNAXPY

Syntax

Language	Syntax
Fortran	CALL SNAXPY | DNAXPY (*n*, *m*, *α*, *inca*, *x*, *incxi*, *incxo*, *y*, *incyi*, *incyo*)
C and C++	snaxpy | dnaxpy (*n*, *m*, *α*, *inca*, *x*, *incxi*, *incxo*, *y*, *incyi*, *incyo*);

On Entry

n

is the number of SAXPY or DAXPY computations to be performed and the number of elements in vector $\boldsymbol{\alpha}$.

Specified as: an integer; $n \geq 0$.

m

is the number of elements in vectors $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ for each SAXPY or DAXPY computation.

Specified as: an integer; $m \geq 0$.

a

is the vector $\boldsymbol{a}$ of length n, containing the scalar values α_i, used in each computation of $\boldsymbol{y}_i + \alpha_i \boldsymbol{x}_i$.

Specified as: a one-dimensional array of (at least) length $1+(n-1)|inca|$, containing numbers of the data type indicated in Table 69 on page 229.

inca

is the stride for vector $\boldsymbol{a}$.

Specified as: an integer. It can have any value.

x

is the vector (or matrix) $\boldsymbol{x}$, containing the $\boldsymbol{x}_i$ vectors of length m, used in the n computations of $\boldsymbol{y}_i + \alpha_i \boldsymbol{x}_i$. Specified as: a one- or two-dimensional array of (at least) length $(1+(n-1)(incxo)) + (m-1)|incxi|$, containing numbers of the data type indicated in Table 69 on page 229.

incxi

is the stride for $\boldsymbol{x}$ in the inner loop—that is, the stride identifying the elements in each vector $\boldsymbol{x}_i$.

Specified as: an integer. It can have any value.

incxo

is the stride for $\boldsymbol{x}$ in the outer loop—that is, the stride identifying each vector $\boldsymbol{x}_i$ in $\boldsymbol{x}$.

Specified as: an integer; $incxo \geq 0$.

y

is the vector (or matrix) $\boldsymbol{y}$, containing the $\boldsymbol{y}_i$ vectors of length m, used in the n computations of $\boldsymbol{y}_i + \alpha_i \boldsymbol{x}_i$. Specified as: a one- or two-dimensional array of (at least) length $(1+(n-1)(incyo)) + (m-1)|incyi|$, containing numbers of the data type indicated in Table 69 on page 229.

incyi

is the stride for $\boldsymbol{y}$ in the inner loop—that is, the stride identifying the elements in each vector $\boldsymbol{y}_i$ in $\boldsymbol{y}$. Specified as: an integer; $incyi > 0$ or $incyi < 0$.

incyo

is the stride for $\boldsymbol{y}$ in the outer loop—that is, the stride identifying each vector $\boldsymbol{y}_i$ in $\boldsymbol{y}$.

Specified as: an integer; $incyo \geq 0$.

On Return

y

is the vector (or matrix) $\boldsymbol{y}$, containing the $\boldsymbol{y}_i$ vectors of length m, which contain the results of the n SAXPY or DAXPY computations, $\boldsymbol{y}_i + \alpha_i \boldsymbol{x}_i$ for $i = 1, n$. Returned as: a one- or two-dimensional array, containing numbers of the data type indicated in Table 69 on page 229.

Notes

Vector $\boldsymbol{y}$ must have no common elements with vector $\boldsymbol{a}$ or vector $\boldsymbol{x}$; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

The SAXPY or DAXPY computations:

$\boldsymbol{y} \leftarrow \boldsymbol{y} + \alpha \boldsymbol{x}$

are performed n times. This is expressed as follows:

$\boldsymbol{y}_i \leftarrow \boldsymbol{y}_i + \alpha_i \boldsymbol{x}_i \quad$ for $i = 1, n$

where each α_i is a scalar value, contained in the vector $\boldsymbol{\alpha}$, and each $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ are vectors, contained in vectors (or matrices) $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively.

Each computation of SAXPY or DAXPY (see “SAXPY, DAXPY, CAXPY, and ZAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y)” on page 216) uses the length of the $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ vectors, *m*, for its input argument, *n*. It also uses the strides for the inner loop, *incxi* and *incyi*, for its parameters *incx* and *incy*, respectively. See “Function” on page 218 for a description of how the computation is done.

The outer loop of the SNAXPY or DNAXPY computation uses the strides of *inca*, *incxo*, and *incyo* to locate the elements in $\boldsymbol{\alpha}$ and vectors in $\boldsymbol{x}$ and $\boldsymbol{y}$ for each *i*-th computation. These are:

For $i = 1, n$:

$\alpha_{((i-1)inca+1)}$ for $inca \geq 0$
$\alpha_{((i-n)inca+1)}$ for $inca < 0$
$x_{((i-1)incxo+1)}$
$y_{((i-1)incyo+1)}$

If *m* or *n* is 0, no computation is performed.

Error conditions

Computational Errors

None

Input-Argument Errors

1. $n < 0$
2. $m < 0$
3. $incxo < 0$
4. $incyi = 0$
5. $incyo < 0$

Examples

Example 1

This example shows vectors, contained in matrices, with the stride of the inner loops *incxi* and *incyi* equal to 1.

Call Statement and Input:

```
             N   M   A  INCA  X  INCXI INCXO  Y  INCYI INCYO
             |   |   |   |    |    |     |    |    |     |
CALL SNAXPY( 3 , 4 , A , 1  , X ,  1  , 10  , Y ,  1  ,  5  )

A        =  (3.0, 2.0, 4.0)
```

```
        | 1.0  4.0  3.0 |
        | 2.0  3.0  4.0 |
        | 3.0  2.0  2.0 |
        | 4.0  1.0  1.0 |
X    =  |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
```

```
        | 4.0  1.0  3.0 |
        | 3.0  2.0  4.0 |
Y    =  | 2.0  3.0  2.0 |
        | 1.0  4.0  1.0 |
        |  .    .    .  |
```

Output:

```
        |  7.0  9.0  15.0 |
        |  9.0  8.0  20.0 |
Y    =  | 11.0  7.0  10.0 |
        | 13.0  6.0   5.0 |
        |   .    .     .  |
```

Example 2

This example shows vectors, contained in matrices, with a stride of the inner loop *incxi* greater than 1.

Call Statement and Input:

```
             N   M   A  INCA  X  INCXI INCXO  Y  INCYI INCYO
             |   |   |   |    |    |     |    |    |     |
CALL SNAXPY( 3 , 4 , A , 1  , X ,  2  , 10  , Y ,  1  ,  5  )

A        =  (3.0, 2.0, 4.0)
```

```
        | 1.0  4.0  3.0 |
        |  .    .    .  |
        | 2.0  3.0  4.0 |
        |  .    .    .  |
X    =  | 3.0  2.0  2.0 |
        |  .    .    .  |
        | 4.0  1.0  1.0 |
        |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
```

```
        | 4.0  1.0  3.0 |
        | 3.0  2.0  4.0 |
Y    =  | 2.0  3.0  2.0 |
        | 1.0  4.0  1.0 |
        |  .    .    .  |
```

Output:

Y =(same as output Y in Example 1)

Example 3

This example shows vectors, contained in matrices, with a negative stride, *incyi*, for the inner loop.

Call Statement and Input:

```
             N   M   A  INCA  X  INCXI INCXO  Y  INCYI INCYO
             |   |   |   |    |    |     |    |    |     |
CALL SNAXPY( 3 , 4 , A , 1  , X ,  1  , 10  , Y , -1  ,  5  )

A        =  (3.0, 2.0, 4.0)
```

```
        | 1.0  4.0  3.0 |
        | 2.0  3.0  4.0 |
        | 3.0  2.0  2.0 |
        | 4.0  1.0  1.0 |
X    =  |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
```

```
        | 1.0  4.0  1.0 |
        | 2.0  3.0  2.0 |
Y    =  | 3.0  2.0  4.0 |
        | 4.0  1.0  3.0 |
        |  .    .    .  |
```

Output:

```
        | 13.0  6.0   5.0 |
        | 11.0  7.0  10.0 |
Y    =  |  9.0  8.0  20.0 |
        |  7.0  9.0  15.0 |
        |   .    .     .  |
```

Example 4

This example shows vectors, contained in matrices, with a negative stride, *inca*, for vector ***a***. For vector ***a***, processing begins at element A(5), which is 3.0.

Call Statement and Input:

```
             N   M   A   INCA  X  INCXI INCXO  Y  INCYI INCYO
             |   |   |    |    |    |     |    |    |     |
CALL SNAXPY( 3 , 4 , A , -2  , X ,  1  , 10  , Y ,  1  ,  5  )

A        =  (4.0, . , 2.0, . , 3.0)
```

```
        | 1.0  4.0  3.0 |
        | 2.0  3.0  4.0 |
        | 3.0  2.0  2.0 |
        | 4.0  1.0  1.0 |
X    =  |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
```

```
        | 4.0  1.0  3.0 |
        | 3.0  2.0  4.0 |
Y    =  | 2.0  3.0  2.0 |
        | 1.0  4.0  1.0 |
        |  .    .    .  |
```

Output:

Y =(same as output Y in Example 1)

SNDOT and DNDOT (Compute Special Dot Products N Times)

Purpose

These subprograms compute one of the following special dot products *n* times:

Special Dot Products	
$s_i \leftarrow \boldsymbol{x}_i \bullet \boldsymbol{y}_i$	Store positive dot product
$s_i \leftarrow -\boldsymbol{x}_i \bullet \boldsymbol{y}_i$	Store negative dot product

Special Dot Products

$s_i \leftarrow s_i + \mathbf{x}_i \bullet \mathbf{y}_i$	Accumulate positive dot product
$s_i \leftarrow s_i - \mathbf{x}_i \bullet \mathbf{y}_i$	Accumulate negative dot product

for $i = 1, n$

where each s_i is an element in vector ***s***, and each $\mathbf{x}_i$ and $\mathbf{y}_i$ are vectors contained in vectors (or matrices) ***x*** and ***y***, respectively.

Table 70: Data Types.

Data Types	
s, ***x***, ***y***	**Subprogram**
Short-precision real	SNDOT
Long-precision real	DNDOT

Syntax

Language	**Syntax**
Fortran	CALL SNDOT | DNDOT (*n*, *m*, *s*, *incs*, *isw*, *x*, *incxi*, *incxo*, *y*, *incyi*, *incyo*)
C and C++	sndot | dndot (*n*, *m*, *s*, *incs*, *isw*, *x*, *incxi*, *incxo*, *y*, *incyi*, *incyo*);

On Entry

n

is the number of dot product computations to be performed and the number of elements in the vector ***s***.

Specified as: an integer; $n \geq 0$.

m

is the number of elements in vectors $\mathbf{x}_i$ and $\mathbf{y}_i$ for each dot product computation.

Specified as: an integer; $m \geq 0$.

s

is the vector ***s***, containing the *n* scalar values s_i, where: If *isw* = 1 or 2, s_i is not used in the computation (no input value specified.)

If *isw* = 3 or 4, s_i is used in the computation (input value specified.)

Specified as: a one-dimensional array of (at least) length $1+(n-1)|incs|$, containing numbers of the data type indicated in Table 70 on page 234.

incs

is the stride for vector ***s***.

Specified as: an integer; *incs* > 0 or *incs* < 0.

isw

indicates the type of computation to perform, depending on the value specified:

If *isw* = 1, $s_i \leftarrow \mathbf{x}_i \bullet \mathbf{y}_i$

If *isw* = 2, $s_i \leftarrow -\mathbf{x}_i \bullet \mathbf{y}_i$

If *isw* = 3, $s_i \leftarrow s_i + \mathbf{x}_i \bullet \mathbf{y}_i$

If *isw* = 4, $s_i \leftarrow s_i - \mathbf{x}_i \bullet \mathbf{y}_i$

where $i = 1, n$

Specified as: an integer. Its value must be 1, 2, 3, or 4.

x
: is the vector (or matrix) $\boldsymbol{x}$, containing the $\boldsymbol{x}_i$ vectors of length m, used in the n dot product computations. Specified as: a one- or two-dimensional array of (at least) length $(1+(n-1)(incxo))+(m-1)|incxi|$, containing numbers of the data type indicated in Table 70 on page 234.

incxi
: is the stride for $\boldsymbol{x}$ in the inner loop—that is, the stride identifying the elements in each vector $\boldsymbol{x}_i$.

Specified as: an integer. It can have any value.

incxo
: is the stride for $\boldsymbol{x}$ in the outer loop—that is, the stride identifying each vector $\boldsymbol{x}_i$ in $\boldsymbol{x}$.

Specified as: an integer; $incxo \geq 0$.

y
: is the vector (or matrix) $\boldsymbol{y}$, containing the $\boldsymbol{y}_i$ vectors of length m, used in the n dot product computations. Specified as: a one- or two-dimensional array of (at least) length $(1+(n-1)(incyo)) + (m-1)|incyi|$, containing numbers of the data type indicated in Table 70 on page 234.

incyi
: is the stride for $\boldsymbol{y}$ in the inner loop—that is, the stride identifying the elements in each vector $\boldsymbol{y}_i$.

Specified as: an integer. It can have any value.

incyo
: is the stride for $\boldsymbol{y}$ in the outer loop—that is, the stride identifying each vector $\boldsymbol{y}_i$ in $\boldsymbol{y}$.

Specified as: an integer; $incyo \geq 0$.

On Return

s
: is the vector $\boldsymbol{s}$ of length n, containing the results of the n dot product computations. The type of dot product computation depends of the value specified for *isw*.

If $isw = 1$, $s_i \leftarrow \boldsymbol{x}_i \bullet \boldsymbol{y}_i$

If $isw = 2$, $s_i \leftarrow -\boldsymbol{x}_i \bullet \boldsymbol{y}_i$

If $isw = 3$, $s_i \leftarrow s_i + \boldsymbol{x}_i \bullet \boldsymbol{y}_i$

If $isw = 4$, $s_i \leftarrow s_i - \boldsymbol{x}_i \bullet \boldsymbol{y}_i$

where $i = 1, n$

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 70 on page 234.

Function

The four possible computations that can be performed by these subprograms are:

Four Possible Computations Performed by These Subroutines

$s_i \leftarrow \boldsymbol{x}_i \bullet \boldsymbol{y}_i$	Store positive dot product
$s_i \leftarrow -\boldsymbol{x}_i \bullet \boldsymbol{y}_i$	Store negative dot product
$s_i \leftarrow s_i+\boldsymbol{x}_i \bullet \boldsymbol{y}_i$	Accumulate positive dot product
$s_i \leftarrow s_i-\boldsymbol{x}_i \bullet \boldsymbol{y}_i$	Accumulate negative dot product

for $i = 1, n$

where each s_i is a scalar element in the vector ***s*** of length *n*, and each of the *n* $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ vectors of length *m* are contained in vectors (or matrices) ***x*** and ***y***, respectively. Each computation uses the dot product, which is expressed:

$\boldsymbol{x}_i \bullet \boldsymbol{y}_i = u_1v_1 + u_2v_2 + \dots + u_mv_m$

where u_i and v_i are elements of $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$, respectively. To find the elements for the computation, it uses:

- The strides for the inner loops, *incxi* and *incyi*, to locate the elements in vectors $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$, respectively.
- The strides for the outer loops, *incs, incxo*, and *incyo*, to locate the element s_i in vector ***s*** and the vectors $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ in vectors (or matrices) ***x*** and ***y***, respectively.

If *m* or *n* is 0, no computation is performed. For SNDOT, intermediate results are accumulated in long precision when the VSX unit is not used.

Error conditions

Computational Errors
None

Input-Argument Errors

1. $n < 0$
2. $m < 0$
3. *incs* = 0
4. *isw* < 1 or *isw* > 4
5. *incxo* < 0
6. *incyo* < 0

Examples

Example 1

This example shows a store positive dot product computation using vectors with positive strides.

Call Statement and Input:

```
             N    M    S  INCS ISW  X  INCXI INCXO  Y  INCYI INCYO
             |    |    |   |    |   |    |     |    |    |     |
CALL SNDOT( 3 , 4 , S , 1  , 1 , X ,  1  ,  4  , Y ,  1  ,  4  )

                      ┐
        | 1.0  2.0  3.0 |
X    =  | 2.0  3.0  4.0 |
        | 3.0  4.0  5.0 |
        | 4.0  5.0  6.0 |
                      ┘

                      ┐
        | 4.0  3.0  2.0 |
Y    =  | 3.0  2.0  1.0 |
        | 2.0  1.0  4.0 |
        | 1.0  4.0  3.0 |
                      ┘
```

Output:

```
S        =  (20.0, 36.0, 48.0)
```

Example 2

This example shows a store negative dot product computation using vectors with positive and negative strides.

Call Statement and Input:

```
            N   M   S   INCS ISW  X  INCXI INCXO  Y  INCYI INCYO
            |   |   |    |    |   |    |     |    |    |     |
CALL SNDOT( 3 , 4 , S , -1  , 2 , X ,  2  , 10  , Y , -1  ,  6  )
```

```
        ┌              ┐
        | 1.0  2.0  3.0 |
        |  .    .    .  |
        | 2.0  3.0  4.0 |
        |  .    .    .  |
X    =  | 3.0  4.0  5.0 |
        |  .    .    .  |
        | 4.0  5.0  6.0 |
        |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
        └              ┘
```

```
        ┌              ┐
        | 4.0  3.0  2.0 |
        | 3.0  2.0  1.0 |
Y    =  | 2.0  1.0  4.0 |
        | 1.0  4.0  3.0 |
        |  .    .    .  |
        |  .    .    .  |
        └              ┘
```

Output:

```
S        =  (-42.0, -34.0, -30.0)
```

Example 3

This example shows an accumulative positive dot product using vectors with positive and negative strides.

Call Statement and Input:

```
            N   M   S  INCS ISW  X  INCXI INCXO  Y  INCYI INCYO
            |   |   |   |    |   |    |     |    |    |     |
CALL SNDOT( 3 , 4 , S , 1  , 3 , X , -2  , 10  , Y ,  2  , 10  )

S        =  (2.0, 5.0, 8.0)
```

```
        ┌              ┐
        | 1.0  2.0  3.0 |
        |  .    .    .  |
        | 2.0  3.0  4.0 |
        |  .    .    .  |
X    =  | 3.0  4.0  5.0 |
        |  .    .    .  |
        | 4.0  5.0  6.0 |
        |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
        └              ┘
```

```
        ┌              ┐
        | 4.0  3.0  2.0 |
        |  .    .    .  |
        | 3.0  2.0  1.0 |
        |  .    .    .  |
Y    =  | 2.0  1.0  4.0 |
        |  .    .    .  |
        | 1.0  4.0  3.0 |
        |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
        └              ┘
```

Output:

```
S        =  (32.0, 39.0, 50.0)
```

Example 4

This example shows an accumulative negative dot product using vectors with positive and negative strides.

Call Statement and Input:

```
            N   M   S   INCS ISW  X  INCXI INCXO  Y  INCYI INCYO
            |   |   |    |    |   |    |     |    |    |     |
CALL SNDOT( 3 , 4 , S , -1  , 4 , X ,  1  ,  6  , Y ,  2  , 10  )
```

```
S        =  (3.0, 6.0, 9.0)
                          ┐
        | 1.0  2.0  3.0 |
        | 2.0  3.0  4.0 |
X    =  | 3.0  4.0  5.0 |
        | 4.0  5.0  6.0 |
        |  .    .    .  |
        |  .    .    .  |
                          ┘
```

```
        ┌               ┐
        | 4.0  3.0  2.0 |
        |  .    .    .  |
        | 3.0  2.0  1.0 |
        |  .    .    .  |
Y    =  | 2.0  1.0  4.0 |
        |  .    .    .  |
        | 1.0  4.0  3.0 |
        |  .    .    .  |
        |  .    .    .  |
        |  .    .    .  |
                          ┘
```

Output:

```
S        =  (-45.0, -30.0, -11.0)
```

SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow)

Purpose

These subprograms compute the Euclidean length (l_2 norm) of vector ***x***, with scaling of input to avoid destructive underflow and overflow.

Table 71: Data Types.

Data Types		
x	**Result**	**Subprogram**
Short-precision real	Short-precision real	SNRM2
Long-precision real	Long-precision real	DNRM2
Short-precision complex	Short-precision real	SCNRM2
Long-precision complex	Long-precision real	DZNRM2

Note:

1. If there is a possibility that your data will cause the computation to overflow or underflow, you should use these subroutines instead of SNORM2, DNORM2, CNORM2, and ZNORM2, because the intermediate computational results may exceed the maximum or minimum limits of the machine. “Notes” on page 242 explains how to estimate whether your data will cause an overflow or underflow.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 34.

Syntax

Language	Syntax
Fortran	SNRM2 | DNRM2 | SCNRM2 | DZNRM2 (*n*, *x*, *incx*)
C and C++	snrm2 | dnrm2 | scnrm2 | dznrm2 (*n*, *x*, *incx*);
CBLAS	cblas_snrm2 | cblas_dnrm2 | cblas_scnrm2 | cblas_dznrm2 (*n*, *x*, *incx*);

On Entry

n

is the number of elements in vector ***x***. Specified as: an integer; $n \geq 0$.

x

is the vector ***x*** of length *n*, whose Euclidean length is to be computed.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 71 on page 238.

incx

is the stride for vector ***x***.

Specified as: an integer. It can have any value.

On Return

Function value

is the Euclidean length (l_2 norm) of the vector ***x***. Returned as: a number of the data type indicated in Table 71 on page 238.

Notes

Declare this function in your program as returning a value of the data type indicated in Table 71 on page 238.

Function

The Euclidean length (l_2 norm) of vector ***x*** is expressed as follows, with scaling of input to avoid destructive underflow and overflow:

$$\sqrt{|x_1|^2 + |x_2|^2 + \ldots + |x_n|^2}$$

See reference [“95” on page 1258]. The result is returned as the function value. If *n* is 0, then 0.0 is returned as the value of the function.

For SNRM2 and SCNRM2, the sum of the squares of the absolute values of the elements is accumulated in long precision. The square root of this long-precision sum is then computed and, if necessary, is unscaled.

Although these subroutines eliminate destructive underflow, nondestructive underflows may occur if the input elements differ greatly in magnitude. This does not affect accuracy, but it degrades performance. The system default is to mask underflow, which improves the performance of these subroutines.

Error conditions

Computational Errors
None

Input-Argument Errors
$n < 0$

Examples

Important Information About the Following Examples: Workstations use workstation architecture precisions: ANSI/IEEE 32-bit and 64-bit binary floating-point format. The ranges are:

- For short-precision: 3.37×10^{-38} to 3.37×10^{38}
- For long-precision: 1.67×10^{-308} to 1.67×10^{308}

Example 1
This example shows a vector, ***x***, whose elements must be scaled to prevent overflow.

```
                N   X  INCX
                |   |   |
DNORM = DNRM2( 6 , X , 1  )

X      = (0.68056D+200, 0.25521D+200, 0.34028D+200,
          0.85071D+200, 0.25521D+200, 0.85071D+200)
```

Output:

```
DNORM    =  0.1469D+201
```

Example 2
This example shows a vector, ***x***, whose elements must be scaled to prevent destructive underflow.

Function Reference and Input:

```
                N   X  INCX
                |   |   |
DNORM = DNRM2( 4 , X , 2  )

X     = (0.10795D-200, . , 0.10795D-200, . , 0.10795D-200,
         . , 0.10795D-200)
```

Output:

```
DNORM    =  0.21590D-200
```

Example 3
This example shows a vector, ***x***, with a stride of 0. The result in SNORM is:

$$\sqrt{nx_1^2}$$

Function Reference and Input:

```
                N   X  INCX
                |   |   |
SNORM = SNRM2( 4 , X , 0  )

X        =  (4.0)
```

Output:

```
SNORM    =  8.0
```

Example 4

This example shows a vector, ***x***, containing complex numbers, and whose elements must be scaled to prevent overflow.

Function Reference and Input:

```
                  N   X  INCX
                  |   |   |
DZNORM = DZNRM2( 3 , X , 1  )

X      = ((0.68056D+200, 0.25521D+200), (0.34028D+200, 0.85071D+200),
          (0.25521D+200, 0.85071D+200))
```

Output:

```
DZNORM   =  0.1469D+201
```

Example 5

This example shows a vector, ***x***, containing complex numbers, and whose elements must be scaled to prevent destructive underflow.

Function Reference and Input:

```
                  N   X  INCX
                  |   |   |
DZNORM = DZNRM2( 2 , X , 2  )

X      = ((0.10795D-200, 0.10795D-200), . ,
          (0.10795D-200, 0.10795D-200))
```

Output:

```
DZNORM   =  0.2159D-200
```

SNORM2, DNORM2, CNORM2, and ZNORM2 (Euclidean Length of a Vector with No Scaling of Input)

Purpose

These subprograms compute the euclidean length (l_2 norm) of vector ***x*** with no scaling of input.

Table 72: Data Types.

Data Types		
x	**Result**	**Subprogram**
Short-precision real	Short-precision real	SNORM2
Long-precision real	Long-precision real	DNORM2
Short-precision complex	Short-precision real	CNORM2
Long-precision complex	Long-precision real	ZNORM2

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 34.

Syntax

Language	Syntax
Fortran	SNORM2 | DNORM2 | CNORM2 | ZNORM2 (*n*, *x*, *incx*)
C and C++	snorm2 | dnorm2 | cnorm2 | znorm2 (*n*, *x*, *incx*);

On Entry

n

is the number of elements in vector ***x***. Specified as: an integer; $n \geq 0$.

x

is the vector ***x*** of length *n*, whose euclidean length is to be computed.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 72 on page 241.

incx

is the stride for vector ***x***.

Specified as: an integer. It can have any value.

On Return

Function value

is the euclidean length (l_2 norm) of the vector ***x***. Returned as: a number of the data type indicated in Table 72 on page 241.

Notes

1. This subroutine does not underflow or overflow if the values of the elements in vector ***x*** conform to the following conditions. If these conditions are violated, overflow or destructive underflow may occur:
   - For short-precision numbers:

     Any valid short-precision number.
   - For long-precision numbers:

     $|x_i| = 0$ or $0.10010E-145 < |x_i| < 0.13408E+155$ for $i = 1, n$
2. Declare this function in your program as returning a value of the data type indicated in Table 72 on page 241.

Function

The euclidean length (l_2 norm) of vector ***x*** is expressed as follows with no scaling of input:

$$\sqrt{|x_1|^2 + |x_2|^2 + \ldots + |x_n|^2}$$

See reference [“95” on page 1258]. The result is returned as the function value. If *n* is 0, then 0.0 is returned as the value of the function.

For SNORM2 and CNORM2, the sum of the squares of the absolute values of the elements is accumulated in long-precision. The square root of this long-precision sum is then computed.

This subroutine should not be used if the values in vector ***x*** do not conform to the restriction given in “Notes” on page 242.

Error conditions

Computational Errors
None

Input-Argument Errors
$n < 0$

Examples

Example 1
This example shows a vector, ***x***, with a stride of 1.

Function Reference and Input:

```
                 N   X  INCX
                 |   |   |
SNORM = SNORM2( 6 , X , 1  )

X        =  (3.0, 4.0, 1.0, 8.0, 1.0, 3.0)
```

Output:

```
SNORM    =  10.0
```

Example 2
This example shows a vector, ***x***, with a stride greater than 1.

Function Reference and Input:

```
                 N   X  INCX
                 |   |   |
SNORM = SNORM2( 6 , X , 2  )

X        =  (3.0, . , 4.0, . , 1.0, . , 8.0, . , 1.0, . , 3.0)
```

Output:

```
SNORM    =  10.0
```

Example 3
This example shows a vector, ***x***, with a stride of 0. The result in SNORM is:

$$\sqrt{nx_1^2}$$

Function Reference and Input:

```
                 N   X  INCX
                 |   |   |
SNORM = SNORM2( 4 , X , 0  )

X        =  (4.0)
```

Output:

```
SNORM    =  8.0
```

Example 4
This example shows a vector, ***x***, containing complex numbers and having a stride of 1.

Function Reference and Input:

```
                 N   X  INCX
                 |   |   |
CNORM = CNORM2( 3 , X , 1  )

X        =  ((3.0, 4.0), (1.0, 8.0), (-1.0, 3.0))
```

Output:

```
CNORM    =  10.0
```

SROTG, DROTG, CROTG, and ZROTG (Construct a Givens Plane Rotation)

Purpose

SROTG and DROTG construct a real Givens plane rotation, and CROTG and ZROTG construct a complex Givens plane rotation. The computations use rotational elimination parameters *a* and *b*. Values are returned for *r*, as well as the cosine *c* and the sine *s* of the angle of rotation. SROTG and DROTG also return a value for *z*.

Note: Throughout this description, the symbols *r* and *z* are used to represent two of the output values returned for this computation. It is important to note that the values for *r* and *z* are actually returned in the input-output arguments *a* and *b*, respectively, overwriting the original values passed in *a* and *b*.

Table 73: Data Types.

Data Types			
a, *b*, *r*, *s*	*c*	*z*	**Subprogram**
Short-precision real	Short-precision real	Short-precision real	SROTG
Long-precision real	Long-precision real	Long-precision real	DROTG
Short-precision complex	Short-precision real	(No value returned)	CROTG
Long-precision complex	Long-precision real	(No value returned)	ZROTG

Syntax

Language	Syntax
Fortran	CALL SROTG | DROTG | CROTG | ZROTG (*a*, *b*, *c*, *s*)
C and C++	srotg | drotg | crotg | zrotg (*a*, *b*, *c*, *s*);
CBLAS	cblas_srotg | cblas_drotg | cblas_crotg | cblas_zrotg (*a*, *b*, *c*, *s*);

On Entry

a

is the rotational elimination parameter *a*.

Specified as: a number of the data type indicated in Table 73 on page 244.

b

is the rotational elimination parameter *b*.

Specified as: a number of the data type indicated in Table 73 on page 244.

c

See On Return.

s

See On Return.

On Return

a

is the value computed for *r*.

For SROTG and DROTG:

$$r = \sigma \sqrt{a^2 + b^2}$$

where:

σ = SIGN(a) if $|a| > |b|$
σ = SIGN(b) if $|a| \le |b|$

For CROTG and ZROTG:

$$r = \psi \sqrt{|a|^2 + |b|^2} \qquad \text{if } |a| \ne 0$$

$$r = b \qquad \text{if } |a| = 0$$

where:

$\psi = a/|a|$

Returned as: a number of the data type indicated in Table 73 on page 244.

b

is the value computed for *z*.

For SROTG and DROTG:

$z = s$ if $|a| > |b|$
$z = 1/c$ if $|a| \le |b|$ and $c \ne 0$ and $r \ne 0$
$z = 1$ if $|a| \le |b|$ and $c = 0$ and $r \ne 0$
$z = 0$ if $r = 0$

For CROTG and ZROTG: no value is returned, and the input value is not changed.

Returned as: a number of the data type indicated in Table 73 on page 244.

c

is the cosine *c* of the angle of (Givens) rotation. For SROTG and DROTG:

$c = a/r$ if $r \ne 0$
$c = 1$ if $r = 0$

For CROTG and ZROTG:

$$c = \frac{|a|}{\sqrt{|a|^2 + |b|^2}} \qquad \text{if } |a| \ne 0$$

$$c = 0 \qquad \text{if } |a| = 0$$

Returned as: a number of the data type indicated in Table 73 on page 244.

s

is the sine *s* of the angle of (Givens) rotation.

For SROTG and DROTG:

$s = b/r$ if $r \neq 0$
$s = 0$ if $r = 0$

For CROTG and ZROTG:

$$s = \frac{\psi \overline{b}}{\sqrt{|a|^2 + |b|^2}} \quad \text{if } |a| \neq 0$$

$$s = (1.0, 0.0) \quad \text{if } |a| = 0$$

where $\psi = a/|a|$

Returned as: a number of the data type indicated in Table 73 on page 244.

Notes

1. In your C program, arguments *a*, *b*, *c*, and *s* must be passed by reference.
2. In your C++ program, for cblas_srotg and cblas_drotg, arguments *a*, *b*, *c*, and *s* must be passed by pointer.

Function

SROTG and DROTG

A real Givens plane rotation is constructed for values *a* and *b* by computing values for *r, c, s*, and *z*, where:

$$r = \sigma \sqrt{a^2 + b^2}$$

where:

$\sigma = \text{SIGN}(a)$ if $|a| > |b|$
$\sigma = \text{SIGN}(b)$ if $|a| \leq |b|$

$c = a/r$ if $r \neq 0$

$c = 1$ if $r = 0$

$s = b/r$ if $r \neq 0$

$s = 0$ if $r = 0$

$z = s$ if $|a| > |b|$

$z = 1/c$ if $|a| \leq |b|$ and $c \neq 0$ and $r \neq 0$

$z = 1$ if $|a| \leq |b|$ and $c = 0$ and $r \neq 0$

$z = 0$ if $r = 0$

See reference [“95” on page 1258].

Following are some important points about the computation:

1. The numbers for *c*, *s*, and *r* satisfy:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

2. Where necessary, scaling is used to avoid overflow and destructive underflow in the computation of *r*, which is expressed as follows:

$$r = \sigma(|a|+|b|)\sqrt{\left(\frac{a}{|a|+|b|}\right)^2 + \left(\frac{b}{|a|+|b|}\right)^2}$$

3. σ is not essential to the computation of a Givens rotation matrix, but its use permits later stable reconstruction of *c* and *s* from just one stored number, *z*. See reference ["112" on page 1259]. *c* and *s* are reconstructed from *z* as follows:

$$\text{For } z = 1, c = 0 \text{ and } s = 1$$
$$\text{For } |z| < 1, c = \sqrt{1-z^2} \text{ and } s = z$$
$$\text{For } |z| > 1, c = 1/z \text{ and } s = \sqrt{1-c^2}$$

A complex Givens plane rotation is constructed for values *a* and *b* by computing values for *r, c*, and *s*, where:

$$r = \psi\sqrt{|a|^2+|b|^2} \qquad \text{if } |a| \neq 0$$

$$r = b \qquad \text{if } |a| = 0$$

where:

ψ = *a*/|*a*|

$$c = \frac{|a|}{\sqrt{|a|^2+|b|^2}} \qquad \text{if } |a| \neq 0$$

$$c = 0 \qquad \text{if } |a| = 0$$

$$s = \frac{\psi \bar{b}}{\sqrt{|a|^2 + |b|^2}} \quad \text{if } |a| \neq 0$$

$$s = (1.0,0.0) \quad \text{if } |a| = 0$$

See reference [“95” on page 1258].

Following are some important points about the computation:

1. The numbers for *c, s*, and *r* satisfy:

$$\begin{bmatrix} c & s \\ -\bar{s} & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

2. Where necessary, scaling is used to avoid overflow and destructive underflow in the computation of *r*, which is expressed as follows:

$$r = \psi \left(|a| + |b|\right) \sqrt{\left|\frac{a}{|a| + |b|}\right|^2 + \left|\frac{b}{|a| + |b|}\right|^2}$$

Error conditions

Computational Errors
None

Input-Argument Errors
None

Examples

Example 1
This example shows the construction of a real Givens plane rotation, where *r* is 0.

Call Statement and Input:

```
              A     B    C   S
              |     |    |   |
CALL SROTG( 0.0 , 0.0 , C , S )
```

Output:

```
A        =  0.0
B        =  0.0
C        =  1.0
S        =  0.0
```

Example 2
This example shows the construction of a real Givens plane rotation, where *c* is 0.

Call Statement and Input:

```
             A     B    C   S
             |     |    |   |
CALL SROTG( 0.0 , 2.0 , C , S )
```

Output:

```
A        =  2.0
B        =  1.0
C        =  0.0
S        =  1.0
```

Example 3

This example shows the construction of a real Givens plane rotation, where |*b*| > |*a*|.

Call Statement and Input:

```
             A      B    C   S
             |      |    |   |
CALL SROTG( 6.0 , -8.0 , C , S )
```

Output:

```
A        =  -10.0

B        =  -1.666̅
C        =  -0.6
S        =  0.8
```

Example 4

This example shows the construction of a real Givens plane rotation, where |*a*| > |*b*|.

Call Statement and Input:

```
             A     B    C   S
             |     |    |   |
CALL SROTG( 8.0 , 6.0 , C , S )
```

Output:

```
A        =  10.0
B        =  0.6
C        =  0.8
S        =  0.6
```

Example 5

This example shows the construction of a complex Givens plane rotation, where |*a*| = 0.

Call Statement and Input:

```
            A   B   C   S
            |   |   |   |
CALL CROTG( A , B , C , S )

A        =  (0.0, 0.0)
B        =  (1.0, 0.0)
```

Output:

```
A        =  (1.0, 0.0)
C        =  0.0
S        =  (1.0, 0.0)
```

Example 6

This example shows the construction of a complex Givens plane rotation, where |*a*| ≠ 0.

Call Statement and Input:

```
             A   B   C   S
             |   |   |   |
CALL CROTG( A , B , C , S )

A        =  (3.0, 4.0)
B        =  (4.0, 6.0)
```

Output:

```
A        =  (5.26, 7.02)
C        =  0.57
S        =  (0.82, -0.05)
```

SROT, DROT, CROT, ZROT, CSROT, and ZDROT (Apply a Plane Rotation)

Purpose

SROT and DROT apply a real plane rotation to real vectors; CROT and ZROT apply a complex plane rotation to complex vectors; and CSROT and ZDROT apply a real plane rotation to complex vectors. The plane rotation is applied to *n* points, where the points to be rotated are contained in vectors ***x*** and ***y***, and where the cosine and sine of the angle of rotation are *c* and *s*, respectively.

Table 74: Data Types.

Data Types			
x, ***y***	*c*	*s*	**Subprogram**
Short-precision real	Short-precision real	Short-precision real	SROT
Long-precision real	Long-precision real	Long-precision real	DROT
Short-precision complex	Short-precision real	Short-precision complex	CROT
Long-precision complex	Long-precision real	Long-precision complex	ZROT
Short-precision complex	Short-precision real	Short-precision real	CSROT
Long-precision complex	Long-precision real	Long-precision real	ZDROT

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SROT | DROT | CROT | ZROT | CSROT | ZDROT (*n*, *x*, *incx*, *y*, *incy*, *c*, *s*)
C and C++	srot | drot | crot | zrot | csrot | zdrot (*n*, *x*, *incx*, *y*, *incy*, *c*, *s*);
CBLAS	cblas_srot | cblas_drot | cblas_crot | cblas_zrot | cblas_csrot | cblas_zdrot (*n*, *x*, *incx*, *y*, *incy*, *c*, *s*);

On Entry

n

is the number of points to be rotated—that is, the number of elements in vectors ***x*** and ***y***.

Specified as: an integer; $n \geq 0$.

x

is the vector ***x*** of length *n*, containing the x_i coordinates of the points to be rotated.

Specified as: a one-dimensional array of (at least) length 1+(n-1)|*incx*|, containing numbers of the data type indicated in Table 74 on page 250.

incx
is the stride for vector ***x***.

Specified as: an integer. It can have any value.

y
is the vector ***y*** of length n, containing the y_i coordinates of the points to be rotated.

Specified as: a one-dimensional array of (at least) length 1+(n-1)|*incy*|, containing numbers of the data type indicated in Table 74 on page 250.

incy
is the stride for vector ***y***.

Specified as: an integer. It can have any value.

c
the cosine, c, of the angle of rotation.

Specified as: a number of the data type indicated in Table 74 on page 250.

s
the sine, s, of the angle of rotation.

Specified as: a number of the data type indicated in Table 74 on page 250.

On Return

x
is the vector ***x*** of length n, containing the rotated x_i coordinates, where:

$x_i \leftarrow cx_i+sy_i$ for $i = 1,$

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 74 on page 250.

y
is the vector ***y*** of length n, containing the rotated y_i coordinates, where:

For SROT, DROT, CSROT, and ZDROT:

$y_i \leftarrow -sx_i+cy_i$ for $i = 1, n$

For CROT and ZROT:

$$y_i \leftarrow -\bar{s}x_i + cy_i \qquad \text{for } i = 1, n$$

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 74 on page 250.

Notes

The vectors ***x*** and ***y*** must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

Applying a plane rotation to n points, where the points to be rotated are contained in vectors ***x*** and ***y***, is expressed as follows, where c and s are the cosine and sine of the angle of rotation, respectively. For SROT, DROT, CSROT, and ZDROT:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \quad \text{for } i = 1, n$$

For CROT and ZROT:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} c & s \\ -\bar{s} & c \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \quad \text{for } i = 1, n$$

See references [“70” on page 1257] and [“95” on page 1258]. No computation is performed if *n* is 0 or if *c* is 1.0 and *s* is zero. For SROT, CROT, and CSROT, intermediate results are accumulated in long precision when the VSX unit is not used.

Error conditions

Computational Errors
None

Input-Argument Errors
n < 0

Examples

Example 1
This example shows how to apply a real plane rotation to real vectors ***x*** and ***y*** having positive strides.

Call Statement and Input:

```
            N   X  INCX  Y  INCY   C    S
            |   |   |    |   |     |    |
CALL SROT( 5 , X , 1  , Y , 2  , 0.5 , S )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (-1.0, . , -2.0, . , -3.0, . , -4.0, . , -5.0)
```

$$S = \frac{\sqrt{3.0}}{2.0}$$

Output:

```
X        =  (-0.366, -0.732, -1.098, -1.464, -1.830)
Y        =  (-1.366, -2.732, -4.098, -5.464, -6.830)
```

Example 2
This example shows how to apply a real plane rotation to real vectors ***x*** and ***y*** having strides of opposite sign.

Call Statement and Input:

```
            N   X  INCX  Y   INCY   C    S
            |   |   |    |    |     |    |
CALL SROT( 5 , X , 1  , Y , -1  , 0.5 , S )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (-5.0, -4.0, -3.0, -2.0, -1.0)
```

$$S = \frac{\sqrt{3.0}}{2.0}$$

Output:

X =(same as output X in Example 1)
Y = (-6.830, -5.464, -4.098, -2.732, -1.366)

Example 3

This example shows how scalar values in vectors ***x*** and ***y*** can be processed by specifying 0 strides and the number of elements to be processed, *n*, equal to 1.

Call Statement and Input:

```
           N   X  INCX  Y  INCY   C    S
           |   |   |    |   |     |    |
CALL SROT( 1 , X , 0  , Y , 0  , 0.5 , S )

X        =  (1.0)
Y        =  (-1.0)
```

$$S = \frac{\sqrt{3.0}}{2.0}$$

Output:

```
X        =  (-0.366)
Y        =  (-1.366)
```

Example 4

This example shows how to apply a complex plane rotation to complex vectors ***x*** and ***y*** having positive strides.

Call Statement and Input:

```
           N   X  INCX  Y  INCY   C    S
           |   |   |    |   |     |    |
CALL CROT( 3 , X , 1  , Y , 2  , 0.5 , S )

X        =  ((1.0, 2.0), (2.0, 3.0), (3.0, 4.0))
Y        =  ((-1.0, 5.0), . , (-2.0, 4.0), . , (-3.0, 3.0))
S        =  (0.75, 0.50)
```

Output:

```
X        =  ((-2.750, 4.250), (-2.500, 3.500), (-2.250, 2.750))
Y        =  ((-2.250, 1.500), . , (-4.000, 0.750), . ,
             (-5.750, 0.000))
```

Example 5

This example shows how to apply a real plane rotation to complex vectors ***x*** and ***y*** having positive strides.

Call Statement and Input:

```
            N   X  INCX  Y  INCY   C    S
            |   |   |    |   |     |    |
CALL CSROT( 3 , X , 1  , Y , 2  , 0.5 , S )

X        =  ((1.0, 2.0), (2.0, 3.0), (3.0, 4.0))
Y        =  ((-1.0, 5.0), . , (-2.0, 4.0), . , (-3.0, 3.0))
```

$$S = \frac{\sqrt{3.0}}{2.0}$$

Output:

```
X        =  ((-0.366, 5.330), (-0.732, 4.964), (-1.098, 4.598))
Y        =  ((-1.366, 0.768), . , (-2.732, -0.598), . ,
             (-4.098, -1.964))
```

SROTMG and DROTMG (Construct a modified Givens Transformation)

Purpose

SROTMG and DROTMG construct the modified Givens transformation matrix ***H***, which zeros the second component of the 2-vector:

$$\begin{bmatrix} \sqrt{d_1}x_1 \\ \sqrt{d_2}x_2 \end{bmatrix}$$

Figure 11: 2-Vector

H has one of the following forms:

$$\begin{array}{lll}
\textbf{PARAM}_1 & & \\
-1.0 & H = & \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \\
0.0 & H = & \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix} \\
1.0 & H = & \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix} \\
-2.0 & H = & \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}
\end{array}$$

Figure 12: The form of matrix ***H***

SROTM and DROTM uses the information in PARAM to apply the modified Givens transformation.

Table 75: Data Types.

d1, d2, x1, x2	**Subprogram**
Short-precision real	SROTMG
Long-precision real	DROTMG

Syntax

Table 76: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SROTMG | DROTMG | (*d1*, *d2*, *x1*, *x2*, *param*)
C and C++	srotmg | drotmg (*d1*, *d2*, *x1*, *x2*, *param*);
CBLAS	cblas_srotmg | cblas_drotmg | (*d1*, *d2*, *x1*, *x2*, *param*);

On Entry

d1

is the value d_1 shown in Figure 11 on page 254.

Specified as: a number of the data type indicated in Table 75 on page 254.

d2

is the value d_2 shown in Figure 11 on page 254.

Specified as: a number of the data type indicated in Table 75 on page 254.

x1

is the value x_1 shown in Figure 11 on page 254.

Specified as: a number of the data type indicated in Table 75 on page 254.

x2

is the value x_2 shown in Figure 11 on page 254.

Specified as: a number of the data type indicated in Table 75 on page 254.

On Return

d1

d1 is overwritten; that is, the original input is not preserved.

Returned as: a number of the data type indicated in Table 75 on page 254.

d2

d2 is overwritten; that is, the original input is not preserved.

Returned as: a number of the data type indicated in Table 75 on page 254.

x1

x1 is overwritten; that is, the original input is not preserved.

Returned as: a number of the data type indicated in Table 75 on page 254.

x2

x2 is overwritten; that is, the original input is not preserved.

Returned as: a number of the data type indicated in Table 75 on page 254.

param

param is a vector of length 5 defining the form and element values of the modified Givens transformation matrix ***H***, where:

$param_1$ = Form of ***H***, either -1.0, 0.0, 1.0, or -2.0, as shown in Figure 12 on page 254.
$param_2$ = H_{11}
$param_3$ = H_{21}
$param_4$= H_{12}
$param_5$= H_{22}

The matrix ***H*** values of 1.0, -1.0, 0.0 implied when $param_1$ = 0, 1, -2 are assumed and are not stored in $param_{2:5}$.

SROTM and DROTM use the information in *param* to apply the modified Givens transformation.

Returned as: a vector of length 5.

Notes:

1. In your C program, arguments *d1*, *d2*, *x1*, and *x2* must be passed by reference.
2. In your C++ program, for cblas_srotmg and cblas_drotmg, arguments *d1*, *d2*, *x1*, and *x2* must be passed by pointer.

Function

SROTMG and DROTMG construct the modified Givens transformation matrix ***H***, which zeros the second component of the 2-vector:

$$\begin{bmatrix} \sqrt{d_1}x_1 \\ \sqrt{d_2}x_2 \end{bmatrix}$$

H has one of the following forms:

$PARAM_1$	
-1.0	$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$
0.0	$H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$
1.0	$H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$
-2.0	$H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$

SROTM and DROTM uses the information in PARAM to apply the modified Givens transformation.

For more information, see references [“95” on page 1258].

Error conditions

Computational Errors
None

Input-Argument Errors
None

Examples

See the combined DROTMG and DROTM example shown in “Example 1” on page 259.

SROTM and DROTM (Apply a modified Givens Transformation)

Purpose

SROTM and DROTM apply a modified GIVENS matrix transformation. For I=1,*n*:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \text{<--}H \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Where H is a modified Givens transformation returned by a preceding call to SROTMG or DROTMG respectively.

For I = 1,*n*:

$$\begin{aligned} x_i &= h_{11}x_i+h_{12}y_i \\ y_i &= h_{21}x_i+h_{22}y_i \end{aligned}$$

Table 77: Data Types.

x, ***y***	**Subprogram**
Short-precision real	SROTM
Long-precision real	DROTM

Syntax

Table 78: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SROTM | DROTM (*n*, *x*, *incx*, *y*, *incy*, *param*)
C and C++	srotm | drotm (*n*, *x*, *incx*, *y*, *incy*, *param*);
CBLAS	cblas_srotm | cblas_drotm (*n*, *x*, *incx*, *y*, *incy*, *param*);

On Entry

n

is the number of elements in vectors ***x*** and ***y***.

Specified as: an integer; *n* ≥ 0.

x

is the vector ***x*** of length *n*.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 77 on page 257.

incx

is the stride for vector ***x***.

Specified as: an integer. It can have any value.

y

is the vector ***y*** of length *n*.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incy*|, containing numbers of the data type indicated in Table 77 on page 257.

incy

is the stride for vector ***y***.

Specified as: an integer. It can have any value.

param

param is a vector of length 5 as returned by a preceding call to SROTMG or DROTMG defining the form and element values of the modified Givens transformation matrix ***H***, where:

$param_1$ = Form of ***H***, either -1.0, 0.0, 1.0, or -2.0, as shown in Figure 12 on page 254.
$param_2$ = H_{11}
$param_3$ = H_{21}
$param_4$= H_{12}
$param_5$= H_{22}

The matrix ***H*** values of 1.0, -1.0, 0.0 implied when $param_1$ = 0, 1, -2 are assumed and do not need to be stored in $param_{2:5}$.

Returned as: a vector of length 5.

On Return

x

is the vector ***x*** of length *n*, where:

$$x_i = h_{11}x_i + h_{12}y_i$$

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 77 on page 257.

y

is the vector ***y*** of length *n*, where:

$$y_i = h_{21}x_i + h_{22}y_i$$

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 77 on page 257.

Notes:

- The vectors ***x*** and ***y*** must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
- If negative stride is specified for a vector, then the position of the first element accessed is (-*n*+1)*inc* + 1, where *n* = vector count and *inc* = vector stride, except when stride of both vectors is -1, in which case both vectors are accessed from position 1. For all other strides, the position of the first element accessed is 1.

Function

SROTM and DROTM apply a modified GIVENS matrix transformation. For I=1,*n*:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} <\!-\!- H \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Where H is a modified Givens transformation matrix returned by a preceding call to SROTMG or DROTMG respectively.

For I = 1,*n*:

```
        $x_i$ = $h_{11}x_i+h_{12}y_i$
        $y_i$ = $h_{21}x_i+h_{22}y_i$
```

For more information, see references [“95” on page 1258].

If *n* = 0 or if ***H*** is the identity matrix ($param_1$ = -2), the subroutine returns immediately.

Error conditions

Computational Errors

None

Input-Argument Errors

n < 0

Examples

Example 1

This example illustrates the construction and application of a modified Givens transformation.

Call Statement and Input:

```
             D1   D2   X1   X2  PARAM
              |    |    |    |     |
CALL DROTMG( D1 , D2 , X1 , X2, PARAM)
```

```
D1 =  1000000000.0
D2 =  1.0
X1 = -2.0
X2 =  4.0
```

Output:

```
PARAM        =  (-1.0, 4096.0, 2.0, -0.8192D-5, 1.0)

On output, D1, D2, X1, and X2 are overwritten.
```

Call Statement and Input:

```
            N   X INCX  Y INCY  PARAM
            |   |   |   |   |     |
CALL DROTM( 5 , X , 1 , Y , 1 , PARAM)


X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (-1.0, -2.0, -3.0, -4.0, -5.0)
PARAM    =  output from the call to DROTMG
```

Output:

```
X        =  (4096.0, 8192.0, 12288.0, 16384.0, 20480.0)
Y        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, and ZDSCAL (Multiply a Vector X by a Scalar and Store in the Vector X)

Purpose

These subprograms perform the following computation, using the scalar α and the vector ***x***:

$\boldsymbol{x} \leftarrow \alpha \boldsymbol{x}$

Table 79: Data Types.		
Data Types		
α	***x***	**Subprogram**
Short-precision real	Short-precision real	SSCAL
Long-precision real	Long-precision real	DSCAL
Short-precision complex	Short-precision complex	CSCAL
Long-precision complex	Long-precision complex	ZSCAL
Short-precision real	Short-precision complex	CSSCAL
Long-precision real	Long-precision complex	ZDSCAL

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SSCAL | DSCAL | CSCAL | ZSCAL | CSSCAL | ZDSCAL (*n*, *alpha*, *x*, *incx*)
C and C++	sscal | dscal | cscal | zscal | csscal | zdscal (*n*, *alpha*, *x*, *incx*);
CBLAS	cblas_sscal | cblas_dscal | cblas_cscal | cblas_zscal | cblas_csscal | cblas_zdscal (*n*, *alpha*, *x*, *incx*);

On Entry

n
is the number of elements in vector ***x***. Specified as: an integer; $n \geq 0$.

alpha
is the scalar α.

Specified as: a number of the data type indicated in Table 79 on page 260.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 79 on page 260.

incx
is the stride for vector ***x***.

Specified as: an integer. It can have any value.

On Return

x
is the vector ***x*** of length *n*, containing the result of the computation α***x***. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 79 on page 260.

Notes

The fastest way in ESSL to zero out contiguous (stride 1) arrays is to call SSCAL or DSCAL, specifying *incx* = 1 and α = 0.

Function

The computation is expressed as follows:

$$\begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} \leftarrow \alpha \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

See reference ["95" on page 1258]. If *n* is 0, no computation is performed. For CSCAL, intermediate results are accumulated in long precision when the VSX unit is not used.

Error conditions

Computational Errors
None

Input-Argument Errors
$n < 0$

Examples

Example 1
This example shows a vector, ***x***, with a stride of 1.

Call Statement and Input:

```
            N  ALPHA  X  INCX
            |    |    |   |
CALL SSCAL( 5 , 2.0 , X , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output:

```
X        =  (2.0, 4.0, 6.0, 8.0, 10.0)
```

Example 2
This example shows vector, ***x***, with a stride greater than 1.

Call Statement and Input:

```
            N  ALPHA  X  INCX
            |    |    |   |
CALL SSCAL( 5 , 2.0 , X , 2  )

X        =  (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0)
```

Output:

```
X        =  (2.0, . , 4.0, . , 6.0, . , 8.0, . , 10.0)
```

Example 3
This example illustrates that when the strides for two similar computations (Example 1 and Example 3) have the same absolute value but have opposite signs, the output is the same. This example is the same as Example 1, except the stride for ***x*** is negative (-1). For performance reasons, it is better to specify the positive stride. For ***x***, processing begins at element `X(5)`, which is 5.0, and results are stored beginning at the same element.

Call Statement and Input:

```
            N  ALPHA  X   INCX
            |    |    |    |
CALL SSCAL( 5 , 2.0 , X , -1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output:

```
X        =  (2.0, 4.0, 6.0, 8.0, 10.0)
```

Example 4
This example shows how SSCAL can be used to compute a scalar value. In this case, input vector ***x*** contains a scalar value, and the stride is 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input:

```
            N  ALPHA  X  INCX
            |    |    |   |
CALL SSCAL( 1 , 2.0 , X , 0  )

X        =  (1.0)
```

Output:

```
X        =  (2.0)
```

Example 5
This example shows a scalar, α, and a vector, ***x***, containing complex numbers, where vector ***x*** has a stride of 1.

Call Statement and Input:

```
            N  ALPHA  X  INCX
            |    |    |   |
CALL CSCAL( 3 ,ALPHA, X , 1  )

ALPHA    =  (2.0, 3.0)
X        =  ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
```

Output:

```
X        =  ((-4.0, 7.0), (4.0, 6.0), (-9.0, 19.0))
```

Example 6
This example shows a scalar, α, containing a real number, and a vector, ***x***, containing complex numbers, where vector ***x*** has a stride of 1.

Call Statement and Input:

```
             N  ALPHA  X  INCX
             |    |    |   |
CALL CSSCAL( 3 , 2.0 , X , 1  )

X        =  ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
```

Output:

```
X        =  ((2.0, 4.0), (4.0, 0.0), (6.0, 10.0))
```

SSWAP, DSWAP, CSWAP, and ZSWAP (Interchange the Elements of Two Vectors)

Purpose

These subprograms interchange the elements of vectors ***x*** and ***y***:

$\boldsymbol{y} \leftarrow\rightarrow \boldsymbol{x}$

Table 80: Data Types.	
Data Types	
x, ***y***	**Subprogram**
Short-precision real	SSWAP
Long-precision real	DSWAP
Short-precision complex	CSWAP
Long-precision complex	ZSWAP

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SSWAP | DSWAP | CSWAP | ZSWAP (*n*, *x*, *incx*, *y*, *incy*)
C and C++	sswap | dswap | cswap | zswap (*n*, *x*, *incx*, *y*, *incy*);
CBLAS	cblas_sswap | cblas_dswap | cblas_cswap | cblas_zswap (*n*, *x*, *incx*, *y*, *incy*);

On Entry

n

is the number of elements in vectors ***x*** and ***y***.

Specified as: an integer; $n \geq 0$

.

x

is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 80 on page 263.

incx

is the stride for vector ***x***.

Specified as: an integer. It can have any value.

y

is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incy*|, containing numbers of the data type indicated in Table 80 on page 263.

incy

is the stride for vector ***y***.

Specified as: an integer. It can have any value.

On Return

x

is the vector ***x*** of length *n*, containing the elements that were swapped from vector ***y***. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 80 on page 263.

y

is the vector ***y*** of length *n*, containing the elements that were swapped from vector ***x***. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 80 on page 263.

Notes

1. If you specify the same vector for ***x*** and ***y***, then *incx* and *incy* must be equal; otherwise, results are unpredictable.
2. If you specify different vectors for ***x*** and ***y***, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

The elements of vectors ***x*** and ***y*** are interchanged as follows:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} \leftarrow \rightarrow \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

See reference ["95" on page 1258]. If *n* is 0, no elements are interchanged.

Error conditions

Computational Errors
None

Input-Argument Errors
$n < 0$

Examples

Example 1

This example shows vectors ***x*** and ***y*** with positive strides.

Call Statement and Input:

```
            N   X  INCX  Y  INCY
            |   |   |    |   |
CALL SSWAP( 5 , X , 1  , Y , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (-1.0, . , -2.0, . , -3.0, . , -4.0, . , -5.0)
```

Output:

```
X        =  (-1.0, -2.0, -3.0, -4.0, -5.0)
Y        =  (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0)
```

Example 2

This example shows how to obtain output vectors ***x*** and ***y*** that are reverse copies of the input vectors ***y*** and ***x***. You must specify strides with the same absolute value, but with opposite signs. For ***y***, which has negative stride, processing begins at element `Y(5)`, which is -5.0, and the results of the swap are stored beginning at the same element.

Call Statement and Input:

```
            N   X  INCX  Y   INCY
            |   |   |    |    |
CALL SSWAP( 5 , X , 1  , Y , -1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (-1.0, -2.0, -3.0, -4.0, -5.0)
```

Output:

```
X        =  (-5.0, -4.0, -3.0, -2.0, -1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Example 3

This example shows how SSWAP can be used to interchange scalar values in vectors ***x*** and ***y*** by specifying 0 strides and the number of elements to be processed as 1.

Call Statement and Input:

```
            N   X  INCX  Y  INCY
            |   |   |    |   |
CALL SSWAP( 1 , X , 0  , Y , 0  )

X        =  (1.0)
Y        =  (-4.0)
```

Output

```
X        =  (-4.0)
Y        =  (1.0)
```

Example 4

This example shows vectors ***x*** and ***y***, containing complex numbers and having positive strides.

Call Statement and Input:

```
            N   X  INCX  Y  INCY
            |   |   |    |   |
CALL CSWAP( 4 , X , 1  , Y , 2  )

X        =  ((1.0, 6.0), (2.0, 7.0), (3.0, 8.0), (4.0, 9.0))
Y        =  ((-1.0, -1.0), . , (-2.0, -2.0), . , (-3.0, -3.0), . ,
             (-4.0, -4.0))
```

Output:

```
X        =  ((-1.0, -1.0), (-2.0, -2.0), (-3.0, -3.0), (-4.0, -4.0))
Y        =  ((1.0, 6.0), . , (2.0, 7.0), . , (3.0, 8.0), . ,
             (4.0, 9.0))
```

SVEA, DVEA, CVEA, and ZVEA (Add a Vector X to a Vector Y and Store in a Vector Z)

Purpose

These subprograms perform the following computation, using vectors ***x, y,*** and ***z***:

$\boldsymbol{z} \leftarrow \boldsymbol{x} + \boldsymbol{y}$

Table 81: Data Types.

Data Types	
x, ***y***, ***z***	**Subprogram**
Short-precision real	SVEA
Long-precision real	DVEA
Short-precision complex	CVEA
Long-precision complex	ZVEA

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SVEA | DVEA | CVEA | ZVEA (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)
C and C++	svea | dvea | cvea | zvea (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*);
CBLAS	cblas_svea | cblas_dvea | cblas_cvea | cblas_zvea (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*);

On Entry

n

is the number of elements in vectors ***x, y,*** and ***z***.

Specified as: an integer; $n \geq 0$.

x

is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 81 on page 265.

incx

is the stride for vector ***x***.

Specified as: an integer. It can have any value.

y

is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incy*|, containing numbers of the data type indicated in Table 81 on page 265.

incy

is the stride for vector ***y***.

Specified as: an integer. It can have any value.

z

See On Return.

incz

is the stride for vector ***z***.

Specified as: an integer. It can have any value.

On Return

z

is the vector ***z*** of length *n*, containing the result of the computation. Returned as: a one-dimensional array of (at least) length 1+(*n*-1)|*incz*|, containing numbers of the data type indicated in Table 81 on page 265.

Notes

1. If you specify the same vector for ***x*** and ***z***, then *incx* and *incz* must be equal; otherwise, results are unpredictable. The same is true for ***y*** and ***z***.
2. If you specify different vectors for ***x*** and ***z***, they must have no common elements; otherwise, results are unpredictable. The same is true for ***y*** and ***z***. See "Vector concepts" on page 71.

Function

The computation is expressed as follows:

$$\begin{bmatrix} z_1 \\ \cdot \\ \cdot \\ \cdot \\ z_n \end{bmatrix} \leftarrow \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix}$$

If *n* is 0, no computation is performed.

Error conditions

Computational Errors
None

Input-Argument Errors
$n < 0$

Examples

Example 1

This example shows vectors ***x***, ***y***, and ***z***, with positive strides.

Call Statement and Input:

```
            N   X  INCX  Y  INCY  Z  INCZ
            |   |   |    |   |    |   |
CALL SVEA( 5 , X , 1  , Y , 2  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0)
```

Output:

```
Z        =  (2.0, 3.0, 4.0, 5.0, 6.0)
```

Example 2

This example shows vectors ***x*** and ***y*** having strides of opposite sign, and an output vector ***z*** having a positive stride. For ***y***, which has negative stride, processing begins at element Y(5), which is 1.0.

Call Statement and Input:

```
            N   X  INCX  Y   INCY  Z  INCZ
            |   |   |    |    |    |   |
CALL SVEA( 5 , X , 1  , Y , -1  , Z , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
Z        =  (2.0, . , 4.0, . , 6.0, . , 8.0, . , 10.0)
```

Example 3

This example shows a vector, ***x***, with 0 stride and a vector, ***z***, with negative stride. ***x*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***x***. For vector ***z***, results are stored beginning in element Z(5).

Call Statement and Input:

```
            N   X  INCX  Y  INCY  Z   INCZ
            |   |   |    |   |    |    |
 CALL SVEA( 5 , X , 0  , Y , 1  , Z , -1  )

 X        =  (1.0)
 Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
Z        =  (2.0, 3.0, 4.0, 5.0, 6.0)
```

Example 4

This example shows a vector, ***y***, with 0 stride. ***y*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***y***.

Call Statement and Input:

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVEA( 5 , X , 1  , Y , 0  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output:

```
Z        =  (6.0, 7.0, 8.0, 9.0, 10.0)
```

Example 5

This example shows the output vector, ***z***, with 0 stride, where the vector ***x*** has positive stride, and the vector ***y*** has 0 stride. The number of elements to be processed, *n*, is greater than 1.

Call Statement and Input:

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVEA( 5 , X , 1  , Y , 0  , Z , 0  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output:

```
Z        =  (10.0)
```

Example 6

This example shows the output vector ***z***, with 0 stride, where the vector ***x*** has 0 stride, and the vector ***y*** has negative stride. The number of elements to be processed, *n*, is greater than 1.

Call Statement and Input:

```
           N   X  INCX  Y   INCY  Z  INCZ
           |   |   |    |    |    |   |
CALL SVEA( 5 , X , 0  , Y , -1  , Z , 0  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
Z        =  (6.0)
```

Example 7

This example shows how SVEA can be used to compute a scalar value. In this case, vectors ***x*** and ***y*** contain scalar values. The strides of all vectors, ***x***, ***y***, and ***z***, are 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input:

```
            N   X  INCX  Y  INCY  Z  INCZ
            |   |   |    |   |    |   |
CALL SVEA( 1 , X , 0  , Y , 0  , Z , 0  )

X        =  (1.0)
Y        =  (5.0)
```

Output:

```
Z        =  (6.0)
```

Example 8

This example shows vectors ***x*** and ***y***, containing complex numbers and having positive strides.

Call Statement and Input:

```
            N   X  INCX  Y  INCY  Z  INCZ
            |   |   |    |   |    |   |
CALL CVEA( 3 , X , 1  , Y , 2  , Z , 1  )

X        =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
Y        =  ((7.0, 8.0), . , (9.0, 10.0), . , (11.0, 12.0))
```

Output:

```
Z        =  ((8.0, 10.0), (12.0, 14.0), (16.0, 18.0))
```

SVES, DVES, CVES, and ZVES (Subtract a Vector Y from a Vector X and Store in a Vector Z)

Purpose

These subprograms perform the following computation, using vectors ***x, y,*** and ***z***:

z←x-y

Table 82: Data Types.

Data Types	
x, ***y***, ***z***	**Subprogram**
Short-precision real	SVES
Long-precision real	DVES
Short-precision complex	CVES
Long-precision complex	ZVES

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	**Syntax**
Fortran	CALL SVES | DVES | CVES | ZVES (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)
C and C++	sves | dves | cves | zves (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*);
CBLAS	cblas_sves | cblas_dves | cblas_cves | cblas_zves (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*);

On Entry

n

is the number of elements in vectors ***x, y,*** and ***z***.

Specified as: an integer; $n \geq 0$.

x

is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 82 on page 269.

incx

is the stride for vector ***x***.

Specified as: an integer. It can have any value.

y

is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incy*|, containing numbers of the data type indicated in Table 82 on page 269.

incy

is the stride for vector ***y***.

Specified as: an integer. It can have any value.

z

See On Return.

incz

is the stride for vector ***z***.

Specified as: an integer. It can have any value.

On Return

z

is the vector ***z*** of length *n*, containing the result of the computation. Returned as: a one-dimensional array of (at least) length 1+(*n*-1)|*incz*|, containing numbers of the data type indicated in Table 82 on page 269.

Notes

1. If you specify the same vector for ***x*** and ***z***, then *incx* and *incz* must be equal; otherwise, results are unpredictable. The same is true for ***y*** and ***z***.
2. If you specify different vectors for ***x*** and ***z***, they must have no common elements; otherwise, results are unpredictable. The same is true for ***y*** and ***z***. See "Vector concepts" on page 71.

Function

The computation is expressed as follows:

$$\begin{bmatrix} z_1 \\ \cdot \\ \cdot \\ \cdot \\ z_n \end{bmatrix} \leftarrow \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} - \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix}$$

If *n* is 0, no computation is performed.

Error conditions

Computational Errors

None

Input-Argument Errors

n < 0

Examples

Example 1

This example shows vectors ***x***, ***y***, and ***z***, with positive strides.

Call Statement and Input:

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVES( 5 , X , 1  , Y , 2  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0)
```

Output:

```
Z        =  (0.0, 1.0, 2.0, 3.0, 4.0)
```

Example 2

This example shows vectors ***x*** and ***y*** having strides of opposite sign, and an output vector ***z*** having a positive stride. For ***y***, which has negative stride, processing begins at element Y(5), which is 1.0.

Call Statement and Input:

```
           N   X  INCX  Y   INCY  Z  INCZ
           |   |   |    |    |    |   |
CALL SVES( 5 , X , 1  , Y , -1  , Z , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
Z        =  (0.0, . , 0.0, . , 0.0, . , 0.0, . , 0.0)
```

Example 3

This example shows a vector, ***x***, with 0 stride, and a vector, ***z***, with negative stride. ***x*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***x***. For vector ***z***, results are stored beginning in element Z(5).

Call Statement and Input:

```
           N   X  INCX  Y  INCY  Z   INCZ
           |   |   |    |   |    |    |
CALL SVES( 5 , X , 0  , Y , 1  , Z , -1  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
Z        =  (0.0, -1.0, -2.0, -3.0, -4.0)
```

Example 4

This example shows a vector, ***y***, with 0 stride. ***y*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***y***.

Call Statement and Input:

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVES( 5 , X , 1  , Y , 0  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output:

```
Z        =  (-4.0, -3.0, -2.0, -1.0, 0.0)
```

Example 5

This example shows the output vector ***z***, with 0 stride, where the vector ***x*** has positive stride, and the vector ***y*** has 0 stride. The number of elements to be processed, *n*, is greater than 1.

Call Statement and Input:

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVES( 5 , X , 1  , Y , 0  , Z , 0  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output:

```
Z        =  (0.0)
```

Example 6

This example shows the output vector ***z***, with 0 stride, where the vector ***x*** has 0 stride, and the vector ***y*** has negative stride. The number of elements to be processed, *n*, is greater than 1.

Call Statement and Input:

```
           N   X  INCX  Y   INCY  Z  INCZ
           |   |   |    |    |    |   |
CALL SVES( 5 , X , 0  , Y , -1  , Z , 0  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
Z        =  (-4.0)
```

Example 7

This example shows how SVES can be used to compute a scalar value. In this case, vectors ***x*** and ***y*** contain scalar values. The strides of all vectors, ***x***, ***y***, and ***z***, are 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input:

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVES( 1 , X , 0  , Y , 0  , Z , 0  )

X        =  (1.0)
Y        =  (5.0)
```

Output:

```
Z        =  (-4.0)
```

Example 8

This example shows vectors ***x*** and ***y***, containing complex numbers and having positive strides.

Call Statement and Input:

```
            N   X  INCX  Y  INCY  Z  INCZ
            |   |   |    |   |    |   |
CALL CVES( 3 , X , 1  , Y , 2  , Z , 1  )

X         =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
Y         =  ((7.0, 8.0), . , (9.0, 10.0), . , (11.0, 12.0))
```

Output:

```
Z         =  ((-6.0, -6.0), (-6.0, -6.0), (-6.0, -6.0))
```

SVEM, DVEM, CVEM, and ZVEM (Multiply a Vector X by a Vector Y and Store in a Vector Z)

Purpose

These subprograms perform the following computation, using vectors ***x, y,*** and ***z***:

z←***xy***

Table 83: Data Types.

Data Types	
x, ***y***, ***z***	**Subprogram**
Short-precision real	SVEM
Long-precision real	DVEM
Short-precision complex	CVEM
Long-precision complex	ZVEM

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	**Syntax**
Fortran	CALL SVEM | DVEM | CVEM | ZVEM (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)
C and C++	svem | dvem | cvem | zvem (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*);
CBLAS	cblas_svem | cblas_dvem | cblas_cvem | cblas_zvem (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*);

On Entry

n

is the number of elements in vectors ***x, y,*** and ***z***.

Specified as: an integer; $n \geq 0$.

x

is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 83 on page 273.

incx

is the stride for vector ***x***.

Specified as: an integer. It can have any value.

y
is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incy*|, containing numbers of the data type indicated in Table 83 on page 273.

incy
is the stride for vector ***y***.

Specified as: an integer. It can have any value.

z
See On Return.

incz
is the stride for vector ***z***.

Specified as: an integer. It can have any value.

On Return

z
is the vector ***z*** of length *n*, containing the result of the computation. Returned as: a one-dimensional array of (at least) length 1+(*n*-1)|*incz*|, containing numbers of the data type indicated in Table 83 on page 273.

Notes

1. If you specify the same vector for ***x*** and ***z***, then *incx* and *incz* must be equal; otherwise, results are unpredictable. The same is true for ***y*** and ***z***.
2. If you specify different vectors for ***x*** and ***z***, they must have no common elements; otherwise, results are unpredictable. The same is true for ***y*** and ***z***. See "Vector concepts" on page 71.

Function

The computation is expressed as follows:

$z_i \leftarrow x_i y_i \quad \text{for } i = 1, n$

If *n* is 0, no computation is performed. For CVEM, intermediate results are accumulated in long precision when the VSX unit is not used.

Error conditions

Computational Errors
None

Input-Argument Errors
$n < 0$

Examples

Example 1
This example shows vectors ***x***, ***y***, and ***z***, with positive strides.

Call Statement and Input:

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVEM( 5 , X , 1  , Y , 2  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0)
```

Output:

```
Z        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Example 2

This example shows vectors ***x*** and ***y*** having strides of opposite sign, and an output vector ***z*** having a positive stride. For ***y***, which has negative stride, processing begins at element Y(5), which is 1.0.

Call Statement and Input:

```
           N   X  INCX  Y   INCY  Z  INCZ
           |   |   |    |    |    |   |
CALL SVEM( 5 , X , 1  , Y , -1  , Z , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
Z        =  (1.0, . , 4.0, . , 9.0, . , 16.0, . , 25.0)
```

Example 3

This example shows a vector, ***x***, with 0 stride, and a vector, ***z***, with negative stride. ***x*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***x***. For vector ***z***, results are stored beginning in element Z(5).

Call Statement and Input:

```
           N   X  INCX  Y  INCY  Z   INCZ
           |   |   |    |   |    |    |
CALL SVEM( 5 , X , 0  , Y , 1  , Z , -1  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
Z        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Example 4

This example shows a vector, ***y***, with 0 stride. ***y*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***y***.

Call Statement and Input:

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVEM( 5 , X , 1  , Y , 0  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output:

```
Z        =  (5.0, 10.0, 15.0, 20.0, 25.0)
```

Example 5

This example shows the output vector, ***z***, with 0 stride, where the vector ***x*** has positive stride, and the vector ***y*** has 0 stride. The number of elements to be processed, *n*, is greater than 1.

Call Statement and Input:

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVEM( 5 , X , 1  , Y , 0  , Z , 0  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output:

```
Z        =  (25.0)
```

Example 6

This example shows the output vector ***z***, with 0 stride, where the vector ***x*** has 0 stride, and the vector ***y*** has negative stride. The number of elements to be processed, *n*, is greater than 1.

Call Statement and Input:

```
            N   X  INCX  Y   INCY  Z  INCZ
            |   |   |    |    |    |   |
CALL SVEM( 5 , X , 0  , Y , -1  , Z , 0  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
Z        =  (5.0)
```

Example 7

This example shows how SVEM can be used to compute a scalar value. In this case, vectors ***x*** and ***y*** contain scalar values. The strides of all vectors, ***x***, ***y***, and ***z***, are 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input:

```
            N   X  INCX  Y  INCY  Z  INCZ
            |   |   |    |   |    |   |
CALL SVEM( 1 , X , 0  , Y , 0  , Z , 0  )

X        =  (1.0)
Y        =  (5.0)
```

Output:

```
Z        =  (5.0)
```

Example 8

This example shows vectors ***x*** and ***y***, containing complex numbers and having positive strides.

Call Statement and Input:

```
            N   X  INCX  Y  INCY  Z  INCZ
            |   |   |    |   |    |   |
CALL CVEM( 3 , X , 1  , Y , 2  , Z , 1  )

X        =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
Y        =  ((7.0, 8.0), . , (9.0, 10.0), . , (11.0, 12.0))
```

Output:

```
Z        =  ((-9.0, 22.0), (-13.0, 66.0), (-17.0, 126.0))
```

SYAX, DYAX, CYAX, ZYAX, CSYAX, and ZDYAX (Multiply a Vector X by a Scalar and Store in a Vector Y)

Purpose

These subprograms perform the following computation, using the scalar α and vectors ***x*** and ***y***:

$\boldsymbol{y} \leftarrow \alpha \boldsymbol{x}$

Table 84: Data Types.

Data Types		
α	***x***, ***y***	**Subprogram**
Short-precision real	Short-precision real	SYAX
Long-precision real	Long-precision real	DYAX
Short-precision complex	Short-precision complex	CYAX
Long-precision complex	Long-precision complex	ZYAX
Short-precision real	Short-precision complex	CSYAX
Long-precision real	Long-precision complex	ZDYAX

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	**Syntax**
Fortran	CALL SYAX | DYAX | CYAX | ZYAX | CSYAX | ZDYAX (*n*, *alpha*, *x*, *incx*, *y*, *incy*)
C and C++	syax | dyax | cyax | zyax | csyax | zdyax (*n*, *alpha*, *x*, *incx*, *y*, *incy*);
CBLAS	cblas_syax | cblas_dyax | cblas_cyax | cblas_zyax | cblas_csyax | cblas_zdyax (*n*, *alpha*, *x*, *incx*, *y*, *incy*);

On Entry

n
is the number of elements in vector ***x*** and ***y***.

Specified as: an integer; $n \geq 0$.

alpha
is the scalar α.

Specified as: a number of the data type indicated in Table 84 on page 277.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 84 on page 277.

incx
is the stride for vector ***x***.

Specified as: an integer. It can have any value.

y
See On Return.

incy
is the stride for vector ***y***.

Specified as: an integer. It can have any value.

On Return

y
is the vector ***y*** of length *n*, containing the result of the computation α***x***. Returned as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 84 on page 277.

Notes

1. If you specify the same vector for ***x*** and ***y***, then *incx* and *incy* must be equal; otherwise, results are unpredictable.
2. If you specify different vectors for ***x*** and ***y***, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

The computation is expressed as follows:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} \leftarrow \alpha \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

See reference ["95" on page 1258]. If *n* is 0, no computation is performed. For CYAX, intermediate results are accumulated in long precision when the VSX unit is not used.

Error conditions

Computational Errors
None

Input-Argument Errors
$n < 0$

Examples

Example 1

This example shows vectors ***x*** and ***y*** with positive strides.

Call Statement and Input:

```
          N  ALPHA  X  INCX  Y  INCY
          |    |    |   |    |   |
CALL SYAX( 5 , 2.0 , X , 1  , Y , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output:

```
Y        =  (2.0, . , 4.0, . , 6.0, . , 8.0, . , 10.0)
```

Example 2

This example shows vectors ***x*** and ***y*** that have strides of opposite signs. For ***y***, which has negative stride, results are stored beginning in element Y(5).

Call Statement and Input:

```
          N  ALPHA  X  INCX  Y   INCY
          |    |    |   |    |    |
CALL SYAX( 5 , 2.0 , X , 1  , Y , -1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output:

```
Y        =  (10.0, 8.0, 6.0, 4.0, 2.0)
```

Example 3

This example shows a vector, ***x***, with 0 stride. ***x*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***x***.

Call Statement and Input:

```
            N  ALPHA  X  INCX  Y  INCY
            |    |    |   |    |   |
CALL SYAX( 5 , 2.0 , X , 0  , Y , 1  )

X        =  (1.0)
```

Output:

```
Y        =  (2.0, 2.0, 2.0, 2.0, 2.0)
```

Example 4

This example shows how SYAX can be used to compute a scalar value. In this case both vectors ***x*** and ***y*** contain scalar values, and the strides for both vectors are 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input:

```
            N  ALPHA  X  INCX  Y  INCY
            |    |    |   |    |   |
CALL SYAX( 1 , 2.0 , X , 0  , Y , 0  )

X        =  (1.0)
```

Output:

```
Y        =  (2.0)
```

Example 5

This example shows a scalar, α, and vectors ***x*** and ***y***, containing complex numbers, where both vectors have a stride of 1.

Call Statement and Input:

```
            N  ALPHA  X  INCX  Y  INCY
            |    |    |   |    |   |
CALL CYAX( 3 ,ALPHA, X , 1  , Y , 1  )

ALPHA    =  (2.0, 3.0)
X        =  ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
```

Output:

```
Y        =  ((-4.0, 7.0), (4.0, 6.0), (-9.0, 19.0))
```

Example 6

This example shows a scalar, α, containing a real number, and vectors ***x*** and ***y***, containing complex numbers, where both vectors have a stride of 1.

Call Statement and Input:

```
             N  ALPHA  X  INCX  Y  INCY
             |    |    |   |    |   |
CALL CSYAX( 3 , 2.0 , X , 1  , Y , 1  )

X        =  ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
```

Output:

```
Y        =  ((2.0, 4.0), (4.0, 0.0), (6.0, 10.0))
```

SZAXPY, DZAXPY, CZAXPY, and ZZAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in a Vector Z)

Purpose

These subprograms perform the following computation, using the scalar α and vectors ***x***, ***y***, and ***z***:

$\boldsymbol{z} \leftarrow \boldsymbol{y} + \alpha \boldsymbol{x}$

Table 85: Data Types.

Data Types	
α, ***x***, ***y***, ***z***	**Subprogram**
Short-precision real	SZAXPY
Long-precision real	DZAXPY
Short-precision complex	CZAXPY
Long-precision complex	ZZAXPY

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SZAXPY | DZAXPY | CZAXPY | ZZAXPY (*n*, *alpha*, *x*, *incx*, *y*, *incy*, *z*, *incz*)
C and C++	szaxpy | dzaxpy | czaxpy | zzaxpy (*n*, *alpha*, *x*, *incx*, *y*, *incy*, *z*, *incz*);
CBLAS	cblas_szaxpy | cblas_dzaxpy | cblas_czaxpy | cblas_zzaxpy (*n*, *alpha*, *x*, *incx*, *y*, *incy*, *z*, *incz*);

On Entry

n
is the number of elements in vectors ***x, y,*** and ***z***.

Specified as: an integer; $n \geq 0$.

alpha
is the scalar α.

Specified as: a number of the data type indicated in Table 85 on page 280.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 85 on page 280.

incx
is the stride for vector ***x***.

Specified as: an integer. It can have any value.

y
is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 85 on page 280.

incy
is the stride for vector ***y***.

Specified as: an integer. It can have any value.

z

See On Return.

incz

is the stride for vector ***z***.

Specified as: an integer. It can have any value.

On Return

z

is the vector ***z*** of length *n*, containing the result of the computation ***y***+α***x*** . Returned as: a one-dimensional array of (at least) length 1+(*n*-1)|*incz*|, containing numbers of the data type indicated in Table 85 on page 280.

Notes

1. If you specify the same vector for ***x*** and ***z***, then *incx* and *incz* must be equal; otherwise, results are unpredictable. The same is true for ***y*** and ***z***.
2. If you specify different vectors for ***x*** and ***z***, they must have no common elements; otherwise, results are unpredictable. The same is true for ***y*** and ***z***. See "Vector concepts" on page 71.

Function

The computation is expressed as follows:

$$\begin{bmatrix} z_1 \\ \cdot \\ \cdot \\ \cdot \\ z_n \end{bmatrix} \leftarrow \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

See reference ["95" on page 1258]. If *n* is 0, no computation is performed. For CZAXPY, intermediate results are accumulated in long precision when the VSX unit is not used.

Error conditions

Computational Errors

None

Input-Argument Errors

n < 0

Examples

Example 1

This example shows vectors ***x*** and ***y*** with positive strides.

Call Statement and Input:

```
             N  ALPHA  X  INCX  Y  INCY  Z  INCZ
             |    |    |   |    |   |    |   |
CALL SZAXPY( 5 , 2.0 , X , 1  , Y , 2  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0)
```

Output:

```
Z        =  (3.0, 5.0, 7.0, 9.0, 11.0)
```

Example 2

This example shows vectors ***x*** and ***y*** having strides of opposite sign, and an output vector ***z*** having a positive stride. For ***y***, which has negative stride, processing begins at element `Y(5)`, which is 1.0.

Call Statement and Input:

```
             N  ALPHA  X  INCX  Y   INCY  Z  INCZ
             |    |    |   |    |    |    |   |
CALL SZAXPY( 5 , 2.0 , X , 1  , Y , -1  , Z , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
Z        =  (3.0, . , 6.0, . , 9.0, . , 12.0, . , 15.0)
```

Example 3

This example shows a vector, ***x***, with 0 stride, and a vector, ***z***, with negative stride. ***x*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***x***. For vector ***z***, results are stored beginning in element `Z(5)`.

Call Statement and Input:

```
             N  ALPHA  X  INCX  Y  INCY  Z   INCZ
             |    |    |   |    |   |    |    |
CALL SZAXPY( 5 , 2.0 , X , 0  , Y , 1  , Z , -1  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

```
Z        =  (3.0, 4.0, 5.0, 6.0, 7.0)
```

Example 4

This example shows a vector, ***y***, with 0 stride. ***y*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***y***.

Call Statement and Input:

```
             N  ALPHA  X  INCX  Y  INCY  Z  INCZ
             |    |    |   |    |   |    |   |
CALL SZAXPY( 5 , 2.0 , X , 1  , Y , 0  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output:

```
Z        =  (7.0, 9.0, 11.0, 13.0, 15.0)
```

Example 5

This example shows how SZAXPY can be used to compute a scalar value. In this case, vectors ***x*** and ***y*** contain scalar values. The strides of all vectors, ***x***, ***y***, and ***z***, are 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input:

```
             N  ALPHA  X  INCX  Y  INCY  Z  INCZ
             |    |    |   |    |   |    |   |
CALL SZAXPY( 1 , 2.0 , X , 0  , Y , 0  , Z , 0  )

X        =  (1.0)
Y        =  (5.0)
```

Output:

```
Z        =  (7.0)
```

Example 6

This example shows vectors ***x*** and ***y***, containing complex numbers and having positive strides.

Call Statement and Input:

```
              N  ALPHA  X  INCX  Y  INCY  Z  INCZ
              |    |    |   |    |   |    |   |
CALL CZAXPY( 3 ,ALPHA, X , 1   , Y , 2   , Z , 1  )

ALPHA    =  (2.0, 3.0)
X        =  ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
Y        =  ((1.0, 1.0), . , (0.0, 2.0), . , (5.0, 4.0))
```

Output:

```
Z        =  ((-3.0, 8.0), (4.0, 8.0), (-4.0, 23.0))
```

Sparse Vector-Scalar Subprograms

This contains the sparse vector-scalar subprogram descriptions.

SSCTR, DSCTR, CSCTR, ZSCTR (Scatter the Elements of a Sparse Vector X in Compressed-Vector Storage Mode into Specified Elements of a Sparse Vector Y in Full-Vector Storage Mode)

Purpose

These subprograms scatter the elements of sparse vector ***x***, stored in compressed-vector storage mode, into specified elements of sparse vector ***y***, stored in full-vector storage mode.

Table 86: Data Types.

Data Types	
x, ***y***	**Subprogram**
Short-precision real	SSCTR
Long-precision real	DSCTR
Short-precision complex	CSCTR
Long-precision complex	ZSCTR

Syntax

Language	**Syntax**
Fortran	CALL SSCTR | DSCTR | CSCTR | ZSCTR (*nz*, *x*, *indx*, *y*)
C and C++	ssctr | dsctr | csctr | zsctr (*nz*, *x*, *indx*, *y*);

On Entry

nz
: is the number of elements in sparse vector ***x***, stored in compressed-vector storage mode. Specified as: an integer; $nz \geq 0$.

x
: is the sparse vector ***x***, containing *nz* elements, stored in compressed-vector storage mode in an array, referred to as X. Specified as: a one-dimensional array of (at least) length *nz*, containing numbers of the data type indicated in Table 86 on page 283.

indx
: is the array, referred to as INDX, containing the *nz* indices that indicate the positions of the elements of the sparse vector ***x*** when in full-vector storage mode. They also indicate the positions in vector ***y*** into which the elements are copied.

  Specified as: a one-dimensional array of (at least) length *nz*, containing integers.

y
: See On Return.

On Return

y
: is the sparse vector ***y***, stored in full-vector storage mode, of (at least) length max(INDX(*i*)) for *i* = 1, *nz*, into which *nz* elements of vector ***x*** are copied at positions indicated by the indices array INDX.

  Returned as: a one-dimensional array of (at least) length max(INDX(*i*)) for *i* = 1, *nz*, containing numbers of the data type indicated in Table 86 on page 283.

Notes

1. Each value specified in array INDX must be unique; otherwise, results are unpredictable.
2. Vectors ***x*** and ***y*** must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
3. For a description of how sparse vectors are stored, see "Sparse Vector" on page 75.

Function

The copy is expressed as follows:

$$y_{\texttt{INDX}(i)} \leftarrow x_i \quad \text{for } i = 1, nz$$

where:

x is a sparse vector, stored in compressed-vector storage mode.
INDX is the indices array for sparse vector ***x***.
y is a sparse vector, stored in full-vector storage mode.

See reference ["39" on page 1255]. If *nz* is 0, no copy is performed.

Error conditions

Computational Errors
: None

Input-Argument Errors
: $nz < 0$

Examples

Example 1
: This example shows how to use SSCTR to copy a sparse vector ***x*** of length 5 into the following vector ***y***, where the elements of array INDX are in ascending order:

```
          Y = (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0 )
```

Call Statement and Input:

```
             NZ  X   INDX   Y
             |   |    |     |
CALL SSCTR( 5 , X , INDX , Y )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
INDX     =  (1, 3, 4, 7, 10)
```

Output:

```
Y        =  (1.0, 2.0, 2.0, 3.0, 6.0, 10.0, 4.0, 8.0, 9.0, 5.0)
```

Example 2

This example shows how to use SSCTR to copy a sparse vector ***x*** of length 5 into the following vector ***y***, where the elements of array INDX are in random order:

```
         Y = (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0 )
```

Call Statement and Input:

```
             NZ  X   INDX   Y
             |   |    |     |
CALL SSCTR( 5 , X , INDX , Y )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
INDX     =  (4, 3, 1, 10, 7)
```

Output:

```
Y        =  (3.0, 2.0, 2.0, 1.0, 6.0, 10.0, 5.0, 8.0, 9.0, 4.0)
```

Example 3

This example shows how to use CSCTR to copy a sparse vector ***x*** of length 3 into the following vector ***y***, where the elements of array INDX are in random order:

```
         Y = ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
```

Call Statement and Input:

```
             NZ  X   INDX   Y
             |   |    |     |
CALL CSCTR( 3 , X , INDX , Y )

X        =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
INDX     =  (4, 1, 3)
```

Output:

```
Y        =  ((3.0, 4.0), (-2.0, 3.0), (5.0, 6.0), (1.0, 2.0))
```

SGTHR, DGTHR, CGTHR, and ZGTHR (Gather Specified Elements of a Sparse Vector Y in Full-Vector Storage Mode into a Sparse Vector X in Compressed-Vector Storage Mode)

Purpose

These subprograms gather specified elements of vector ***y***, stored in full-vector storage mode, into sparse vector ***x***, stored in compressed-vector storage mode.

Table 87: Data Types.

Data Types	
x, ***y***	**Subprogram**

Table 87: Data Types. (continued)

Data Types	
Short-precision real	SGTHR
Long-precision real	DGTHR
Short-precision complex	CGTHR
Long-precision complex	ZGTHR

Syntax

Language	Syntax
Fortran	CALL SGTHR | DGTHR | CGTHR | ZGTHR (*nz*, *y*, *x*, *indx*)
C and C++	sgthr | dgthr | cgthr | zgthr (*nz*, *y*, *x*, *indx*);

On Entry

nz
is the number of elements in sparse vector ***x***, stored in compressed-vector storage mode. Specified as: an integer; *nz* ≥ 0.

y
is the sparse vector ***y***, stored in full-vector storage mode, of (at least) length max(`INDX`(*i*)) for *i* = 1, *nz*, from which *nz* elements are copied from positions indicated by the indices array `INDX`.

Specified as: a one-dimensional array of (at least) length max(`INDX`(*i*)) for *i* = 1, *nz*, containing numbers of the data type indicated in Table 87 on page 285.

x
See On Return.

indx
is the array, referred to as `INDX`, containing the *nz* indices that indicate the positions of the elements of the sparse vector ***x*** when in full-vector storage mode. They also indicate the positions in vector ***y*** from which elements are copied.

Specified as: a one-dimensional array of (at least) length *nz*, containing integers.

On Return

x
is the sparse vector ***x***, containing *nz* elements, stored in compressed-vector storage mode in an array, referred to as X, into which are copied the elements of vector ***y*** from positions indicated by the indices array `INDX`.

Returned as: a one-dimensional array of (at least) length *nz*, containing numbers of the data type indicated in Table 87 on page 285.

Notes

1. Vectors ***x*** and ***y*** must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
2. For a description of how sparse vectors are stored, see "Sparse Vector" on page 75.

Function

The copy is expressed as follows:

$x_i \leftarrow y_{\mathtt{INDX}(i)}$ for $i = 1, nz$

where:

x is a sparse vector, stored in compressed-vector storage mode.
`INDX` is the indices array for sparse vector ***x***.
y is a sparse vector, stored in full-vector storage mode.

See reference [“39” on page 1255]. If *nz* is 0, no copy is performed.

Error conditions

Computational Errors
None

Input-Argument Errors
nz < 0

Examples

Example 1
This example shows how to use SGTHR to copy specified elements of a vector ***y*** into a sparse vector ***x*** of length 5, where the elements of array `INDX` are in ascending order.

Call Statement and Input:

```
             NZ  Y   X   INDX
             |   |   |    |
CALL SGTHR( 5 , Y , X , INDX )

Y        =  (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
INDX     =  (1, 3, 4, 7, 9)
```

Output:

```
X        =  (6.0, 4.0, 7.0, -2.0, 9.0)
```

Example 2
This example shows how to use SGTHR to copy specified elements of a vector ***y*** into a sparse vector ***x*** of length 5, where the elements of array `INDX` are in random order. (Note that the element 0.0 occurs in output vector ***x***. This does not produce an error.)

Call Statement and Input:

```
             NZ  Y   X   INDX
             |   |   |    |
CALL SGTHR( 5 , Y , X , INDX )

Y        =  (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
INDX     =  (4, 3, 1, 10, 7)
```

Output:

```
X        =  (7.0, 4.0, 6.0, 0.0, -2.0)
```

Example 3
This example shows how to use CGTHR to copy specified elements of a vector, ***y***, into a sparse vector, ***x***, of length 3, where the elements of array `INDX` are in random order.

Call Statement and Input:

```
             NZ  Y   X   INDX
             |   |   |    |
CALL CGTHR( 3 , Y , X , INDX )

Y        =  ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
INDX     =  (4, 1, 3)
```

Output:

```
X        =  ((9.0, 0.0), (6.0, 5.0), (15.0, 4.0))
```

SGTHRZ, DGTHRZ, CGTHRZ, and ZGTHRZ (Gather Specified Elements of a Sparse Vector Y in Full-Vector Mode into a Sparse Vector X in Compressed-Vector Mode, and Zero the Same Specified Elements of Y)

Purpose

These subprograms gather specified elements of sparse vector ***y***, stored in full-vector storage mode, into sparse vector ***x***, stored in compressed-vector storage mode, and zero the same specified elements of vector ***y***.

Table 88: Data Types.

Data Types	
x, ***y***	**Subprogram**
Short-precision real	SGTHRZ
Long-precision real	DGTHRZ
Short-precision complex	CGTHRZ
Long-precision complex	ZGTHRZ

Syntax

Language	**Syntax**
Fortran	CALL SGTHRZ | DGTHRZ | CGTHRZ | ZGTHRZ (*nz*, *y*, *x*, *indx*)
C and C++	sgthrz | dgthrz | cgthrz | zgthrz (*nz*, *y*, *x*, *indx*);

On Entry

nz

is the number of elements in sparse vector ***x***, stored in compressed-vector storage mode. Specified as: an integer; *nz* ≥ 0.

y

is the sparse vector ***y***, stored in full-vector storage mode, of (at least) length max(`INDX`(*i*)) for *i* = 1, *nz*, from which *nz* elements are copied from positions indicated by the indices array `INDX`.

Specified as: a one-dimensional array of (at least) length max(`INDX`(*i*)) for *i* = 1, *nz*, containing numbers of the data type indicated in Table 88 on page 288.

x

See On Return.

indx

is the array, referred to as `INDX`, containing the *nz* indices that indicate the positions of the elements of the sparse vector ***x*** when in full-vector storage mode. They also indicate the positions in vector ***y*** from which elements are copied then set to zero.

Specified as: a one-dimensional array of (at least) length *nz*, containing integers.

On Return

y

is the sparse vector ***y***, stored in full-vector storage mode, of (at least) length max(`INDX`(*i*)) for *i* = 1, *nz*, whose elements are set to zero at positions indicated by the indices array `INDX`.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 88 on page 288.

x

is the sparse vector ***x***, containing *nz* elements stored in compressed-vector storage mode in an array, referred to as X, into which are copied the elements of vector ***y*** from positions indicated by the indices array `INDX`.

Returned as: a one-dimensional array of (at least) length *nz*, containing numbers of the data type indicated in Table 88 on page 288.

Notes

1. Each value specified in array `INDX` must be unique; otherwise, results are unpredictable.
2. Vectors ***x*** and ***y*** must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.
3. For a description of how sparse vectors are stored, see “Sparse Vector” on page 75.

Function

The copy is expressed as follows:

$x_i \leftarrow y_{\mathtt{INDX}(i)}$
$y_{\mathtt{INDX}(i)} \leftarrow 0.0$ (for SGTHRZ and DGTHRZ)
$y_{\mathtt{INDX}(i)} \leftarrow (0.0,0.0)$ (for CGTHRZ and ZGTHRZ)
for $i = 1,nz$

where:

x is a sparse vector, stored in compressed-vector storage mode.
`INDX` is the indices array for sparse vector ***x***.
y is a sparse vector, stored in full-vector storage mode.

See reference [“39” on page 1255]. If *nz* is 0, no computation is performed.

Error conditions

Computational Errors
None

Input-Argument Errors
$nz < 0$

Examples

Example 1

This example shows how to use SGTHRZ to copy specified elements of a vector ***y*** into a sparse vector ***x*** of length 5, where the elements of array `INDX` are in ascending order.

Call Statement and Input:

```
              NZ  Y   X   INDX
              |   |   |    |
CALL SGTHRZ( 5 , Y , X , INDX )

Y        =  (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
INDX     =  (1, 3, 4, 7, 9)
```

Output:

```
Y        =  (0.0, 2.0, 0.0, 0.0, 6.0, 10.0, 0.0, 8.0, 0.0, 0.0)
X        =  (6.0, 4.0, 7.0, -2.0, 9.0)
```

Example 2

This example shows how to use SGTHRZ to copy specified elements of a vector ***y*** into a sparse vector ***x*** of length 5, where the elements of array `INDX` are in random order. (Note that the element 0.0 occurs in output vector ***x***. This does not produce an error.)

Call Statement and Input:

```
             NZ  Y   X   INDX
             |   |   |    |
CALL SGTHRZ( 5 , Y , X , INDX )

Y        =  (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
INDX     =  (4, 3, 1, 10, 7)
```

Output:

```
Y        =  (0.0, 2.0, 0.0, 0.0, 6.0, 10.0, 0.0, 8.0, 9.0, 0.0)
X        =  (7.0, 4.0, 6.0, 0.0, -2.0)
```

Example 3

This example shows how to use CGTHRZ to copy specified elements of a vector ***y*** into a sparse vector ***x*** of length 3, where the elements of array INDX are in random order.

Call Statement and Input:

```
             NZ  Y   X   INDX
             |   |   |    |
CALL CGTHRZ( 3 , Y , X , INDX )

Y        =  ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
INDX     =  (4, 1, 3)
```

Output:

```
Y        =  ((0.0, 0.0), (-2.0, 3.0), (0.0, 0.0), (0.0, 0.0))
X        =  ((9.0, 0.0), (6.0, 5.0), (15.0, 4.0))
```

SAXPYI, DAXPYI, CAXPYI, and ZAXPYI (Multiply a Sparse Vector X in Compressed-Vector Storage Mode by a Scalar, Add to a Sparse Vector Y in Full-Vector Storage Mode, and Store in the Vector Y)

Purpose

These subprograms multiply sparse vector ***x***, stored in compressed-vector storage mode, by scalar α, add it to sparse vector ***y***, stored in full-vector storage mode, and store the result in vector ***y***.

Table 89: Data Types.

Data Types	
α, ***x***, ***y***	**Subprogram**
Short-precision real	SAXPYI
Long-precision real	DAXPYI
Short-precision complex	CAXPYI
Long-precision complex	ZAXPYI

Syntax

Language	**Syntax**
Fortran	CALL SAXPYI | DAXPYI | CAXPYI | ZAXPYI (*nz*, *alpha*, *x*, *indx*, *y*)
C and C++	saxpyi | daxpyi | caxpyi | zaxpyi (*nz*, *alpha*, *x*, *indx*, *y*);

On Entry

nz
: is the number of elements in sparse vector ***x***, stored in compressed-vector storage mode. Specified as: an integer; $nz \geq 0$.

alpha
: is the scalar α. Specified as: a number of the data type indicated in Table 89 on page 290.

x
: is the sparse vector ***x***, containing *nz* elements, stored in compressed-vector storage mode in an array, referred to as X. Specified as: a one-dimensional array of (at least) length *nz*, containing numbers of the data type indicated in Table 89 on page 290.

indx
: is the array, referred to as `INDX`, containing the *nz* indices that indicate the positions of the elements of the sparse vector ***x*** when in full-vector storage mode. They also indicate the positions of the elements in vector ***y*** that are used in the computation.

  Specified as: a one-dimensional array of (at least) length *nz*, containing integers.

y
: is the sparse vector ***y***, stored in full-vector storage mode, of (at least) length max(`INDX`(*i*)) for *i* = 1, *nz*. Specified as: a one-dimensional array of (at least) length max(`INDX`(*i*)) for *i* = 1, *nz*, containing numbers of the data type indicated in Table 89 on page 290.

On Return

y
: is the sparse vector ***y***, stored in full-vector storage mode, of (at least) length max(`INDX`(*i*)) for *i* = 1, *nz* containing the results of the computation, stored at positions indicated by the indices array `INDX`.

  Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 89 on page 290.

Notes

1. Each value specified in array `INDX` must be unique; otherwise, results are unpredictable.
2. Vectors ***x*** and ***y*** must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
3. For a description of how sparse vectors are stored, see "Sparse Vector" on page 75.

Function

The computation is expressed as follows:

$$y_{\mathtt{INDX}(i)} \leftarrow y_{\mathtt{INDX}(i)} + \alpha x_i \quad \text{for } i = 1, nz$$

where:

x is a sparse vector, stored in compressed-vector storage mode.
`INDX` is the indices array for sparse vector ***x***.
y is a sparse vector, stored in full-vector storage mode.

See reference ["39" on page 1255]. If α or *nz* is zero, no computation is performed. For SAXPYI and CAXPYI, intermediate results are accumulated in long-precision.

Error conditions

Computational Errors
: None

Input-Argument Errors
: $nz < 0$

Examples

Example 1

This example shows how to use SAXPYI to perform a computation using a sparse vector ***x*** of length 5, where the elements of array INDX are in ascending order.

Call Statement and Input:

```
             NZ ALPHA  X   INDX   Y
             |    |    |    |     |
CALL SAXPYI( 5 , 2.0 , X , INDX , Y )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
INDX     =  (1, 3, 4, 7, 10)
Y        =  (1.0, 5.0, 4.0, 3.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
```

Output:

```
Y        =  (3.0, 5.0, 8.0, 9.0, 6.0, 10.0, 6.0, 8.0, 9.0, 10.0)
```

Example 2

This example shows how to use SAXPYI to perform a computation using a sparse vector ***x*** of length 5, where the elements of array INDX are in random order.

Call Statement and Input:

```
             NZ ALPHA  X   INDX   Y
             |    |    |    |     |
CALL SAXPYI( 5 , 2.0 , X , INDX , Y )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
INDX     =  (4, 3, 1, 10, 7)
Y        =  (1.0, 5.0, 4.0, 3.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
```

Output:

```
Y        =  (7.0, 5.0, 8.0, 5.0, 6.0, 10.0, 8.0, 8.0, 9.0, 8.0)
```

Example 3

This example shows how to use CAXPYI to perform a computation using a sparse vector ***x*** of length 3, where the elements of array INDX are in random order.

Call Statement and Input:

```
             NZ  ALPHA   X   INDX   Y
             |     |     |    |     |
CALL CAXPYI( 3 , ALPHA , X , INDX , Y )

ALPHA    =  (2.0, 3.0)
X        =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
INDX     =  (4, 1, 3)
Y        =  ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
```

Output:

```
Y        =  ((0.0, 22.0), (-2.0, 3.0), (7.0, 31.0), (5.0, 7.0))
```

SDOTI, DDOTI, CDOTUI, ZDOTUI, CDOTCI, and ZDOTCI (Dot Product of a Sparse Vector X in Compressed-Vector Storage Mode and a Sparse Vector Y in Full-Vector Storage Mode)

Purpose

SDOTI, DDOTI, CDOTUI, and ZDOTUI compute the dot product of sparse vector ***x***, stored in compressed-vector storage mode, and full vector ***y***, stored in full-vector storage mode.

CDOTCI and ZDOTCI compute the dot product of the complex conjugate of sparse vector ***x***, stored in compressed-vector storage mode, and full vector ***y***, stored in full-vector storage mode.

Table 90: Data Types.

Data Types	
***x*, *y*, Result**	**Subprogram**
Short-precision real	SDOTI
Long-precision real	DDOTI
Short-precision complex	CDOTUI
Long-precision complex	ZDOTUI
Short-precision complex	CDOTCI
Long-precision complex	ZDOTCI

Syntax

Language	Syntax
Fortran	SDOTI | DDOTI | CDOTUI | ZDOTUI | CDOTCI | ZDOTCI (*nz*, *x*, *indx*, *y*)
C and C++	sdoti | ddoti | cdotui | zdotui | cdotci | zdotci (*nz*, *x*, *indx*, *y*);

On Entry

nz

is the number of elements in sparse vector ***x***, stored in compressed-vector storage mode. Specified as: an integer; *nz* ≥ 0.

x

is the sparse vector ***x***, containing *nz* elements, stored in compressed-vector storage mode in an array, referred to as X. Specified as: a one-dimensional array of (at least) length *nz*, containing numbers of the data type indicated in Table 90 on page 293.

indx

is the array, referred to as `INDX`, containing the *nz* indices that indicate the positions of the elements of the sparse vector ***x*** when in full-vector storage mode. They also indicate the positions of elements in vector ***y*** that are used in the computation.

Specified as: a one-dimensional array of (at least) length *nz*, containing integers.

y

is the sparse vector ***y***, stored in full-vector storage mode, of (at least) length max(`INDX`(*i*)) for *i* = 1, *nz*. Specified as: a one-dimensional array of (at least) length max(`INDX`(*i*)) for *i* = 1, *nz*, containing numbers of the data type indicated in Table 90 on page 293.

On Return

Function value

is the result of the dot product computation.

Returned as: a number of the data type indicated in Table 90 on page 293.

Notes

1. Declare this function in your program as returning a value of the data type indicated in Table 90 on page 293.
2. For a description of how sparse vectors are stored, see "Sparse Vector" on page 75.

Function

For SDOTI, DDOTI, CDOTUI, and ZDOTUI, the dot product computation is expressed as follows:

$$\sum_{i=1}^{nz} x_i y_{\mathrm{INDX}(i)} = x_1 y_{\mathrm{INDX}(1)} + x_2 y_{\mathrm{INDX}(2)} + \dots + x_{nz} y_{\mathrm{INDX}(nz)}$$

For CDOTCI and ZDOTCI, the dot product computation is expressed as follows:

$$\sum_{i=1}^{nz} \bar{x}_i y_{\mathrm{INDX}(i)} = \bar{x}_1 y_{\mathrm{INDX}(1)} + \bar{x}_2 y_{\mathrm{INDX}(2)} + \dots + \bar{x}_{nz} y_{\mathrm{INDX}(nz)}$$

where:

x is a sparse vector, stored in compressed-vector storage mode.

$\bar{x}$ is the complex conjugate of a sparse vector, stored in compressed - vector storage mode.

INDX is the indices array for sparse vector ***x***.

y is a sparse vector, stored in full-vector storage mode.

See reference [“39” on page 1255]. The result is returned as the function value. If *nz* is 0, then zero is returned as the value of the function.

For SDOTI, CDOTUI, and CDOTCI, intermediate results are accumulated in long-precision.

Error conditions

Computational Errors
None

Input-Argument Errors
nz < 0

Examples

Example 1

This example shows how to use SDOTI to compute a dot product using a sparse vector ***x*** of length 5, where the elements of array INDX are in ascending order.

Function Reference and Input:

```
              NZ  X   INDX   Y
              |   |    |     |
DOTT = SDOTI( 5 , X , INDX , Y )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
INDX     =  (1, 3, 4, 7, 10)
Y        =  (1.0, 5.0, 4.0, 3.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
```

Output:

```
DOTT     =  (1.0 + 8.0 + 9.0 -8.0 + 0.0) = 10.0
```

Example 2

This example shows how to use SDOTI to compute a dot product using a sparse vector ***x*** of length 5, where the elements of array INDX are in random order.

Function Reference and Input:

```
              NZ  X   INDX   Y
              |   |    |     |
DOTT = SDOTI( 5 , X , INDX , Y )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
INDX     =  (4, 3, 1, 10, 7)
Y        =  (1.0, 5.0, 4.0, 3.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
```

Output:

```
DOTT     =  (3.0 + 8.0 + 3.0 + 0.0 -10.0) = 4.0
```

Example 3

This example shows how to use CDOTUI to compute a dot product using a sparse vector ***x*** of length 3, where the elements of array INDX are in ascending order.

Function Reference and Input:

```
               NZ  X   INDX   Y
               |   |    |     |
DOTT = CDOTUI( 3 , X , INDX , Y )

X        =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
INDX     =  (1, 3, 4)
Y        =  ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
```

Output:

```
DOTT     =  (70.0, 143.0)
```

Example 4

This example shows how to use CDOTCI to compute a dot product using the complex conjugate of a sparse vector ***x*** of length 3, where the elements of array INDX are in random order.

Function Reference and Input:

```
               NZ  X   INDX   Y
               |   |    |     |
DOTT = CDOTCI( 3 , X , INDX , Y )

X        =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
INDX     =  (4, 1, 3)
Y        =  ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
```

Output:

```
DOTT     =  (146.0, -97.0)
```

Matrix-Vector Subprograms

This contains the matrix-vector subprogram descriptions.

SGEMV, DGEMV, CGEMV, ZGEMV, SGEMX, DGEMX, SGEMTX, and DGEMTX (Matrix-Vector Product for a General Matrix, Its Transpose, or Its Conjugate Transpose)

Purpose

SGEMV and DGEMV compute the matrix-vector product for either a real general matrix or its transpose, using the scalars α and β, vectors ***x*** and ***y***, and matrix ***A*** or its transpose:

$\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A}\boldsymbol{x}$

$\boldsymbol{y} \leftarrow \beta\boldsymbol{y}+\alpha\,\boldsymbol{A}^T\boldsymbol{x}$

CGEMV and ZGEMV compute the matrix-vector product for either a complex general matrix, its transpose, or its conjugate transpose, using the scalars α and β, vectors ***x*** and ***y***, and matrix ***A***, its transpose, or its conjugate transpose:

$\boldsymbol{y} \leftarrow \beta\,\boldsymbol{y}+\alpha\,\boldsymbol{A}\boldsymbol{x}$
$\boldsymbol{y} \leftarrow \beta\,\boldsymbol{y}+\alpha\,\boldsymbol{A}^T\boldsymbol{x}$
$\boldsymbol{y} \leftarrow \beta\,\boldsymbol{y}+\alpha\,\boldsymbol{A}^H\boldsymbol{x}$

SGEMX and DGEMX compute the matrix-vector product for a real general matrix, using the scalar α, vectors ***x*** and ***y***, and matrix ***A***:

$\boldsymbol{y} \leftarrow \boldsymbol{y}+\alpha\,\boldsymbol{A}\boldsymbol{x}$

SGEMTX and DGEMTX compute the matrix-vector product for the transpose of a real general matrix, using the scalar α, vectors ***x*** and ***y***, and the transpose of matrix ***A***:

$\boldsymbol{y} \leftarrow \boldsymbol{y}+\alpha\,\boldsymbol{A}^T\boldsymbol{x}$

Table 91: Data Types.

Data Types	
α, β, ***x***, ***y***, ***A***	**Subprogram**
Short-precision real	SGEMV, SGEMX, and SGEMTX
Long-precision real	DGEMV, DGEMX, and DGEMTX
Short-precision complex	CGEMV
Long-precision complex	ZGEMV

Note:

1. SGEMV and DGEMV are Level 2 BLAS subroutines. It is suggested that these subroutines be used instead of SGEMX, DGEMX, SGEMTX, and DGEMTX, which are provided only for compatibility with earlier releases of ESSL.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	**Syntax**
Fortran	CALL SGEMV | DGEMV | CGEMV | ZGEMV (*transa*, *m*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*) CALL SGEMX | DGEMX | SGEMTX | DGEMTX (*m*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *y*, *incy*)

C and C++	sgemv | dgemv | cgemv | zgemv (*transa*, *m*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*);
	sgemx | dgemx | sgemtx | dgemtx (*m*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *y*, *incy*);

CBLAS	cblas_sgemv | cblas_dgemv | cblas_cgemv | cblas_zgemv (*cblas_layout*, *cblas_transa*, *m*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*);

On Entry

cblas_layout
indicates whether the input matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.

- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

transa
indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation.

Specified as: a single character. It must be 'N', 'T', or 'C'.

cblas_transa
indicates the form of matrix ***A*** to use in the computation, where:

If *cblas_transa* = CblasNoTrans, ***A*** is used in the computation.

If *cblas_transa* = CblasTrans, $\boldsymbol{A}^T$ is used in the computation.

If *cblas_transa* = CblasConjTrans, $\boldsymbol{A}^H$ is used in the computation.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

m
is the number of rows in matrix ***A***, and:

For SGEMV, DGEMV, CGEMV, and ZGEMV:

If *transa* = 'N', it is the length of vector ***y***.
If *transa* = 'T' or 'C', it is the length of vector ***x***.

For SGEMX and DGEMX, it is the length of vector ***y***.

For SGEMTX and DGEMTX, it is the length of vector ***x***.

Specified as: an integer; $0 \le m \le lda$.

n
is the number of columns in matrix ***A***, and:

For SGEMV, DGEMV, CGEMV, and ZGEMV:

If *transa* = 'N', it is the length of vector ***x***.
If *transa* = 'T' or 'C', it is the length of vector ***y***.

For SGEMX and DGEMX, it is the length of vector ***x***.

For SGEMTX and DGEMTX, it is the length of vector ***y***.

Specified as: an integer; $n \ge 0$.

alpha
is the scaling constant α.

Specified as: a number of the data type indicated in Table 91 on page 296.

a
is the *m* by *n* matrix ***A***, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:

If *transa* = 'N', ***A*** is used in the computation.
If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation.
If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation.

For SGEMX and DGEMX, ***A*** is used in the computation.

For SGEMTX and DGEMTX, $\boldsymbol{A}^T$ is used in the computation.

Note: No data should be moved to form $\boldsymbol{A}^T$ or $\boldsymbol{A}^H$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 91 on page 296.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *m*.

x

is the vector ***x***, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:

If *transa* = 'N', it has length *n*.
If *transa* = 'T' or 'C', it has length *m*.

For SGEMX and DGEMX, it has length *n*.

For SGEMTX and DGEMTX, it has length *m*.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 91 on page 296, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:

If *transa* = 'N', it must have at least 1+(*n*-1)|*incx*| elements.
If *transa* = 'T' or 'C', it must have at least 1+(*m*-1)|*incx*| elements.

For SGEMX and DGEMX, it must have at least 1+(*n*-1)|*incx*| elements.

For SGEMTX and DGEMTX, it must have at least 1+(*m*-1)|*incx*| elements.

beta

is the scaling constant β.

Specified as: a number of the data type indicated in Table 91 on page 296.

incx

is the stride for vector ***x***.

Specified as: an integer; It can have any value.

y

is the vector ***y***, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:

If *transa* = 'N', it has length *m*.
If *transa* = 'T' or 'C', it has length *n*.

For SGEMX and DGEMX, it has length *m*.

For SGEMTX and DGEMTX, it has length *n*.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 91 on page 296, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:

If *transa* = 'N', it must have at least 1+(*m*-1)|*incy*| elements.
If *transa* = 'T' or 'C', it must have at least 1+(*n*-1)|*incy*| elements.

For SGEMX and DGEMX, it must have at least 1+(*m*-1)|*incy*| elements.

For SGEMTX and DGEMTX, it must have at least 1+(*n*-1)|*incy*| elements.

incy
is the stride for vector ***y***.

Specified as: an integer; *incy* > 0 or *incy* < 0.

On Return

y
is the vector ***y***, containing the result of the computation, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:

If *transa* = 'N', it has length *m*.
If *transa* = 'T' or 'C', it has length *n*.

For SGEMX and DGEMX, it has length *m*.

For SGEMTX and DGEMTX, it has length *n*.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 91 on page 296.

Notes

1. For SGEMV and DGEMV, if you specify 'C' for the *transa* argument, it is interpreted as though you specified 'T'.
2. The SGEMV, DGEMV, CGEMV, and ZGEMV subroutines accept lowercase letters for the *transa* argument.
3. In the SGEMV, DGEMV, CGEMV, and ZGEMV subroutines, *incx* = 0 is valid; however, the Level 2 BLAS standard considers *incx* = 0 to be invalid. See references [“44” on page 1255] and [“45” on page 1255].
4. Vector ***y*** must have no common elements with matrix ***A*** or vector ***x***; otherwise, results are unpredictable. See “Vector concepts” on page 71.

Function

Varying implementation techniques are used for this computation to improve performance. As a result, accuracy of the computational result may vary for different computations.

For SGEMV, CGEMV, SGEMX, and SGEMTX, intermediate results are accumulated in long precision when the VSX unit is not used. Occasionally, for performance reasons, these intermediate results are stored.

See references [“44” on page 1255], [“45” on page 1255], [“48” on page 1255], [“56” on page 1256], and [“95” on page 1258]. No computation is performed if *m* or *n* is 0 or if α is zero and β is one.

General Matrix

For SGEMV, DGEMV, CGEMV, and ZGEMV, the matrix-vector product for a general matrix:

$\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A}\boldsymbol{x}$

is expressed as follows:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_m \end{bmatrix} \leftarrow \beta \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_m \end{bmatrix} + \alpha \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

For SGEMX and DGEMX, the matrix-vector product for a real general matrix:

y←***y***+α***Ax***

is expressed as follows:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_m \end{bmatrix} \leftarrow \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_m \end{bmatrix} + \alpha \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

In these expressions:

y is a vector of length *m*.
α is a scalar.
β is a scalar.
A is an *m* by *n* matrix.
x is a vector of length *n*.

Transpose of a General Matrix

For SGEMV, DGEMV, CGEMV and ZGEMV, the matrix-vector product for the transpose of a general matrix:

$\boldsymbol{y} \leftarrow \beta\boldsymbol{y}+\alpha\boldsymbol{A}^{T}\boldsymbol{x}$

is expressed as follows:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} \leftarrow \beta \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} a_{11} & \dots & a_{m1} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{1n} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_m \end{bmatrix}$$

For SGEMTX and DGEMTX, the matrix-vector product for the transpose of a real general matrix:

$\boldsymbol{y} \leftarrow \boldsymbol{y}+\alpha\boldsymbol{A}^{T}\boldsymbol{x}$

is expressed as follows:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} \leftarrow \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} a_{11} & \dots & a_{m1} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{1n} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_m \end{bmatrix}$$

In these expressions:

y is a vector of length *n*.
α is a scalar.
β is a scalar.

$\boldsymbol{A}^T$ is the transpose of matrix ***A***, where ***A*** is an *m* by *n* matrix.
x is a vector of length *m*.

Conjugate Transpose of a General Matrix

For CGEMV and ZGEMV, the matrix-vector product for the conjugate transpose of a general matrix:

$\boldsymbol{y} \leftarrow \beta\boldsymbol{y}+\alpha\boldsymbol{A}^H\boldsymbol{x}$

is expressed as follows:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} \leftarrow \beta \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} \bar{a}_{11} & \dots & \bar{a}_{m1} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \bar{a}_{1n} & \dots & \bar{a}_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_m \end{bmatrix}$$

where:

y is a vector of length *n*.
α is a scalar.
β is a scalar.
$\boldsymbol{A}^H$ is the conjugate transpose of matrix ***A***, where ***A*** is an *m* by *n* matrix.
x is a vector of length *m*.

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. *transa* ≠ 'N', 'T', or 'C'
3. *cblas_transa* ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
4. *m* < 0
5. *m* > *lda*
6. *n* < 0
7. *lda* ≤ 0
8. *incy* = 0

Examples

Example 1
This example shows the computation for TRANSA equal to 'N', where the real general matrix ***A*** is used in the computation. Because *lda* is 10 and *n* is 3, array A must be declared as `A(E1:E2,F1:F2)`, where `E2-E1+1=10` and `F2-F1+1` ≥ 3. In this example, array A is declared as `A(1:10,0:2)`.

Call Statement and Input:

```
           TRANSA M   N  ALPHA    A      LDA  X  INCX BETA   Y  INCY
             |    |   |    |      |       |   |   |     |    |   |
CALL SGEMV( 'N' , 4 , 3 , 1.0 , A(1,0) , 10 , X , 1  , 1.0 , Y , 2  )
```

```
    | 1.0  2.0  3.0 |
    | 2.0  2.0  4.0 |
    | 3.0  2.0  2.0 |
    | 4.0  2.0  1.0 |
A = |  .    .    .  |
    |  .    .    .  |
    |  .    .    .  |
    |  .    .    .  |
    |  .    .    .  |
    |  .    .    .  |
```

```
X        =  (3.0, 2.0, 1.0)
Y        =  (4.0, . , 5.0, . , 2.0, . , 3.0)
```

Output:

```
Y        =  (14.0, . , 19.0, . , 17.0, . , 20.0)
```

Example 2

This example shows the computation for TRANSA equal to 'T', where the transpose of the real general matrix ***A*** is used in the computation. Array A must follow the same rules as given in Example 1. In this example, array A is declared as `A(-1:8,1:3)`.

Call Statement and Input:

```
           TRANSA M   N  ALPHA     A     LDA   X  INCX  BETA  Y  INCY
             |    |   |    |       |      |    |   |     |    |   |
CALL SGEMV( 'T' , 4 , 3 , 1.0 , A(-1,1) , 10 , X , 1  , 2.0 , Y , 2  )
```

A =(same as input A in Example 1)
`X = (3.0, 2.0, 1.0, 4.0)`
`Y = (1.0, . , 2.0, . , 3.0)`

Output:

```
Y        =  (28.0, . , 24.0, . , 29.0)
```

Example 3

This example shows the computation for TRANSA equal to 'N', where the complex general matrix ***A*** is used in the computation.

Call Statement and Input:

```
           TRANSA M   N   ALPHA   A   LDA  X  INCX  BETA   Y  INCY
             |    |   |     |     |    |   |   |     |     |   |
CALL CGEMV( 'N' , 5 , 3 , ALPHA , A , 10 , X , 1  , BETA , Y , 1  )

ALPHA    =  (1.0, 0.0)
```

```
    | (1.0, 2.0)  (3.0, 5.0)  (2.0, 0.0) |
    | (2.0, 3.0)  (7.0, 9.0)  (4.0, 8.0) |
    | (7.0, 4.0)  (1.0, 4.0)  (6.0, 0.0) |
    | (8.0, 2.0)  (2.0, 5.0)  (8.0, 0.0) |
A = | (9.0, 1.0)  (3.0, 6.0)  (1.0, 0.0) |
    |     .           .           .      |
    |     .           .           .      |
    |     .           .           .      |
    |     .           .           .      |
    |     .           .           .      |
```

```
X        =  ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0))
BETA     =  (1.0, 0.0)
Y        =  ((1.0, 2.0), (4.0, 0.0), (1.0, -1.0), (3.0, 4.0),
             (2.0, 0.0))
```

Output:

```
Y        =  ((12.0, 28.0), (24.0, 55.0), (10.0, 39.0), (23.0, 50.0),
             (22.0, 44.0))
```

Example 4

This example shows the computation for TRANSA equal to 'T', where the transpose of complex general matrix ***A*** is used in the computation. Because β is zero, the result of the computation is $\alpha \boldsymbol{A}^{T}\boldsymbol{x}$

Call Statement and Input:

```
           TRANSA M   N   ALPHA   A   LDA  X  INCX  BETA   Y  INCY
             |    |   |     |     |    |   |   |     |     |   |
CALL CGEMV( 'T' , 5 , 3 , ALPHA , A , 10 , X , 1  , BETA , Y , 1  )
```

ALPHA = (1.0, 0.0)
A =(same as input A in Example 3)
X = ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0), (3.0, 4.0), (2.0, 0.0))
BETA = (0.0, 0.0)
Y =(not relevant)

Output:

```
Y        =  ((42.0, 67.0), (10.0, 87.0), (50.0, 74.0))
```

Example 5

This example shows the computation for TRANSA equal to 'C', where the conjugate transpose of the complex general matrix ***A*** is used in the computation.

Call Statement and Input:

```
           TRANSA M   N   ALPHA   A   LDA  X  INCX  BETA   Y  INCY
             |    |   |     |     |    |   |   |     |     |   |
CALL CGEMV( 'C' , 5 , 3 , ALPHA , A , 10 , X , 1  , BETA , Y , 1  )
```

ALPHA = (-1.0, 0.0)
A =(same as input A in Example 3)
X = ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0), (3.0, 4.0), (2.0, 0.0))
BETA = (1.0, 0.0)
Y = ((1.0, 2.0), (4.0, 0.0), (1.0, -1.0))

Output:

```
Y        =  ((-73.0, -13.0), (-74.0, 57.0), (-49.0, -11.0))
```

Example 6

This example shows a matrix, ***A***, contained in a larger array, A. The strides of vectors ***x*** and ***y*** are positive. Because *lda* is 10 and *n* is 3, array A must be declared as `A(E1:E2,F1:F2)`, where E2-E1+1=10 and F2-F1+1 ≥ 3. For this example, array A is declared as `A(1:10,0:2)`.

Call Statement and Input:

```
            M   N  ALPHA    A     LDA   X  INCX  Y  INCY
            |   |    |      |      |    |   |    |   |
CALL SGEMX( 4 , 3 , 1.0 , A(1,0) , 10 , X , 1  , Y , 2  )
```

```
    | 1.0  2.0  3.0 |
    | 2.0  2.0  4.0 |
    | 3.0  2.0  2.0 |
    | 4.0  2.0  1.0 |
A = |  .    .    .  |
    |  .    .    .  |
    |  .    .    .  |
    |  .    .    .  |
    |  .    .    .  |
    |  .    .    .  |
```

```
X        =  (3.0, 2.0, 1.0)
Y        =  (4.0, . , 5.0, . , 2.0, . , 3.0)
```

Output:

```
Y        =  (14.0, . , 19.0, . , 17.0, . , 20.0)
```

Example 7

This example shows a matrix, ***A***, contained in a larger array, A. The strides of vectors ***x*** and ***y*** are of opposite sign. For ***y***, which has negative stride, processing begins at element `Y(7)`, which is 4.0. Array A must follow the same rules as given in Example 6. For this example, array A is declared as `A(-1:8,1:3)`.

Call Statement and Input:

```
            M   N  ALPHA     A     LDA   X  INCX  Y   INCY
            |   |    |       |      |    |   |    |    |
CALL SGEMX( 4 , 3 , 1.0 , A(-1,1) , 10 , X , 1  , Y , -2  )
```

A =(same as input A in Example 6)
`X = (3.0, 2.0, 1.0)`
`Y = (3.0, . , 2.0, . , 5.0, . , 4.0)`

Output:

```
Y        =  (20.0, . , 17.0, . , 19.0, . , 14.0)
```

Example 8

This example shows a matrix, ***A***, contained in a larger array, A, and the first element of the matrix is not the first element of the array. Array A must follow the same rules as given in Example 6. For this example, array A is declared as `A(1:10,1:3)`.

Call Statement and Input:

```
            M   N  ALPHA    A     LDA   X  INCX  Y  INCY
            |   |    |      |      |    |   |    |   |
CALL SGEMX( 4 , 3 , 1.0 , A(5,1) , 10 , X , 1  , Y , 1  )
```

```
    | .    .    .   |
    | .    .    .   |
    | .    .    .   |
    | .    .    .   |
A = | 1.0  2.0  3.0 |
    | 2.0  2.0  4.0 |
    | 3.0  2.0  2.0 |
    | 4.0  2.0  1.0 |
    |  .    .    .  |
    | .    .    . |
```

```
X        =  (3.0, 2.0, 1.0)
Y        =  (4.0, 5.0, 2.0, 3.0)
```

Output:

```
Y        =  (14.0, 19.0, 17.0, 20.0)
```

Example 9

This example shows a matrix, ***A***, and an array, A, having the same number of rows. For this case, *m* and *lda* are equal. Because *lda* is 4 and *n* is 3, array A must be declared as `A(E1:E2,F1:F2)`, where `E2-E1+1=4` and `F2-F1+1 ≥ 3`. For this example, array A is declared as `A(1:4,0:2)`.

Call Statement and Input:

```
            M   N  ALPHA    A     LDA  X  INCX  Y  INCY
            |   |    |      |      |   |   |    |   |
CALL SGEMX( 4 , 3 , 1.0 , A(1,0) , 4 , X , 1  , Y , 1  )


    | 1.0  2.0  3.0 |
A = | 2.0  2.0  4.0 |
    | 3.0  2.0  2.0 |
    | 4.0  2.0  1.0 |


X        =  (3.0, 2.0, 1.0)
Y        =  (4.0, 5.0, 2.0, 3.0)
```

Output:

```
Y        =  (14.0, 19.0, 17.0, 20.0)
```

Example 10

This example shows a matrix, ***A***, and an array, A, having the same number of rows. For this case, *m* and *lda* are equal. Because *lda* is 4 and *n* is 3, array A must be declared as `A(E1:E2,F1:F2)`, where `E2-E1+1=4` and `F2-F1+1 ≥ 3`. For this example, array A is declared as `A(1:4,0:2)`.

Call Statement and Input:

```
             M   N  ALPHA    A     LDA  X  INCX  Y  INCY
             |   |    |      |      |   |   |    |   |
CALL SGEMTX( 4 , 3 , 1.0 , A(1,0) , 4 , X , 1  , Y , 1  )


    | 1.0  2.0  3.0 |
A = | 2.0  2.0  4.0 |
    | 3.0  2.0  2.0 |
    | 4.0  2.0  1.0 |


X        =  (3.0, 2.0, 1.0, 4.0)
Y        =  (1.0, 2.0, 3.0)
```

Output:

```
Y        =  (27.0, 22.0, 26.0)
```

Example 11

This example shows a computation in which *alpha* is greater than 1. Array A must follow the same rules as given in Example 10. For this example, array A is declared as `A(-1:2,1:3)`.

Call Statement and Input:

```
             M   N  ALPHA     A     LDA  X  INCX  Y  INCY
             |   |    |       |      |   |   |    |   |
CALL SGEMTX( 4 , 3 , 2.0 , A(-1,1) , 4 , X , 1  , Y , 1  )
```

A =(same as input A in Example 10)
```
X = (3.0, 2.0, 1.0, 4.0)
Y = (1.0, 2.0, 3.0)
```

Output:

```
Y        =  (53.0, 42.0, 49.0)
```

SGER, DGER, CGERU, ZGERU, CGERC, and ZGERC (Rank-One Update of a General Matrix)

Purpose

SGER, DGER, CGERU, and ZGERU compute the rank-one update of a general matrix, using the scalar α, matrix ***A***, vector ***x***, and the transpose of vector ***y***:

$\boldsymbol{A} \leftarrow \boldsymbol{A} + \alpha \boldsymbol{x}\boldsymbol{y}^{T}$

CGERC and ZGERC compute the rank-one update of a general matrix, using the scalar α, matrix ***A***, vector ***x***, and the conjugate transpose of vector ***y***:

$\boldsymbol{A} \leftarrow \boldsymbol{A} + \alpha \boldsymbol{x}\boldsymbol{y}^{H}$

Table 92: Data Types.

Data Types	
α, ***A***, ***x***, ***y***	**Subprogram**
Short-precision real	SGER
Long-precision real	DGER
Short-precision complex	CGERU and CGERC
Long-precision complex	ZGERU and ZGERC

Note:

1. For compatibility with earlier releases of ESSL, you can use the names SGER1 and DGER1 for SGER and DGER, respectively.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SGER | DGER | CGERU | ZGERU | CGERC | ZGERC (*m*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *a*, *lda*)
C and C++	sger | dger | cgeru | zgeru | cgerc | zgerc (*m*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *a*, *lda*);
CBLAS	cblas_sger | cblas_dger | cblas_cgeru | cblas_zgeru | cblas_cgerc | cblas_zgerc (*cblas_layout*, *m*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *a*, *lda*);

On Entry

cblas_layout
indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

m

is the number of rows in matrix ***A*** and the number of elements in vector ***x***.

Specified as: an integer; $0 \le m \le lda$.

n

is the number of columns in matrix ***A*** and the number of elements in vector ***y***.

Specified as: an integer; $n \ge 0$.

alpha

is the scaling constant α.

Specified as: a number of the data type indicated in Table 92 on page 306.

x

is the vector ***x*** of length *m*.

Specified as: a one-dimensional array of (at least) length 1+(*m*-1)|*incx*|, containing numbers of the data type indicated in Table 92 on page 306.

incx

is the stride for vector ***x***.

Specified as: an integer. It can have any value.

y

is the vector ***y*** of length *n*, whose transpose or conjugate transpose is used in the computation.

Note: No data should be moved to form $\boldsymbol{y}^T$ or $\boldsymbol{y}^H$; that is, the vector ***y*** should always be stored in its untransposed form.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incy*|, containing numbers of the data type indicated in Table 92 on page 306.

incy

is the stride for vector ***y***.

Specified as: an integer. It can have any value.

a

is the *m* by *n* matrix ***A***. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 92 on page 306.

lda

is the size of the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *m*.

On Return

a

is the *m* by *n* matrix ***A***, containing the result of the computation.

Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 92 on page 306.

Notes

1. In these subroutines, *incx* = 0 and *incy* = 0 are valid; however, the Level 2 BLAS standard considers *incx* = 0 and *incy* = 0 to be invalid. See references [“44” on page 1255] and [“45” on page 1255].

2. Matrix ***A*** can have no common elements with vectors ***x*** and ***y***; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

SGER, DGER, CGERU, and ZGERU compute the rank-one update of a general matrix:

$\boldsymbol{A} \leftarrow \boldsymbol{A}+\alpha \boldsymbol{x}\boldsymbol{y}^T$

where:

A is an *m* by *n* matrix.
α is a scalar.
x is a vector of length *m*.
$\boldsymbol{y}^T$ is the transpose of vector ***y*** of length *n*.

It is expressed as follows:

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} & \dots & a_{mn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_m \end{bmatrix} \begin{bmatrix} y_1 & \dots & y_n \end{bmatrix}$$

It can also be expressed as:

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} + \alpha x_1 y_1 & \dots & a_{1n} + \alpha x_1 y_n \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} + \alpha x_m y_1 & \dots & a_{mn} + \alpha x_m y_n \end{bmatrix}$$

CGERC and ZGERC compute a slightly different rank-one update of a general matrix:

$\boldsymbol{A} \leftarrow \boldsymbol{A}+\alpha \boldsymbol{x}\boldsymbol{y}^H$

where:

A is an *m* by *n* matrix.
α is a scalar.
x is a vector of length *m*.
$\boldsymbol{y}^H$ is the conjugate transpose of vector ***y*** of length *n*.

It is expressed as follows:

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{m1} & \dots & a_{mn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_m \end{bmatrix} \begin{bmatrix} \bar{y}_1 & \dots & \bar{y}_n \end{bmatrix}$$

It can also be expressed as:

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} + \alpha x_1 \bar{y}_1 & \dots & a_{1n} + \alpha x_1 \bar{y}_n \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{m1} + \alpha x_m \bar{y}_1 & \dots & a_{mn} + \alpha x_m \bar{y}_n \end{bmatrix}$$

See references [“44” on page 1255], [“45” on page 1255], and [“95” on page 1258]. No computation is performed if *m, n*, or α is zero. For CGERU and CGERC, intermediate results are accumulated in long precision when the VSX unit is not used. For SGER, intermediate results are accumulated in long precision on some platforms when the VSX unit is not used.

Error conditions

Resource Errors:
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. $m < 0$
3. $n < 0$
4. $lda \leq 0$
5. $m > lda$

Examples

Example 1

This example shows a matrix, ***A***, contained in a larger array, A. The strides of vectors ***x*** and ***y*** are positive. Because *lda* is 10 and *n* is 3, array A must be declared as `A(E1:E2,F1:F2)`, where `E2-E1+1=10` and `F2-F1+1 ≥ 3`. For this example, array A is declared as `A(1:10,0:2)`.

Call Statement and Input:

```
            M   N  ALPHA  X  INCX  Y  INCY    A     LDA
            |   |    |    |   |    |   |      |      |
CALL SGER( 4 , 3 , 1.0 , X , 1  , Y , 2  , A(1,0) , 10 )

X        =  (3.0, 2.0, 1.0, 4.0)
Y        =  (1.0, . , 2.0, . , 3.0)
```

```
    | 1.0  2.0  3.0 |
    | 2.0  2.0  4.0 |
    | 3.0  2.0  2.0 |
    | 4.0  2.0  1.0 |
A = |  .    .    .  |
    |  .    .    .  |
    |  .    .    .  |
    |  .    .    .  |
    |  .    .    .  |
    |  .    .    .  |
```

Output:

```
    | 4.0   8.0  12.0 |
    | 4.0   6.0  10.0 |
    | 4.0   4.0   5.0 |
    | 8.0  10.0  13.0 |
A = |  .     .     .  |
    |  .     .     .  |
    |  .     .     .  |
    |  .     .     .  |
    |  .     .     .  |
    |  .     .     .  |
```

Example 2

This example shows a matrix, ***A***, contained in a larger array, A. The strides of vectors ***x*** and ***y*** are of opposite sign. For ***y***, which has negative stride, processing begins at element `Y(5)`, which is 1.0. Array A must follow the same rules as given in Example 1. For this example, array A is declared as `A(-1:8,1:3)`.

Call Statement and Input:

```
           M   N  ALPHA  X  INCX  Y   INCY     A     LDA
           |   |    |    |   |    |    |       |      |
CALL SGER( 4 , 3 , 1.0 , X , 1   , Y , -2  , A(-1,1) , 10 )
```

`X = (3.0, 2.0, 1.0, 4.0)`
`Y = (3.0, . , 2.0, . , 1.0)`
A =(same as input A in Example 1)

Output:

A =(same as input A in Example 1)

Example 3

This example shows a matrix, ***A***, contained in a larger array, A, and the first element of the matrix is not the first element of the array. Array A must follow the same rules as given in Example 1. For this example, array A is declared as `A(1:10,1:3)`.

Call Statement and Input:

```
           M   N  ALPHA  X  INCX  Y  INCY    A     LDA
           |   |    |    |   |    |   |      |      |
CALL SGER( 4 , 3 , 1.0 , X , 3   , Y , 1  , A(4,1) , 10 )

X        =  (3.0, . , . , 2.0, . , . , 1.0, . , . , 4.0)
Y        =  (1.0, 2.0, 3.0)
```

```
    | .    .    .   |
    | .    .    .   |
    | .    .    .   |
    | 1.0  2.0  3.0 |
A = | 2.0  2.0  4.0 |
    | 3.0  2.0  2.0 |
    | 4.0  2.0  1.0 |
    |  .    .    .  |
    |  .    .    .  |
    |  .    .    .  |
```

Output:

```
    | .     .     .   |
    | .     .     .   |
    | .     .     .   |
    | 4.0   8.0  12.0 |
A = | 4.0   6.0  10.0 |
    | 4.0   4.0   5.0 |
    | 8.0  10.0  13.0 |
    |  .     .     .  |
    |  .     .     .  |
    |  .     .     .  |
```

Example 4

This example shows a matrix, ***A***, and array, A, having the same number of rows. For this case, *m* and *lda* are equal. Because *lda* is 4 and *n* is 3, array A must be declared as `A(E1:E2,F1:F2)`, where `E2-E1+1`=4 and `F2-F1+1 ≥ 3.` For this example, array A is declared as `A(1:4,0:2)`.

Call Statement and Input:

```
           M   N  ALPHA  X  INCX  Y  INCY    A     LDA
           |   |    |    |   |    |   |      |      |
CALL SGER( 4 , 3 , 1.0 , X , 1  , Y , 1  , A(1,0) , 4 )

X        =  (3.0, 2.0, 1.0, 4.0)
Y        =  (1.0, 2.0, 3.0)
```

```
    | 1.0  2.0  3.0 |
A = | 2.0  2.0  4.0 |
    | 3.0  2.0  2.0 |
    | 4.0  2.0  1.0 |
```

Output:

```
    | 4.0   8.0  12.0 |
A = | 4.0   6.0  10.0 |
    | 4.0   4.0   5.0 |
    | 8.0  10.0  13.0 |
```

Example 5

This example shows a computation in which scalar value for *alpha* is greater than 1. Array A must follow the same rules as given in Example 4. For this example, array A is declared as `A(-1:2,1:3)`.

Call Statement and Input:

```
           M   N  ALPHA  X  INCX  Y  INCY     A     LDA
           |   |    |    |   |    |   |       |      |
CALL SGER( 4 , 3 , 2.0 , X , 1  , Y , 1  , A(-1,1) , 4 )
```

`X = (3.0, 2.0, 1.0, 4.0)`
`Y = (1.0, 2.0, 3.0)`
A =(same as input A in Example 4)

Output:

```
    |  7.0  14.0  21.0 |
A = |  6.0  10.0  16.0 |
    |  5.0   6.0   8.0 |
    | 12.0  18.0  25.0 |
```

Example 6

This example shows a rank-one update in which all data items contain complex numbers, and the transpose ***y***^{T} is used in the computation. Matrix ***A*** is contained in a larger array, A. The strides of vectors ***x*** and ***y*** are positive. The Fortran DIMENSION statement for array A must follow the same rules as given in Example 1. For this example, array A is declared as `A(1:10,0:2)`.

Call Statement and Input:

```
            M   N   ALPHA   X  INCX  Y  INCY    A     LDA
            |   |     |     |   |    |   |      |      |
CALL CGERU( 5 , 3 , ALPHA , X , 1  , Y , 1  , A(1,0) , 10 )

ALPHA    =  (1.0, 0.0)
X        =  ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0), (3.0, 4.0),
             (2.0, 0.0))
Y        =  ((1.0, 2.0), (4.0, 0.0), (1.0, -1.0))
```

```
    | (1.0, 2.0)  (3.0, 5.0)  (2.0, 0.0) |
    | (2.0, 3.0)  (7.0, 9.0)  (4.0, 8.0) |
    | (7.0, 4.0)  (1.0, 4.0)  (6.0, 0.0) |
    | (8.0, 2.0)  (2.0, 5.0)  (8.0, 0.0) |
A = | (9.0, 1.0)  (3.0, 6.0)  (1.0, 0.0) |
    |     .           .           .      |
    |     .           .           .      |
    |     .           .           .      |
    |     .           .           .      |
    |     .           .           .      |
```

Output:

```
    | (-2.0,   6.0)   (7.0,  13.0)   (5.0,   1.0) |
    |  (6.0,  11.0)  (23.0,   9.0)   (8.0,   4.0) |
    |  (6.0,   7.0)   (5.0,   8.0)   (8.0,   0.0) |
    |  (3.0,  12.0)  (14.0,  21.0)  (15.0,   1.0) |
A = | (11.0,   5.0)  (11.0,   6.0)   (3.0,  -2.0) |
    |      .              .              .        |
    |      .              .              .        |
    |      .              .              .        |
    |      .              .              .        |
    |      .              .              .        |
```

Example 7

This example shows a rank-one update in which all data items contain complex numbers, and the conjugate transpose ***y***^{H} is used in the computation. Matrix ***A*** is contained in a larger array, A. The strides of vectors ***x*** and ***y*** are positive. The Fortran DIMENSION statement for array A must follow the same rules as given in Example 1. For this example, array A is declared as `A(1:10,0:2)`.

Call Statement and Input:

```
            M   N   ALPHA   X  INCX  Y  INCY    A     LDA
            |   |     |     |   |    |   |      |      |
CALL CGERC( 5 , 3 , ALPHA , X , 1  , Y , 1  , A(1,0) , 10 )
```

`ALPHA = (1.0, 0.0)`
`X = ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0), (3.0, 4.0),`
`(2.0, 0.0))`
`Y = ((1.0, 2.0), (4.0, 0.0), (1.0, -1.0))`
A =(same as input A in Example 6)

Output:

```
    |  (6.0,   2.0)   (7.0,  13.0)  (1.0,   3.0) |
    |  (6.0,  -5.0)  (23.0,   9.0)  (8.0,  12.0) |
    | (10.0,   3.0)   (5.0,   8.0)  (6.0,   2.0) |
    | (19.0,   0.0)  (14.0,  21.0)  (7.0,   7.0) |
A = | (11.0,  -3.0)  (11.0,   6.0)  (3.0,   2.0) |
        |       .               .              .           |
        |       .               .              .           |
        |       .               .              .           |
        |       .               .              .           |
        |       .               .              .           |
```

SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)

Purpose

SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, and ZHEMV compute the matrix-vector product for either a real symmetric matrix or a complex Hermitian matrix, using the scalars α and β, matrix ***A***, and vectors ***x*** and ***y***:

$\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A} \boldsymbol{x}$

SSLMX and DSLMX compute the matrix-vector product for a real symmetric matrix, using the scalar α, matrix ***A***, and vectors ***x*** and ***y***:

$\boldsymbol{y} \leftarrow \boldsymbol{y} + \alpha \boldsymbol{A} \boldsymbol{x}$

The following storage modes are used:

- For SSPMV, DSPMV, CHPMV, and ZHPMV, matrix ***A*** is stored in upper- or lower-packed storage mode.
- For SSYMV, DSYMV, CHEMV, and ZHEMV, matrix ***A*** is stored in upper or lower storage mode.
- For SSLMX and DSLMX, matrix ***A*** is stored in lower-packed storage mode.

Table 93: Data Types.

Data Types	
α, β, ***A***, ***x***, ***y***	**Subprogram**
Short-precision real	SSPMV, SSYMV, and SSLMX
Long-precision real	DSPMV, DSYMV, and DSLMX
Short-precision complex	CHPMV and CHEMV
Long-precision complex	ZHPMV and ZHEMV

Note:

1. SSPMV and DSPMV are Level 2 BLAS subroutines. You should use these subroutines instead of SSLMX and DSLMX, which are provided only for compatibility with earlier releases of ESSL.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SSPMV | DSPMV | CHPMV | ZHPMV (*uplo*, *n*, *alpha*, *ap*, *x*, *incx*, *beta*, *y*, *incy*) CALL SSYMV | DSYMV | CHEMV | ZHEMV (*uplo*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*) CALL SSLMX | DSLMX (*n*, *alpha*, *ap*, *x*, *incx*, *y*, *incy*)

Language	Syntax

C and C++	sspmv | dspmv | chpmv | zhpmv (*uplo*, *n*, *alpha*, *ap*, *x*, *incx*, *beta*, *y*, *incy*);
	ssymv | dsymv | chemv | zhemv (*uplo*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*);
	sslmx | dslmx (*n*, *alpha*, *ap*, *x*, *incx*, *y*, *incy*);

CBLAS	cblas_sspmv | cblas_dspmv | cblas_chpmv | cblas_zhpmv (*cblas_layout*, *cblas_uplo*, *n*, *alpha*, *ap*, *x*, *incx*, *beta*, *y*, *incy*);
	cblas_ssymv | cblas_dsymv | cblas_chemv | cblas_zhemv (*cblas_layout*, *cblas_uplo*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*);

On Entry

cblas_layout

indicates whether the input matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

uplo

indicates the storage mode used for matrix ***A***, where:

If *uplo* = 'U', ***A*** is stored in upper-packed or upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower-packed or lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

cblas_uplo

indicates the storage mode used for matrix ***A***, where:

If *cblas_uplo* = CblasUpper, ***A*** is stored in upper-packed or upper storage mode.

If *cblas_uplo* = CblasLower, ***A*** is stored in lower-packed or lower storage mode.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

n

is the number of elements in vectors ***x*** and ***y*** and the order of matrix ***A***.

Specified as: an integer; $n \geq 0$.

alpha

is the scaling constant α.

Specified as: a number of the data type indicated in Table 93 on page 313.

ap

has the following meaning:

For SSPMV and DSPMV, *ap* is the real symmetric matrix ***A*** of order *n*, stored in upper- or lower-packed storage mode.

For CHPMV and ZHPMV, *ap* is the complex Hermitian matrix ***A*** of order *n*, stored in upper- or lower-packed storage mode.

For SSLMX and DSLMX, *ap* is the real symmetric matrix ***A*** of order *n*, stored in lower-packed storage mode.

Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 93 on page 313.

a
has the following meaning:

For SSYMV and DSYMV, *a* is the real symmetric matrix ***A*** of order *n*, stored in upper or lower storage mode.

For CHEMV and ZHEMV, *a* is the complex Hermitian matrix ***A*** of order *n*, stored in upper or lower storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 93 on page 313.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

x
is the vector ***x*** of length *n*.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 93 on page 313.

incx
is the stride for vector ***x***.

Specified as: an integer, where:

For SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, and ZHEMV, *incx* < 0 or *incx* > 0.

For SSLMX and DSLMX, *incx* can have any value.

beta
is the scaling constant β.

Specified as: a number of the data type indicated in Table 93 on page 313.

y
is the vector ***y*** of length *n*.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incy*|, containing numbers of the data type indicated in Table 93 on page 313.

incy
is the stride for vector ***y***.

Specified as: an integer; *incy* > 0 or *incy* < 0.

On Return

y
is the vector ***y*** of length *n*, containing the result of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 93 on page 313.

Notes

1. All subroutines accept lowercase letters for the *uplo* argument.
2. The vector ***y*** must have no common elements with vector ***x*** or matrix ***A***; otherwise, results are unpredictable. See "Vector concepts" on page 71.
3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.
4. For a description of how symmetric matrices are stored in upper- or lower-packed storage mode and upper or lower storage mode, see "Symmetric Matrix" on page 80. For a description of how complex Hermitian matrices are stored in upper- or lower-packed storage mode and upper or lower storage mode, see "Complex Hermitian Matrix" on page 84.

Function

These subroutines perform the calculations described. See references [“44” on page 1255], [“45” on page 1255], and [“95” on page 1258].

For SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, and ZHEMV
If n is zero or if α is zero and β is one, no computation is performed.

For SSLMX and DSLMX
If n or α is zero, no computation is performed.

For SSLMX, SSPMV, SSYMV, CHPMV, and CHEMV
Intermediate results are accumulated in long precision when the VSX unit is not used. However, several intermediate stores may occur for each element of the vector ***y***.

For SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, and ZHEMV

These subroutines compute the matrix-vector product for either a real symmetric matrix or a complex Hermitian matrix:

$\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A}\boldsymbol{x}$

where:

y is a vector of length n.
α is a scalar.
β is a scalar.
A is a real symmetric or complex Hermitian matrix of order n.
x is a vector of length n.

It is expressed as follows:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} \leftarrow \beta \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} a_{11} & \dots a_{1n} \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ a_{n1} & \dots a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

For SSLMX and DSLMX

These subroutines compute the matrix-vector product for a real symmetric matrix stored in lower-packed storage mode:

$\boldsymbol{y} \leftarrow \boldsymbol{y} + \alpha \boldsymbol{A}\boldsymbol{x}$

where:

y is a vector of length n.
α is a scalar.
A is a real symmetric matrix of order n.
x is a vector of length n.

It is expressed as follows:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} \leftarrow \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. *uplo* ≠ 'L' or 'U'
3. *cblas_uplo* ≠ CblasLower or CblasUpper
4. $n < 0$
5. $lda < n$
6. $lda \le 0$
7. $incx = 0$
8. $incy = 0$

Examples

Example 1
This example shows vectors ***x*** and ***y*** with positive strides and a real symmetric matrix ***A*** of order 3, stored in lower-packed storage mode. Matrix ***A*** is:

```
                        ┌                ┐
                        | 8.0  4.0  2.0 |
                        | 4.0  6.0  7.0 |
                        | 2.0  7.0  3.0 |
                        └                ┘
```

Call Statement and Input:

```
            UPLO  N  ALPHA  AP   X  INCX BETA  Y  INCY
             |    |    |     |   |   |    |    |   |
CALL SSPMV( 'L' , 3 , 1.0 , AP , X , 1 , 1.0 , Y , 2 )

AP       =  (8.0, 4.0, 2.0, 6.0, 7.0, 3.0)
X        =  (3.0, 2.0, 1.0)
Y        =  (5.0, . , 3.0, . , 2.0)
```

Output:

```
Y        =  (39.0, . , 34.0, . , 25.0)
```

Example 2
This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element `X(5)`, which is 1.0. The real symmetric matrix ***A*** of order 3 is stored in upper-packed storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input:

```
             UPLO  N  ALPHA  AP   X  INCX  BETA  Y  INCY
              |    |    |     |   |    |    |    |   |
CALL SSPMV( 'U' , 3 , 1.0 , AP , X , -2 , 2.0 , Y , 1 )

AP       =  (8.0, 4.0, 6.0, 2.0, 7.0, 3.0)
X        =  (4.0, . , 2.0, . , 1.0)
Y        =  (6.0, 5.0, 4.0)
```

Output:

```
Y        =  (36.0, 54.0, 36.0)
```

Example 3

This example shows vector ***x*** and ***y*** with positive stride and a complex Hermitian matrix ***A*** of order 3, stored in lower-packed storage mode. Matrix ***A*** is:

```
                                                 ┐
             |  (1.0, 0.0)  (3.0, 5.0)  (2.0, -3.0) |
             | (3.0, -5.0)  (7.0, 0.0)  (4.0, -8.0) |
             |  (2.0, 3.0)  (4.0, 8.0)   (6.0, 0.0) |
                                                 ┘
```

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.

Call Statement and Input:

```
             UPLO  N   ALPHA   AP   X  INCX BETA   Y  INCY
              |    |     |      |   |   |    |     |   |
CALL CHPMV( 'L' , 3 , ALPHA , AP , X , 1 , BETA , Y , 2 )
```

```
ALPHA    =  (1.0, 0.0)
AP       =  ((1.0, . ), (3.0, -5.0), (2.0, 3.0), (7.0, . ),
             (4.0, 8.0), (6.0, . ))
X        =  ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0))
BETA     =  (1.0, 0.0)
Y        =  ((1.0, 0.0), . , (2.0, -1.0), . , (2.0, 1.0))
```

Output:

```
Y        =  ((32.0, 21.0), . , (87.0, -8.0), . , (32.0, 64.0))
```

Example 4

This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element `X(5)`, which is (1.0, 2.0). The complex Hermitian matrix ***A*** of order 3 is stored in upper-packed storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.

Call Statement and Input:

```
             UPLO  N   ALPHA   AP   X  INCX  BETA   Y INCY
              |    |     |     |    |    |    |     |   |
CALL CHPMV( 'U' , 3 , ALPHA , AP , X , -2 , BETA , Y , 2 )
```

```
ALPHA = (1.0, 0.0)
AP = ((1.0, . ), (3.0, 5.0), (7.0, . ), (2.0, -3.0),
(4.0, -8.0), (6.0, . ))
X = ((3.0, 4.0), . , (4.0, 0.0), . , (1.0, 2.0))
BETA = (0.0, 0.0)
```
Y =(not relevant)

Output:

```
Y        =  ((31.0, 21.0), . , (85.0, -7.0), . , (30.0, 63.0))
```

Example 5

This example shows vectors ***x*** and ***y*** with positive strides and a real symmetric matrix ***A*** of order 3, stored in lower storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input:

```
            UPLO  N  ALPHA  A  LDA  X  INCX BETA  Y INCY
             |    |    |    |   |   |   |    |    |   |
CALL SSYMV( 'L' , 3 , 1.0 , A , 3 , X , 1 , 1.0 , Y , 2 )

                    ┐
    | 8.0  .    .   |
A = | 4.0  6.0  .   |
    | 2.0  7.0  3.0 |
                    ┘

X        =  (3.0, 2.0, 1.0)
Y        =  (5.0, . , 3.0, . , 2.0)
```

Output:

```
Y        =  (39.0, . , 34.0, . , 25.0)
```

Example 6

This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element `X(5)`, which is 1.0. The real symmetric matrix ***A*** of order 3 is stored in upper storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input:

```
            UPLO  N  ALPHA  A  LDA  X  INCX  BETA  Y  INCY
             |    |    |    |   |   |    |    |    |   |
CALL SSYMV( 'U' , 3 , 1.0 , A , 4 , X , -2 , 2.0 , Y , 1 )

                    ┐
    | 8.0  4.0  2.0 |
A = |  .   6.0  7.0 |
    |  .    .   3.0 |
    |  .    .    .  |
                    ┘

X        =  (4.0, . , 2.0, . , 1.0)
Y        =  (6.0, 5.0, 4.0)
```

Output:

```
A        =  (36.0, 54.0, 36.0)
```

Example 7

This example shows vector ***x*** and ***y*** with positive stride and a complex Hermitian matrix ***A*** of order 3, stored in lower storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.

Call Statement and Input:

```
             UPLO  N   ALPHA   A  LDA  X  INCX BETA   Y INCY
              |    |     |     |   |   |   |    |     |   |
CALL CHEMV( 'L' , 3 , ALPHA , A , 3 , X , 1 , BETA , Y , 2 )

ALPHA    =  (1.0, 0.0)
```

```
                                                 ┐
        | (1.0,  . )       .          .           |
A    =  | (3.0, -5.0)  (7.0, .  )      .          |
        | (2.0,  3.0)  (4.0, 8.0) (6.0,  . )      |
                                                 ┘
```

```
X        =  ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0))
BETA     =  (1.0, 0.0)
Y        =  ((1.0, 0.0), . , (2.0, -1.0), . , (2.0, 1.0))
```

Output:

```
Y        =  ((32.0, 21.0), . , (87.0, -8.0), . , (32.0, 64.0))
```

Example 8

This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element `X(5)`, which is (1.0, 2.0). The complex Hermitian matrix ***A*** of order 3 is stored in upper storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.

Call Statement and Input:

```
             UPLO  N   ALPHA   A  LDA  X  INCX  BETA   Y INCY
              |    |     |     |   |   |    |    |     |   |
CALL CHEMV( 'U' , 3 , ALPHA , A , 3 , X , -2 , BETA , Y , 2 )

ALPHA    =  (1.0, 0.0)
```

```
        ┌                                      ┐
        | (1.0, . )  (3.0, 5.0)  (2.0, -3.0) |
A    =  |     .      (7.0,  . )  (4.0, -8.0) |
        |     .          .       (6.0,   . ) |
                                             ┘
```

`X = ((3.0, 4.0), . , (4.0, 0.0), . , (1.0, 2.0))`
`BETA = (0.0, 0.0)`
Y =(not relevant)

Output:

```
Y        =  ((31.0, 21.0), . , (85.0, -7.0), . , (30.0, 63.0))
```

Example 9

This example shows vectors ***x*** and ***y*** with positive strides and a real symmetric matrix ***A*** of order 3. Matrix ***A*** is:

```
                        ┌              ┐
                        | 8.0  4.0  2.0 |
                        | 4.0  6.0  7.0 |
                        | 2.0  7.0  3.0 |
                                       ┘
```

Call Statement and Input:

```
             N  ALPHA  AP   X  INCX  Y  INCY
             |    |    |    |   |    |   |
CALL SSLMX( 3 , 1.0 , AP , X , 1  , Y , 2  )
```

```
AP       =  (8.0, 4.0, 2.0, 6.0, 7.0, 3.0)
X        =  (3.0, 2.0, 1.0)
Y        =  (5.0, . , 3.0, . , 2.0)
```

Output:

```
Y        =  (39.0, . , 34.0, . , 25.0)
```

SSPR, DSPR, CHPR, ZHPR, SSYR, DSYR, CHER, ZHER, SSLR1, and DSLR1 (Rank-One Update of a Real Symmetric or Complex Hermitian Matrix)

Purpose

SSPR, DSPR, SSYR, DSYR, SSLR1, and DSLR1 compute the rank-one update of a real symmetric matrix, using the scalar α, matrix ***A***, vector ***x***, and its transpose $\boldsymbol{x}^T$:

$\boldsymbol{A} \leftarrow \boldsymbol{A} + \alpha \boldsymbol{x}\boldsymbol{x}^T$

CHPR, ZHPR, CHER, and ZHER compute the rank-one update of a complex Hermitian matrix, using the scalar α, matrix ***A***, vector ***x***, and its conjugate transpose $\boldsymbol{x}^H$:

$\boldsymbol{A} \leftarrow \boldsymbol{A} + \alpha \boldsymbol{x}\boldsymbol{x}^H$

The following storage modes are used:

- For SSPR, DSPR, CHPR, and ZHPR, matrix ***A*** is stored in upper- or lower-packed storage mode.
- For SSYR, DSYR, CHER, and ZHER, matrix ***A*** is stored in upper or lower storage mode.
- For SSLR1 and DSLR1, matrix ***A*** is stored in lower-packed storage mode.

Table 94: Data Types.

Data Types		
A, ***x***	α	**Subprogram**
Short-precision real	Short-precision real	SSPR, SSYR, and SSLR1
Long-precision real	Long-precision real	DSPR, DSYR, and DSLR1
Short-precision complex	Short-precision real	CHPR and CHER
Long-precision complex	Long-precision real	ZHPR and ZHER

Note:

1. SSPR and DSPR are Level 2 BLAS subroutines. You should use these subroutines instead of SSLR1 and DSLR1, which are only provided for compatibility with earlier releases of ESSL.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SSPR | DSPR | CHPR | ZHPR (*uplo*, *n*, *alpha*, *x*, *incx*, *ap*) CALL SSYR | DSYR | CHER | ZHER (*uplo*, *n*, *alpha*, *x*, *incx*, *a*, *lda*) CALL SSLR1 | DSLR1 (*n*, *alpha*, *x*, *incx*, *ap*)

Language	Syntax

C and C++	sspr | dspr | chpr | zhpr (*uplo*, *n*, *alpha*, *x*, *incx*, *ap*);
	ssyr | dsyr | cher | zher (*uplo*, *n*, *alpha*, *x*, *incx*, *a*, *lda*);
	sslr1 | dslr1 (*n*, *alpha*, *x*, *incx*, *ap*);

CBLAS	cblas_sspr | cblas_dspr | cblas_chpr | cblas_zhpr (*cblas_layout*, *cblas_uplo*, *n*, *alpha*, *x*, *incx*, *ap*);
	cblas_ssyr | cblas_dsyr | cblas_cher | cblas_zher (*cblas_layout*, *cblas_uplo*, *n*, *alpha*, *x*, *incx*, *a*, *lda*);

On Entry

cblas_layout
indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

uplo
indicates the storage mode used for matrix ***A***, where:

If *uplo* = 'U', ***A*** is stored in upper-packed or upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower-packed or lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

cblas_uplo
indicates the storage mode used for matrix ***A***, where:

If *cblas_uplo* = CblasUpper, ***A*** is stored in upper-packed or upper storage mode.

If *cblas_uplo* = CblasLower, ***A*** is stored in lower-packed or lower storage mode.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

n
is the number of elements in vector ***x*** and the order of matrix ***A***.

Specified as: an integer; $n \geq 0$.

alpha
is the scaling constant α.

Specified as: a number of the data type indicated in Table 94 on page 321.

x
is the vector ***x*** of length *n*.

Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 94 on page 321.

incx
is the stride for vector ***x***.

Specified as: an integer, where:

For SSPR, DSPR, CHPR, ZHPR, SSYR, DSYR, CHER, and ZHER, *incx* < 0 or *incx* > 0.

For SSLR1 and DSLR1, *incx* can have any value.

ap

has the following meaning:

For SSPR and DSPR, *ap* is the real symmetric matrix ***A*** of order *n*, stored in upper- or lower-packed storage mode.

For CHPR and ZHPR, *ap* is the complex Hermitian matrix ***A*** of order *n*, stored in upper- or lower-packed storage mode.

For SSLR1 and DSLR1, *ap* is the real symmetric matrix ***A*** of order *n*, stored in lower-packed storage mode.

Specified as: a one-dimensional array of (at least) length *n*(*n*+1)/2, containing numbers of the data type indicated in Table 94 on page 321.

a

has the following meaning:

For SSYR and DSYR, *a* is the real symmetric matrix ***A*** of order *n*, stored in upper or lower storage mode.

For CHER and ZHER, *a* is the complex Hermitian matrix ***A*** of order *n*, stored in upper or lower storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 94 on page 321.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

On Return

ap

is the matrix ***A*** of order *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 94 on page 321.

a

is the matrix ***A*** of order *n*, containing the results of the computation. Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 94 on page 321.

Notes

1. All subroutines accept lowercase letters for the *uplo* argument.
2. The vector ***x*** must have no common elements with matrix ***A***; otherwise, results are unpredictable. See "Vector concepts" on page 71.
3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if α ≠ 0.0, they are set to zero.
4. For a description of how symmetric matrices are stored in upper- or lower-packed storage mode and upper or lower storage mode, see "Symmetric Matrix" on page 80. For a description of how complex Hermitian matrices are stored in upper- or lower-packed storage mode and upper or lower storage mode, see "Complex Hermitian Matrix" on page 84.

Function

These subroutines perform the computations described. See references ["44" on page 1255], ["45" on page 1255], and ["95" on page 1258].

Note: If *n* or α is 0, no computation is performed.

For CHPR and CHER

Intermediate results are accumulated in long precision when the VSX unit is not used.

For SSPR, SSYR, and SSLR1
Intermediate results are accumulated in long precision on some platforms when the VSX unit is not used.

For SSPR, DSPR, SSYR, DSYR, SSLR1, and DSLR1

These subroutines compute the rank-one update of a real symmetric matrix:

$\boldsymbol{A} \leftarrow \boldsymbol{A} + \alpha \boldsymbol{x}\boldsymbol{x}^T$

where:

$\boldsymbol{A}$ is a real symmetric matrix of order n.
α is a scalar.
$\boldsymbol{x}$ is a vector of length n.
$\boldsymbol{x}^T$ is the transpose of vector $\boldsymbol{x}$.

It is expressed as follows:

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{n1} & \dots & a_{nn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix}$$

For CHPR, ZHPR, CHER, and ZHER

These subroutines compute the rank-one update of a complex Hermitian matrix:

$\boldsymbol{A} \leftarrow \boldsymbol{A} + \alpha \boldsymbol{x}\boldsymbol{x}^H$

where:

$\boldsymbol{A}$ is a complex Hermitian matrix of order n.
α is a scalar.
$\boldsymbol{x}$ is a vector of length n.
$\boldsymbol{x}^H$ is the conjugate transpose of vector $\boldsymbol{x}$.

It is expressed as follows:

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{n1} & \dots & a_{nn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} \begin{bmatrix} \bar{x}_1 & \dots & \bar{x}_n \end{bmatrix}$$

Error conditions

Computational Errors
None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. *uplo* ≠ 'L' or 'U'

3. *cblas_uplo* ≠ CblasLower or CblasUpper
4. $n < 0$
5. *incx* = 0
6. *lda* ≤ 0
7. *lda* < *n*

Examples

Example 1

This example shows a vector ***x*** with a positive stride, and a real symmetric matrix ***A*** of order 3, stored in lower-packed storage mode. Matrix ***A*** is:

```
                        | 8.0  4.0  2.0 |
                        | 4.0  6.0  7.0 |
                        | 2.0  7.0  3.0 |
```

Call Statement and Input:

```
          UPLO   N  ALPHA  X  INCX  AP
            |    |    |    |   |    |
CALL SSPR( 'L' , 3 , 1.0 , X , 1  , AP )

X        =  (3.0, 2.0, 1.0)
AP       =  (8.0, 4.0, 2.0, 6.0, 7.0, 3.0)
```

Output:

```
AP       =  (17.0, 10.0, 5.0, 10.0, 9.0, 4.0)
```

Example 2

This example shows a vector ***x*** with a negative stride, and a real symmetric matrix ***A*** of order 3, stored in upper-packed storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input:

```
          UPLO   N  ALPHA  X  INCX  AP
            |    |    |    |    |   |
CALL SSPR( 'U' , 3 , 1.0 , X , -2 , AP )

X        =  (1.0, . , 2.0, . , 3.0)
AP       =  (8.0, 4.0, 6.0, 2.0, 7.0, 3.0)
```

Output:

```
AP       =  (17.0, 10.0, 10.0, 5.0, 9.0, 4.0)
```

Example 3

This example shows a vector ***x*** with a positive stride, and a complex Hermitian matrix ***A*** of order 3, stored in lower-packed storage mode. Matrix ***A*** is:

```
               |  (1.0, 0.0)  (3.0, 5.0)  (2.0, -3.0) |
               | (3.0, -5.0)  (7.0, 0.0)  (4.0, -8.0) |
               |  (2.0, 3.0)  (4.0, 8.0)   (6.0, 0.0) |
```

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if α ≠ 0.0, they are set to zero.

Call Statement and Input:

```
           UPLO  N  ALPHA  X  INCX AP
            |    |    |    |   |   |
CALL CHPR( 'L' , 3 , 1.0 , X , 1 , AP )
```

```
X        =  ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0))
AP       =  ((1.0, . ), (3.0, -5.0), (2.0, 3.0), (7.0, . ),
             (4.0, 8.0), (6.0, . ))
```

Output:

```
AP       =  ((6.0, 0.0), (7.0, -13.0), (13.0, 1.0), (23.0, 0.0),
             (16.0, 24.0), (31.0, 0.0))
```

Example 4

This example shows a vector ***x*** with a negative stride, and a complex Hermitian matrix ***A*** of order 3, stored in upper-packed storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if α ≠ 0.0, they are set to zero.

Call Statement and Input:

```
           UPLO  N  ALPHA  X  INCX  AP
             |   |    |    |    |   |
CALL CHPR( 'U' , 3 , 1.0 , X , -2 , AP )

X        =  ((3.0, 4.0), . , (4.0, 0.0), . , (1.0, 2.0))
AP       =  ((1.0, . ), (3.0, 5.0), (7.0, . ), (2.0, -3.0),
             (4.0, -8.0), (6.0, . ))
```

Output:

```
AP       =  ((6.0, 0.0), (7.0, 13.0), (23.0, 0.0), (13.0, -1.0),
             (16.0, -24.0), (31.0, 0.0))
```

Example 5

This example shows a vector ***x*** with a positive stride, and a real symmetric matrix ***A*** of order 3, stored in lower storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input:

```
           UPLO  N  ALPHA  X  INCX A  LDA
             |   |    |    |   |   |   |
CALL SSYR( 'L' , 3 , 1.0 , X , 1 , A , 3 )

X        =  (3.0, 2.0, 1.0)


        | 8.0   .    .  |
A    =  | 4.0  6.0   .  |
        | 2.0  7.0  3.0 |
```

Output:

```
        | 17.0    .    .  |
A    =  | 10.0  10.0   .  |
        |  5.0   9.0  4.0 |
```

Example 6

This example shows a vector ***x*** with a negative stride, and a real symmetric matrix ***A*** of order 3, stored in upper storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input:

```
           UPLO  N  ALPHA  X  INCX  A  LDA
             |   |    |    |    |   |   |
CALL SSYR( 'U' , 3 , 1.0 , X , -2 , A , 4 )

X        =  (1.0, . , 2.0, . , 3.0)


        | 8.0  4.0  2.0 |
A    =  |  .   6.0  7.0 |
        |  .    .   3.0 |
        |  .    .    .  |
```

Output:

```
        | 17.0  10.0  5.0 |
A    =  |   .   10.0  9.0 |
        |   .     .   4.0 |
        |   .     .    .  |
```

Example 7

This example shows a vector ***x*** with a positive stride, and a complex Hermitian matrix ***A*** of order 3, stored in lower storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if α ≠ 0.0, they are set to zero.

Call Statement and Input:

```
           UPLO  N  ALPHA  X  INCX A  LDA
             |   |    |    |   |   |   |
CALL CHER( 'L' , 3 , 1.0 , X , 1 , A , 3 )

X        =  ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0))


        |   (1.0, . )        .            .   |
A    =  | (3.0, -5.0)   (7.0, . )         .   |
        |  (2.0, 3.0)  (4.0, 8.0)  (6.0, . ) |
```

Output:

```
        |   (6.0, 0.0)        .            .      |
A    =  | (7.0, -13.0)   (23.0, 0.0)       .      |
        |  (13.0, 1.0)  (16.0, 24.0)  (31.0, 0.0) |
```

This example shows a vector ***x*** with a negative stride, and a complex Hermitian matrix ***A*** of order 3, stored in upper storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if α ≠ 0.0, they are set to zero.

Call Statement and Input:

```
           UPLO  N  ALPHA  X  INCX  A  LDA
             |   |    |    |    |   |   |
CALL CHER( 'U' , 3 , 1.0 , X , -2 , A , 3 )

X        =  ((3.0, 4.0), . , (4.0, 0.0), . , (1.0, 2.0))


        | (1.0, . )  (3.0, 5.0)  (2.0, -3.0) |
A    =  |     .      (7.0,  . )  (4.0, -8.0) |
        |     .           .        (6.0, . ) |
```

Output:

```
        | (6.0, 0.0)  (7.0, 13.0)   (13.0, -1.0) |
A    =  |     .       (23.0, 0.0)  (16.0, -24.0) |
        |     .            .         (31.0, 0.0) |
```

Example 9

This example shows a vector $\boldsymbol{x}$ with a positive stride, and a real symmetric matrix $\boldsymbol{A}$ of order 3, stored in lower-packed storage mode. It uses the same input matrix $\boldsymbol{A}$ as in Example 1.

Call Statement and Input:

```
             N  ALPHA  X  INCX  AP
             |    |    |   |     |
CALL SSLR1( 3 , 1.0 , X , 1  , AP )

X        =  (3.0, 2.0, 1.0)
AP       =  (8.0, 4.0, 2.0, 6.0, 7.0, 3.0)
```

Output:

```
AP       =  (17.0, 10.0, 5.0, 10.0, 9.0, 4.0)
```

SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, ZHER2, SSLR2, and DSLR2 (Rank-Two Update of a Real Symmetric or Complex Hermitian Matrix)

Purpose

SSPR2, DSPR2, SSYR2, DSYR2, SSLR2, and DSLR2 compute the rank-two update of a real symmetric matrix, using the scalar α, matrix $\boldsymbol{A}$, vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, and their transposes $\boldsymbol{x}^T$ and $\boldsymbol{y}^T$:

$\boldsymbol{A} \leftarrow \boldsymbol{A}+\alpha\boldsymbol{x}\boldsymbol{y}^T + \alpha\boldsymbol{y}\boldsymbol{x}^T$

CHPR2, ZHPR2, CHER2, and ZHER2, compute the rank-two update of a complex Hermitian matrix, using the scalar α, matrix $\boldsymbol{A}$, vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, and their conjugate transposes $\boldsymbol{x}^H$ and $\boldsymbol{y}^H$:

$$\boldsymbol{A} \leftarrow \boldsymbol{A} + \alpha\boldsymbol{x}\boldsymbol{y}^H + \bar{\alpha}\boldsymbol{y}\boldsymbol{x}^H$$

The following storage modes are used:

- For SSPR2, DSPR2, CHPR2, and ZHPR2, matrix $\boldsymbol{A}$ is stored in upper- or lower-packed storage mode.
- For SSYR2, DSYR2, CHER2, and ZHER2, matrix $\boldsymbol{A}$ is stored in upper or lower storage mode.
- For SSLR2 and DSLR2, matrix $\boldsymbol{A}$ is stored in lower-packed storage mode.

Table 95: Data Types.

Data Types	
α, $\boldsymbol{A}$, $\boldsymbol{x}$, $\boldsymbol{y}$	**Subprogram**
Short-precision real	SSPR2, SSYR2, and SSLR2
Long-precision real	DSPR2, DSYR2, and DSLR2
Short-precision complex	CHPR2 and CHER2
Long-precision complex	ZHPR2 and ZHER2

Note:

1. SSPR2 and DSPR2 are Level 2 BLAS subroutines. You should use these subroutines instead of SSLR2 and DSLR2, which are only provided for compatibility with earlier releases of ESSL.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SSPR2 | DSPR2 | CHPR2 | ZHPR2 (*uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *ap*) CALL SSYR2 | DSYR2 | CHER2 | ZHER2 (*uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *a*, *lda*) CALL SSLR2 | DSLR2 (*n*, *alpha*, *x*, *incx*, *y*, *incy*, *ap*)

C and C++	sspr2 | dspr2 | chpr2 | zhpr2 (*uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *ap*);
	ssyr2 | dsyr2 | cher2 | zher2 (*uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *a*, *lda*);
	sslr2 | dslr2 (*n*, *alpha*, *x*, *incx*, *y*, *incy*, *ap*);

CBLAS	cblas_sspr2 | cblas_dspr2 | cblas_chpr2 | cblas_zhpr2 (*cblas_layout*, *cblas_uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *ap*);
	cblas_ssyr2 | cblas_dsyr2 | cblas_cher2 | cblas_zher2 (*cblas_layout*, *cblas_uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *a*, *lda*);

On Entry

cblas_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

uplo

indicates the storage mode used for matrix ***A***, where:

If *uplo* = 'U', ***A*** is stored in upper-packed or upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower-packed or lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

cblas_uplo

indicates the storage mode used for matrix ***A***, where:

If *cblas_uplo* = CblasUpper, ***A*** is stored in upper-packed or upper storage mode.

If *cblas_uplo* = CblasLower, ***A*** is stored in lower-packed or lower storage mode.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

n

is the number of elements in vectors ***x*** and ***y*** and the order of matrix ***A***.

Specified as: an integer; $n \geq 0$.

alpha

is the scaling constant α.

Specified as: a number of the data type indicated in Table 95 on page 328.

x

is the vector ***x*** of length *n*.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 95 on page 328.

incx

is the stride for vector ***x***.

Specified as: an integer, where:

For SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, and ZHER2, *incx* < 0 or *incx* > 0.

For SSLR2 and DSLR2, *incx* can have any value.

y

is the vector ***y*** of length *n*.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incy*|, containing numbers of the data type indicated in Table 95 on page 328.

incy

is the stride for vector ***y***.

Specified as: an integer, where:

For SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, and ZHER2, *incy* < 0 or *incy* > 0.

For SSLR2 and DSLR2, *incy* can have any value.

ap

has the following meaning:

For SSPR2 and DSPR2, *ap* is the real symmetric matrix ***A*** of order *n*, stored in upper- or lower-packed storage mode.

For CHPR2 and ZHPR2, *ap* is the complex Hermitian matrix ***A*** of order *n*, stored in upper- or lower-packed storage mode.

For SSLR2 and DSLR2, *ap* is the real symmetric matrix ***A*** of order *n*, stored in lower-packed storage mode.

Specified as: a one-dimensional array of (at least) length *n*(*n*+1)/2, containing numbers of the data type indicated in Table 95 on page 328.

a

has the following meaning:

For SSYR2 and DSYR2, *a* is the real symmetric matrix ***A*** of order *n*, stored in upper or lower storage mode.

For CHER2 and ZHER2, *a* is the complex Hermitian matrix ***A*** of order *n*, stored in upper or lower storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 95 on page 328.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

On Return

ap

is the matrix ***A*** of order *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 95 on page 328.

a

is the matrix ***A*** of order *n*, containing the results of the computation. Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 95 on page 328.

Notes

1. All subroutines accept lowercase letters for the *uplo* argument.
2. The vectors ***x*** and ***y*** must have no common elements with matrix ***A***; otherwise, results are unpredictable. See "Vector concepts" on page 71.

3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq$ zero, the imaginary parts of the diagonal elements are set to zero.
4. For a description of how symmetric matrices are stored in upper- or lower-packed storage mode and upper or lower storage mode, see "Symmetric Matrix" on page 80. For a description of how complex Hermitian matrices are stored in upper- or lower-packed storage mode and upper or lower storage mode, see "Complex Hermitian Matrix" on page 84.

Function

These subroutines perform the computation described. See references ["44" on page 1255], ["45" on page 1255], and ["95" on page 1258]. If *n* or α is zero, no computation is performed.

For SSPR2, SSYR2, SSLR2, CHPR2, and CHER2, intermediate results are accumulated in long precision when the VSX unit is not used.

SSPR2, DSPR2, SSYR2, DSYR2, SSLR2, and DSLR2

These subroutines compute the rank-two update of a real symmetric matrix:

$\boldsymbol{A} \leftarrow \boldsymbol{A} + \alpha \boldsymbol{x}\boldsymbol{y}^T + \alpha \boldsymbol{y}\boldsymbol{x}^T$

where:

A is a real symmetric matrix of order *n*.
α is a scalar.
x is a vector of length *n*.
$\boldsymbol{x}^T$ is the transpose of vector ***x***.
y is a vector of length *n*.
$\boldsymbol{y}^T$ is the transpose of vector ***y***.

It is expressed as follows:

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \dots & a_{nn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix} \begin{bmatrix} y_1 & \dots & y_n \end{bmatrix}$$

$$+ \alpha \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix}$$

CHPR2, ZHPR2, CHER2, and ZHER2

These subroutines compute the rank-two update of a complex Hermitian matrix:

$$\boldsymbol{A} \leftarrow \boldsymbol{A} + \alpha \boldsymbol{x}\boldsymbol{y}^H + \overline{\alpha} \boldsymbol{y}\boldsymbol{x}^H$$

where:

A is a complex Hermitian matrix of order *n*.
α is a scalar.
x is a vector of length *n*.
$\boldsymbol{x}^H$ is the conjugate transpose of vector ***x***.
y is a vector of length *n*.
$\boldsymbol{y}^H$ is the conjugate transpose of vector ***y***.

It is expressed as follows:

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{n1} & \dots & a_{nn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} \begin{bmatrix} \bar{y}_1 & \dots & \bar{y}_n \end{bmatrix}$$

$$+ \bar{\alpha} \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} \begin{bmatrix} \bar{x}_1 & \dots & \bar{x}_n \end{bmatrix}$$

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. *uplo* ≠ 'L' or 'U'
3. *cblas_uplo* ≠ CblasLower or CblasUpper
4. *n* < 0
5. *incx* = 0
6. *incy* = 0
7. *lda* ≤ 0
8. *lda* < *n*

Examples

Example 1
This example shows vectors ***x*** and ***y*** with positive strides and a real symmetric matrix ***A*** of order 3, stored in lower-packed storage mode. Matrix ***A*** is:

```
                        ┌                ┐
                        | 8.0  4.0  2.0 |
                        | 4.0  6.0  7.0 |
                        | 2.0  7.0  3.0 |
                                       ┘
```

Call Statement and Input:

```
            UPLO  N  ALPHA  X  INCX  Y  INCY AP
             |    |    |    |   |    |   |   |
CALL SSPR2( 'L' , 3 , 1.0 , X , 1 ,  Y , 2 , AP )

X        =  (3.0, 2.0, 1.0)
Y        =  (5.0, . , 3.0, . , 2.0)
AP       =  (8.0, 4.0, 2.0, 6.0, 7.0, 3.0)
```

Output:

```
AP       =  (38.0, 23.0, 13.0, 18.0, 14.0, 7.0)
```

Example 2

This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element `X(5)`, which is 3.0. The real symmetric matrix ***A*** of order 3 is stored in upper-packed storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input:

```
            UPLO  N  ALPHA  X   INCX Y  INCY AP
             |    |    |    |    |   |   |   |
CALL SSPR2( 'U' , 3 , 1.0 , X , -2 , Y , 2 , AP )

X        =  (1.0, . , 2.0, . , 3.0)
Y        =  (5.0, . , 3.0, . , 2.0)
AP       =  (8.0, 4.0, 6.0, 2.0, 7.0, 3.0)
```

Output:

```
AP       =  (38.0, 23.0, 18.0, 13.0, 14.0, 7.0)
```

Example 3

This example shows vector ***x*** and ***y*** with positive stride and a complex Hermitian matrix ***A*** of order 3, stored in lower-packed storage mode. Matrix ***A*** is:

```
                    ┌                                        ┐
                    |  (1.0, 0.0)  (3.0, 5.0)  (2.0, -3.0) |
                    | (3.0, -5.0)  (7.0, 0.0)  (4.0, -8.0) |
                    |  (2.0, 3.0)  (4.0, 8.0)   (6.0, 0.0) |
                    └                                        ┘
```

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if α ≠ zero, the imaginary parts of the diagonal elements are set to zero.

Call Statement and Input:

```
            UPLO  N   ALPHA   X  INCX Y  INCY AP
             |    |     |     |   |   |   |   |
CALL CHPR2( 'L' , 3 , ALPHA , X , 1 , Y , 2 , AP )

ALPHA    =  (1.0, 0.0)
X        =  ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0))
Y        =  ((1.0, 0.0), . , (2.0, -1.0), . , (2.0, 1.0))
AP       =  ((1.0, . ), (3.0, -5.0), (2.0, 3.0), (7.0, . ),
             (4.0, 8.0), (6.0, . ))
```

Output:

```
AP       =  ((3.0, 0.0), (7.0, -10.0), (9.0, 4.0), (23.0, 0.0),
             (14.0, 23.0), (26.0, 0.0))
```

Example 4

This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element X(5), which is (1.0,2.0). The complex Hermitian matrix ***A*** of order 3 is stored in upper-packed storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if α ≠ zero, the imaginary parts of the diagonal elements are set to zero.

Call Statement and Input:

```
            UPLO  N   ALPHA   X  INCX  Y  INCY AP
             |    |     |     |    |   |   |   |
CALL CHPR2( 'U' , 3 , ALPHA , X , -2 , Y , 2 , AP )

ALPHA    =  (1.0, 0.0)
X        =  ((3.0, 4.0), . , (4.0, 0.0), . , (1.0, 2.0))
Y        =  ((1.0, 0.0), . , (2.0, -1.0), . , (2.0, 1.0))
AP       =  ((1.0, . ), (3.0, 5.0), (7.0, . ), (2.0, -3.0),
             (4.0, -8.0), (6.0, . ))
```

Output:

```
AP       =  ((3.0, 0.0), (7.0, 10.0), (23.0, 0.0), (9.0, -4.0),
             (14.0, -23.0), (26.0, 0.0))
```

Example 5

This example shows vectors ***x*** and ***y*** with positive strides, and a real symmetric matrix ***A*** of order 3, stored in lower storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input:

```
            UPLO  N  ALPHA  X  INCX Y  INCY A  LDA
             |    |    |    |   |   |   |   |   |
CALL SSYR2( 'L' , 3 , 1.0 , X , 1 , Y , 2 , A , 3 )

X        =  (3.0, 2.0, 1.0)
Y        =  (5.0, . , 3.0, . , 2.0)

                      ┐
        | 8.0   .    . |
A    =  | 4.0  6.0   . |
        | 2.0  7.0  3.0|
                      ┘
```

Output:

```
        ┌                 ┐
        | 38.0    .    .  |
A    =  | 23.0  18.0   .  |
        | 13.0  14.0  7.0 |
                        ┘
```

Example 6

This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element X(5), which is 3.0. The real symmetric matrix ***A*** of order 3 is stored in upper storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input:

```
            UPLO  N  ALPHA  X  INCX  Y  INCY A  LDA
             |    |    |    |    |   |    |   |   |
CALL SSYR2( 'U' , 3 , 1.0 , X , -2 , Y , 2 , A , 4 )

X        =  (1.0, . , 2.0, . , 3.0)
Y        =  (5.0, . , 3.0, . , 2.0)

                      ┐
        | 8.0  4.0  2.0 |
A    =  |  .   6.0  7.0 |
        |  .    .   3.0 |
        |  .    .    . |
                       ┘
```

Output:

```
        ┌                 ┐
        | 38.0  23.0  13.0 |
A    =  |   .   18.0  14.0 |
        |   .     .    7.0 |
        |   .     .     .  |
                          ┘
```

Example 7

This example shows vector ***x*** and ***y*** with positive stride, and a complex Hermitian matrix ***A*** of order 3, stored in lower storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if α ≠ zero, the imaginary parts of the diagonal elements are set to zero.

Call Statement and Input:

```
            UPLO  N   ALPHA   X  INCX Y  INCY A  LDA
             |    |     |     |   |   |   |   |   |
CALL CHER2( 'L' , 3 , ALPHA , X , 1 , Y , 2 , A , 3 )

ALPHA    =  (1.0, 0.0)
X        =  ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0))
Y        =  ((1.0, 0.0), . , (2.0, -1.0), . , (2.0, 1.0))
```

```
        ┌                                  ┐
        |   (1.0, . )       .          .     |
A    =  | (3.0, -5.0)   (7.0, . )      .     |
        |  (2.0, 3.0)  (4.0, 8.0)  (6.0, . ) |
                                           ┘
```

Output:

```
        ┌                                        ┐
        |   (3.0, 0.0)       .             .       |
A    =  | (7.0, -10.0)  (23.0, 0.0 )       .       |
        |   (9.0, 4.0)  (14.0, 23.0)  (26.0, 0.0 ) |
                                                 ┘
```

Example 8

This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element `X(5)`, which is (1.0, 2.0). The complex Hermitian matrix ***A*** of order 3 is stored in upper storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if α ≠ zero, the imaginary parts of the diagonal elements are set to zero.

Call Statement and Input:

```
            UPLO  N   ALPHA   X  INCX  Y  INCY A  LDA
             |    |     |     |    |   |   |   |   |
CALL CHER2( 'U' , 3 , ALPHA , X , -2 , Y , 2 , A , 3 )

ALPHA    =  (1.0, 0.0)
X        =  ((3.0, 4.0), . , (4.0, 0.0), . , (1.0, 2.0))
Y        =  ((1.0, 0.0), . , (2.0, -1.0), . , (2.0, 1.0))
```

```
        ┌                                  ┐
        | (1.0, . ) (3.0, 5.0)   (2.0, -3.0) |
A    =  |     .      (7.0, . )   (4.0, -8.0) |
        |     .          .         (6.0, . ) |
                                           ┘
```

Output:

```
        ┌                                         ┐
        | (3.0, 0.0)  (7.0, 10.0)    (9.0, -4.0) |
A    =  |     .       (23.0, 0.0)  (14.0, -23.0) |
        |     .            .         (26.0, 0.0) |
                                                 ┘
```

Example 9

This example shows vectors ***x*** and ***y*** with positive strides and a real symmetric matrix ***A*** of order 3, stored in lower-packed storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input:

```
            N  ALPHA  X  INCX  Y  INCY  AP
            |    |    |   |    |   |    |
CALL SSLR2( 3 , 1.0 , X , 1  , Y , 2  , AP )

X        =  (3.0, 2.0, 1.0)
Y        =  (5.0, . , 3.0, . , 2.0)
AP       =  (8.0, 4.0, 2.0, 6.0, 7.0, 3.0)
```

Output:

```
AP       =  (38.0, 23.0, 13.0, 18.0, 14.0, 7.0)
```

SGBMV, DGBMV, CGBMV, and ZGBMV (Matrix-Vector Product for a General Band Matrix, Its Transpose, or Its Conjugate Transpose)

Purpose

SGBMV and DGBMV compute the matrix-vector product for either a real general band matrix or its transpose, where the general band matrix is stored in BLAS-general-band storage mode. It uses the scalars α and β, vectors ***x*** and ***y***, and general band matrix ***A*** or its transpose:

$\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A} \boldsymbol{x}$

$\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A}^T \boldsymbol{x}$

CGBMV and ZGBMV compute the matrix-vector product for either a complex general band matrix, its transpose, or its conjugate transpose, where the general band matrix is stored in BLAS-general-band storage mode. It uses the scalars α and β, vectors ***x*** and ***y***, and general band matrix ***A***, its transpose, or its conjugate transpose:

$\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A} \boldsymbol{x}$
$\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A}^T \boldsymbol{x}$
$\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A}^H \boldsymbol{x}$

Table 96: Data Types.

Data Types	
α, β, ***x***, ***y***, ***A***	**Subprogram**
Short-precision real	SGBMV
Long-precision real	DGBMV
Short-precision complex	CGBMV
Long-precision complex	ZGBMV

Syntax

Language	Syntax
Fortran	CALL SGBMV | DGBMV | CGBMV | ZGBMV (*transa*, *m*, *n*, *ml*, *mu*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
C and C++	sgbmv | dgbmv | cgbmv | zgbmv (*transa*, *m*, *n*, *ml*, *mu*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*);
CBLAS	cblas_sgbmv | cblas_dgbmv |cblas_cgbmv | cblas_zgbmv (*cblas_layout*, *cblas_transa*, *m*, *n*, *ml*, *mu*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*);

On Entry

cblas_layout

indicates whether the input matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

transa

indicates the form of matrix $\boldsymbol{A}$ to use in the computation, where:

If *transa* = 'N', $\boldsymbol{A}$ is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation.

Specified as: a single character. It must be 'N', 'T', or 'C'.

cblas_transa

indicates the form of matrix $\boldsymbol{A}$ to use in the computation, where:

If *cblas_transa* = CblasNoTrans, $\boldsymbol{A}$ is used in the computation.

If *cblas_transa* = CblasTrans, $\boldsymbol{A}^T$ is used in the computation.

If *cblas_transa* = CblasConjTrans, $\boldsymbol{A}^H$ is used in the computation.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

m

is the number of rows in matrix $\boldsymbol{A}$, and:

If *transa* = 'N', it is the length of vector $\boldsymbol{y}$.

If *transa* = 'T' or 'C', it is the length of vector $\boldsymbol{x}$.

Specified as: an integer; $m \geq 0$.

n

is the number of columns in matrix $\boldsymbol{A}$, and:

If *transa* = 'N', it is the length of vector $\boldsymbol{x}$.

If *transa* = 'T' or 'C', it is the length of vector $\boldsymbol{y}$.

Specified as: an integer; $n \geq 0$.

ml

is the lower band width *ml* of the matrix $\boldsymbol{A}$.

Specified as: an integer; $ml \geq 0$.

mu

is the upper band width *mu* of the matrix $\boldsymbol{A}$.

Specified as: an integer; $mu \geq 0$.

alpha
is the scaling constant α.

Specified as: a number of the data type indicated in Table 96 on page 336.

a
is the *m* by *n* general band matrix ***A***, stored in BLAS-general-band storage mode. It has an upper band width *mu* and a lower band width *ml*. Also:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation.

Note: No data should be moved to form $\boldsymbol{A}^T$ or $\boldsymbol{A}^H$; that is, the matrix ***A*** should always be stored in its untransposed form in BLAS-general-band storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 96 on page 336, where $lda \geq ml+mu+1$.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and $lda \geq ml+mu+1$.

x
is the vector ***x***, where:

If *transa* = 'N', it has length *n*.

If *transa* = 'T' or 'C', it has length *m*.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 96 on page 336, where:

If *transa* = 'N', it must have at least $1+(n-1)|incx|$ elements.

If *transa* = 'T' or 'C', it must have at least $1+(m-1)|incx|$ elements.

incx
is the stride for vector ***x***.

Specified as: an integer; $incx > 0$ or $incx < 0$.

beta
is the scaling constant β.

Specified as: a number of the data type indicated in Table 96 on page 336.

y
is the vector ***y***, where:

If *transa* = 'N', it has length *m*.

If *transa* = 'T' or 'C', it has length *n*.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 96 on page 336, where:

If *transa* = 'N', it must have at least $1+(m-1)|incy|$ elements.

If *transa* = 'T' or 'C', it must have at least $1+(n-1)|incy|$ elements.

incy
is the stride for vector ***y***.

Specified as: an integer; $incy > 0$ or $incy < 0$.

On Return

y

is the vector ***y***, containing the result of the computation, where:

If *transa* = 'N', it has length *m*.

If *transa* = 'T' or 'C', it has length *n*.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 96 on page 336.

Notes

1. For SGBMV and DGBMV, if you specify 'C' for the *transa* argument, it is interpreted as though you specified 'T'.
2. All subroutines accept lowercase letters for the *transa* argument.
3. Vector ***y*** must have no common elements with matrix ***A*** or vector ***x***; otherwise, results are unpredictable. See "Vector concepts" on page 71.
4. To achieve optimal performance, use $lda = mu+ml+1$.
5. For general band matrices, if you specify $ml \geq m$ or $mu \geq n$, ESSL assumes, **only for purposes of the computation**, that the lower band width is m-1 or the upper band width is n-1, respectively. However, ESSL uses the original values for *ml* and *mu* **for the purposes of finding the locations** of element a_{11} and all other elements in the array specified for ***A***, as described in "General Band Matrix" on page 94. For an illustration of this technique, see Example 4.
6. For a description of how a general band matrix is stored in BLAS-general-band storage mode in an array, see "General Band Matrix" on page 94.

Function

The possible computations that can be performed by these subroutines are described. Varying implementation techniques are used for this computation to improve performance. As a result, accuracy of the computational result may vary for different computations.

In all the computations, general band matrix ***A*** is stored in its untransposed form in an array, using BLAS-general-band storage mode.

For SGBMV and CGBMV, intermediate results are accumulated in long precision. Occasionally, for performance reasons, these intermediate results are truncated to short precision and stored.

See references ["44" on page 1255], ["45" on page 1255], ["48" on page 1255], ["56" on page 1256], and ["95" on page 1258]. No computation is performed if *m* or *n* is 0 or if α is zero and β is one.

General Band Matrix

For SGBMV, DGBMV, CGBMV, and ZGBMV, the matrix-vector product for a general band matrix is expressed as follows:

$\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A} \boldsymbol{x}$

where:

x is a vector of length *n*.

y is a vector of length *m*.

α is a scalar.

β is a scalar.

A is an *m* by *n* general band matrix, having a lower band width of *ml* and an upper band width of *mu*.

Transpose of a General Band Matrix

For SGBMV, DGBMV, CGBMV, and ZGBMV, the matrix-vector product for the transpose of a general band matrix is expressed as:

$\boldsymbol{y} \leftarrow \beta\boldsymbol{y}+\alpha\boldsymbol{A}^T\boldsymbol{x}$

where:

x is a vector of length *m*.

y is a vector of length *n*.

α is a scalar.

β is a scalar.

$\boldsymbol{A}^T$ is the transpose of an *m* by *n* general band matrix ***A***, having a lower band width of *ml* and an upper band width of *mu*.

Conjugate Transpose of a General Band Matrix

For CGBMV and ZGBMV, the matrix-vector product for the conjugate transpose of a general band matrix is expressed as follows:

$\boldsymbol{y} \leftarrow \beta\boldsymbol{y}+\alpha\boldsymbol{A}^H\boldsymbol{x}$

where:

x is a vector of length *m*.

y is a vector of length *n*.

α is a scalar.

β is a scalar.

$\boldsymbol{A}^H$ is the conjugate transpose of an *m* by *n* general band matrix ***A*** of order *n*, having a lower band width of *ml* and an upper band width of *mu*.

Error conditions

Resource Errors
Unable to allocate internal work area

Computational Errors
None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. *transa* ≠ 'N', 'T', or 'C'
3. *cblas_transa* ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
4. $m < 0$
5. $n < 0$
6. $ml < 0$
7. $mu < 0$
8. $lda \leq 0$
9. $lda < ml+mu+1$
10. $incx = 0$
11. $incy = 0$

Examples

Example 1

This example shows how to use SGBMV to perform the computation $\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A x}$, where TRANSA is equal to 'N', and the following real general band matrix ***A*** is used in the computation. Matrix ***A*** is:

```
                      | 1.0  1.0  1.0  0.0 |
                      | 2.0  2.0  2.0  2.0 |
                      | 3.0  3.0  3.0  3.0 |
                      | 4.0  4.0  4.0  4.0 |
                      | 0.0  5.0  5.0  5.0 |
```

Call Statement and Input:

```
           TRANSA M   N   ML  MU  ALPHA  A  LDA  X  INCX  BETA   Y  INCY
             |    |   |   |   |     |    |   |   |   |     |     |   |
CALL SGBMV( 'N' , 5 , 4 , 3 , 2 ,  2.0 , A , 8 , X , 1  , 10.0 , Y , 2  )
```

```
         |  .    .   1.0  2.0 |
         |  .   1.0  2.0  3.0 |
         | 1.0  2.0  3.0  4.0 |
A    =   | 2.0  3.0  4.0  5.0 |
         | 3.0  4.0  5.0   .  |
         | 4.0  5.0   .    .  |
         |  .    .    .    .  |
         |  .    .    .    .  |
```

```
X        =  (1.0, 2.0, 3.0, 4.0)
Y        =  (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0, . )
```

Output:

```
Y        =  (22.0, . , 60.0, . , 90.0, . , 120.0, . , 140.0, . )
```

Example 2

This example shows how to use SGBMV to perform the computation $\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A}^{T} \boldsymbol{x}$, where TRANSA is equal to 'T', and the transpose of a real general band matrix ***A*** is used in the computation. It uses the same input as Example 1.

Call Statement and Input:

```
           TRANSA M   N   ML  MU  ALPHA  A  LDA  X  INCX  BETA   Y  INCY
             |    |   |   |   |     |    |   |   |   |     |     |   |
CALL SGBMV( 'T' , 5 , 4 , 3 , 2 ,  2.0 , A , 8 , X , 1  , 10.0 , Y , 2  )
```

Output:

```
Y        =  (70.0, . , 130.0, . , 140.0, . , 148.0, . )
```

Example 3

This example shows how to use CGBMV to perform the computation $\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A}^{H} \boldsymbol{x}$, where TRANSA is equal to 'C', and the complex conjugate of the following general band matrix ***A*** is used in the computation. Matrix ***A*** is:

```
              | (1.0, 1.0)  (1.0, 1.0)  (1.0, 1.0)  (0.0, 0.0) |
              | (2.0, 2.0)  (2.0, 2.0)  (2.0, 2.0)  (2.0, 2.0) |
              | (3.0, 3.0)  (3.0, 3.0)  (3.0, 3.0)  (3.0, 3.0) |
              | (4.0, 4.0)  (4.0, 4.0)  (4.0, 4.0)  (4.0, 4.0) |
              | (0.0, 0.0)  (5.0, 5.0)  (5.0, 5.0)  (0.0, 0.0) |
```

Call Statement and Input:

```
            TRANSA M   N   ML  MU  ALPHA   A  LDA  X  INCX  BETA   Y  INCY
              |    |   |   |   |     |     |   |   |   |     |     |   |
CALL CGBMV( 'C' , 5 , 4 , 3 , 2 , ALPHA , A , 8 , X , 1  , BETA , Y , 2  )
```

```
        |     .           .         (1.0, 1.0)  (2.0, 2.0) |
        |     .       (1.0, 1.0)  (2.0, 2.0)  (3.0, 3.0) |
        | (1.0, 1.0)  (2.0, 2.0)  (3.0, 3.0)  (4.0, 4.0) |
A    =  | (2.0, 2.0)  (3.0, 3.0)  (4.0, 4.0)  (5.0, 5.0) |
        | (3.0, 3.0)  (4.0, 4.0)  (5.0, 5.0)      .      |
        | (4.0, 4.0)  (5.0, 5.0)      .           .      |
        |     .           .           .           .      |
        |     .           .           .           .      |
```

```
X        =  ((1.0, 2.0), (2.0, 3.0), (3.0, 4.0), (4.0, 5.0),
             (5.0, 6.0))
ALPHA    =  (1.0, 1.0)
BETA     =  (10.0, 0.0)
Y        =  ((1.0, 2.0), . , (2.0, 3.0), . , (3.0, 4.0), . ,
             (4.0, 5.0), . )
```

Output:

```
Y        =  ((70.0, 100.0), . , (130.0, 170.0), . ,
             (140.0, 180.0), . , (148.0, 186.0), . )
```

Example 4

This example shows how to use SGBMV to perform the computation ***y***←β***y***+α***Ax***, where *ml* ≥ *m* and *mu* ≥ *n*, TRANSA is equal to 'N', and the following real general band matrix ***A*** is used in the computation. Matrix ***A*** is:

```
                        | 1.0  1.0  1.0  1.0  1.0 |
                        | 2.0  2.0  2.0  2.0  2.0 |
                        | 3.0  3.0  3.0  3.0  3.0 |
                        | 4.0  4.0  4.0  4.0  4.0 |
```

Call Statement and Input:

```
            TRANSA M   N   ML  MU  ALPHA  A  LDA   X  INCX  BETA   Y  INCY
              |    |   |   |   |     |    |   |    |   |     |     |   |
CALL SGBMV( 'N' , 4 , 5 , 6 , 5 ,  2.0 , A , 12 , X , 1  , 10.0 , Y , 2  )
```

```
        |  .    .    .    .    .  |
        |  .    .    .    .   1.0 |
        |  .    .    .   1.0  2.0 |
        |  .    .   1.0  2.0  3.0 |
        |  .   1.0  2.0  3.0  4.0 |
A    =  | 1.0  2.0  3.0  4.0   .  |
        | 2.0  3.0  4.0   .    .  |
        | 3.0  4.0   .    .    .  |
        | 4.0   .    .    .    .  |
        |  .    .    .    .    .  |
        |  .    .    .    .    .  |
        |  .    .    .    .    .  |


X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 2.0, . , 3.0, . , 4.0, . )
```

Output:

```
Y        =  (40.0, . , 80.0, . , 120.0, . , 160.0, . )
```

SSBMV, DSBMV, CHBMV, and ZHBMV (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Band Matrix)

Purpose

SSBMV and DSBMV compute the matrix-vector product for a real symmetric band matrix. CHBMV and ZHBMV compute the matrix-vector product for a complex Hermitian band matrix. The band matrix ***A*** is stored in either upper- or lower-band-packed storage mode. It uses the scalars α and β, vectors ***x*** and ***y***, and band matrix ***A***:

$\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A}\boldsymbol{x}$
$\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A}\boldsymbol{x}$

Table 97: Data Types.

Data Types	
α, β, ***x***, ***y***, ***A***	**Subprogram**
Short-precision real	SSBMV
Long-precision real	DSBMV
Short-precision complex	CHBMV
Long-precision complex	ZHBMV

Syntax

Language	**Syntax**
Fortran	CALL SSBMV | DSBMV | CHBMV | ZHBMV (*uplo*, *n*, *k*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
C and C++	ssbmv | dsbmv | chbmv | zhbmv (*uplo*, *n*, *k*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*);
CBLAS	cblas_ssbmv | cblas_dsbmv | cblas_chbmv | cblas_zhbmv (*cblas_layout*, *cblas_uplo*, *n*, *k*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*);

On Entry

cblas_layout
indicates whether the input matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

uplo
indicates the storage mode used for matrix ***A***, where either the upper or lower triangle can be stored:

If *uplo* = 'U', ***A*** is stored in upper-band-packed storage mode.

If *uplo* = 'L', ***A*** is stored in lower-band-packed storage mode.

Specified as: a single character. It must be 'U' or 'L'.

cblas_uplo
indicates the storage mode used for matrix ***A***, where:

If *cblas_uplo* = CblasUpper, ***A*** is stored in upper-band-packed storage mode.

If *cblas_uplo* = CblasLower, ***A*** is stored in lower-band-packed storage mode.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

n
is the order of matrix ***A*** and the number of elements in vectors ***x*** and ***y***.

Specified as: an integer; $n \geq 0$.

k
is the half band width *k* of the matrix ***A***.

Specified as: an integer; $k \geq 0$.

alpha
is the scaling constant α.

Specified as: a number of the data type indicated in Table 97 on page 343.

a
is the real symmetric or complex Hermitian band matrix ***A*** of order *n*, having a half band width of *k*, where:

If *uplo* = 'U', ***A*** is stored in upper-band-packed storage mode.

If *uplo* = 'L', ***A*** is stored in lower-band-packed storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 97 on page 343, where $lda \geq k+1$.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and $lda \geq k+1$.

x
is the vector ***x*** of length *n*.

Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 97 on page 343.

incx
is the stride for vector ***x***.

Specified as: an integer; $incx > 0$ or $incx < 0$.

beta
is the scaling constant β.

Specified as: a number of the data type indicated in Table 97 on page 343.

y
is the vector ***y*** of length *n*.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 97 on page 343.

incy
is the stride for vector ***y***.

Specified as: an integer; $incy > 0$ or $incy < 0$.

On Return

y
is the vector ***y*** of length *n*, containing the result of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 97 on page 343.

Notes

1. All subroutines accept lowercase letters for the *uplo* argument.

2. Vector ***y*** must have no common elements with matrix ***A*** or vector ***x***; otherwise, results are unpredictable. See "Vector concepts" on page 71.
3. To achieve optimal performance in these subroutines, use *lda* = *k*+1.
4. The imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.
5. For real symmetric and complex Hermitian band matrices, if you specify $k \geq n$, ESSL assumes, **only for purposes of the computation**, that the half band width of matrix ***A*** is *n*-1; that is, it processes matrix ***A***, of order *n*, as though it is a (nonbanded) real symmetric or complex Hermitian matrix. However, ESSL uses the original value for *k* **for the purposes of finding the locations** of element a_{11} and all other elements in the array specified for ***A***, as described in the storage modes referenced in the next note. For an illustration of this technique, see Example 3.
6. For a description of how a real symmetric band matrix is stored, see "Upper-Band-Packed Storage Mode" on page 100 or "Lower-Band-Packed Storage Mode" on page 100. For a description of how a complex Hermitian band matrix is stored, see "Complex Hermitian Matrix" on page 84.

Function

These subroutines perform the following matrix-vector product, using a real symmetric or complex Hermitian band matrix ***A***, stored in either upper- or lower-band-packed storage mode:

$\boldsymbol{y} \leftarrow \beta \boldsymbol{y} + \alpha \boldsymbol{A}\boldsymbol{x}$

where:

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

α and β are scalars.

A is an real symmetric or complex Hermitian band matrix of order *n*, having a half bandwidth of *k*.

For SSBMV and CHBMV, intermediate results are accumulated in long precision when the VSX unit is not used. Occasionally, for performance reasons, these intermediate results are truncated to short precision and stored.

See references ["44" on page 1255], ["48" on page 1255], ["56" on page 1256], and ["95" on page 1258]. No computation is performed if *n* is 0 or if α is zero and β is one.

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. *uplo* ≠ 'U' or 'L'
3. *cblas_uplo* ≠ CblasLower or CblasUpper
4. $n < 0$
5. $k < 0$
6. $lda \leq 0$
7. $lda < k+1$
8. $incx = 0$
9. $incy = 0$

Examples

Example 1

This example shows how to use SSBMV to perform the matrix-vector product, where the real symmetric band matrix ***A*** of order 7 and half band width of 3 is stored in upper-band-packed storage mode. Matrix ***A*** is:

```
                   | 1.0  1.0  1.0  1.0  0.0  0.0  0.0 |
                   | 1.0  2.0  2.0  2.0  2.0  0.0  0.0 |
                   | 1.0  2.0  3.0  3.0  3.0  3.0  0.0 |
                   | 1.0  2.0  3.0  4.0  4.0  4.0  4.0 |
                   | 0.0  2.0  3.0  4.0  5.0  5.0  5.0 |
                   | 0.0  0.0  3.0  4.0  5.0  6.0  6.0 |
                   | 0.0  0.0  0.0  4.0  5.0  6.0  7.0 |
```

Call Statement and Input:

```
            UPLO  N   K  ALPHA  A  LDA  X  INCX  BETA   Y  INCY
             |    |   |    |    |   |   |   |     |     |   |
CALL SSBMV( 'U' , 7 , 3 , 2.0 , A , 5 , X , 1  , 10.0 , Y , 2  )


         |  .    .    .   1.0  2.0  3.0  4.0 |
         |  .    .   1.0  2.0  3.0  4.0  5.0 |
A    =   |  .   1.0  2.0  3.0  4.0  5.0  6.0 |
         | 1.0  2.0  3.0  4.0  5.0  6.0  7.0 |
         |  .    .    .    .    .    .    .  |


X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)
Y        =  (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0, . , 6.0, . , 7.0)
```

Output:

```
Y        =  (30.0, . , 78.0, . , 148.0, . , 244.0, . , 288.0, . ,
             316.0, . , 322.0)
```

Example 2

This example shows how to use CHBMV to perform the matrix-vector product, where the complex Hermitian band matrix ***A*** of order 7 and half band width of 3 is stored in lower-band-packed storage mode. Matrix ***A*** is:

```


         |  (1.0, 0.0)  (1.0, 1.0)  (1.0, 1.0)  (1.0, 1.0)  (0.0, 0.0)  (0.0, 0.0)  (0.0,
0.0) |
         | (1.0, -1.0)  (2.0, 0.0)  (2.0, 2.0)  (2.0, 2.0)  (2.0, 2.0)  (0.0, 0.0)  (0.0,
0.0) |
         | (1.0, -1.0) (2.0, -2.0)  (3.0, 0.0)  (3.0, 3.0)  (3.0, 3.0)  (3.0, 3.0)  (0.0,
0.0) |
         | (1.0, -1.0) (2.0, -2.0) (3.0, -3.0)  (4.0, 0.0)  (4.0, 4.0)  (4.0, 4.0)  (4.0,
4.0) |
         |  (0.0, 0.0) (2.0, -2.0) (3.0, -3.0) (4.0, -4.0)  (5.0, 0.0)  (5.0, 5.0)  (5.0,
5.0) |
         |  (0.0, 0.0)  (0.0, 0.0) (3.0, -3.0) (4.0, -4.0) (5.0, -5.0)  (6.0, 0.0)  (6.0,
6.0) |
         |  (0.0, 0.0)  (0.0, 0.0)  (0.0, 0.0) (4.0, -4.0) (5.0, -5.0) (6.0, -6.0)  (7.0,
0.0) |
```

Note: The imaginary parts of the diagonal elements of a complex Hermitian matrix are assumed to be zero, so you do not need to set these values.

Call Statement and Input:

```
            UPLO  N   K   ALPHA    A  LDA  X  INCX  BETA   Y  INCY
             |    |   |     |      |   |   |   |     |     |   |
CALL CHBMV( 'L' , 7 , 3 , ALPHA , A , 5 , X , 1  , BETA , Y , 2  )
```

```
ALPHA    =  (2.0, 0.0)
BETA     =  (10.0, 0.0)
```

```
        |  (1.0, . )  (2.0, . )  (3.0, . )  (4.0, . )  (5.0, . )  (6.0, . ) (7.0, . ) |
        | (1.0, 1.0) (2.0, 2.0) (3.0, 3.0) (4.0, 4.0) (5.0, 5.0) (6.0, 6.0)     .     |
A    =  | (1.0, 1.0) (2.0, 2.0) (3.0, 3.0) (4.0, 4.0) (5.0, 5.0)   .            .     |
        | (1.0, 1.0) (2.0, 2.0) (3.0, 3.0) (4.0, 4.0)   .          .            .     |
        |     .          .          .          .        .          .            .     |
```

```
X        =  ((1.0, 1.0), (2.0, 2.0), (3.0, 3.0), (4.0, 4.0),
             (5.0, 5.0), (6.0, 6.0), (7.0, 7.0))
Y        =  ((1.0, 1.0), . , (2.0, 2.0), . , (3.0, 3.0), . ,
             (4.0, 4.0), . , (5.0, 5.0), . , (6.0, 6.0), . ,
             (7.0, 7.0))
```

Output:

```
Y        =  ((48.0, 12.0), . , (124.0, 32.0), . , (228.0, 68.0), . ,
             (360.0, 128.0), . , (360.0, 216.0), . ,
             (300.0, 332.0), . , (168.0, 476.0))
```

Example 3

This example shows how to use SSBMV to perform the matrix-vector product, where $n \geq k$. Matrix ***A*** is a real 5 by 5 symmetric band matrix with a half band width of 5, stored in upper-band-packed storage mode. Matrix ***A*** is:

```
                        | 1.0  1.0  1.0  1.0  1.0 |
                        | 1.0  2.0  2.0  2.0  2.0 |
                        | 1.0  2.0  3.0  3.0  3.0 |
                        | 1.0  2.0  3.0  4.0  4.0 |
                        | 1.0  2.0  3.0  4.0  5.0 |
```

Call Statement and Input:

```
            UPLO  N   K  ALPHA  A  LDA  X  INCX  BETA   Y  INCY
             |    |   |    |    |   |   |   |     |     |   |
CALL SSBMV( 'U' , 5 , 5 , 2.0 , A , 7 , X , 1  , 10.0 , Y , 2  )


        |  .    .    .    .    .  |
        |  .    .    .    .   1.0 |
        |  .    .    .   1.0  2.0 |
A    =  |  .    .   1.0  2.0  3.0 |
        |  .   1.0  2.0  3.0  4.0 |
        | 1.0  2.0  3.0  4.0  5.0 |
        |  .    .    .    .    .  |


X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0, . )
```

Output:

```
Y        =  (40.0, . , 78.0, . , 112.0, . , 140.0, . , 160.0, . )
```

STRMV, DTRMV, CTRMV, ZTRMV, STPMV, DTPMV, CTPMV, and ZTPMV (Matrix-Vector Product for a Triangular Matrix, Its Transpose, or Its Conjugate Transpose)

Purpose

STRMV, DTRMV, STPMV, and DTPMV compute one of the following matrix-vector products, using the vector ***x*** and triangular matrix ***A*** or its transpose:

$\boldsymbol{x} \leftarrow \boldsymbol{A}\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{x}$

CTRMV, ZTRMV, CTPMV, and ZTPMV compute one of the following matrix-vector products, using the vector ***x*** and triangular matrix ***A***, its transpose, or its conjugate transpose:

$\boldsymbol{x} \leftarrow \boldsymbol{A}\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^{\mathrm{H}}\boldsymbol{x}$

Matrix ***A*** can be either upper or lower triangular, where:

- For the _TRMV subroutines, it is stored in upper- or lower-triangular storage mode, respectively.
- For the _TPMV subroutines, it is stored in upper- or lower-triangular-packed storage mode, respectively.

Table 98: Data Types.

Data Types	
A, ***x***	**Subprogram**
Short-precision real	STRMV and STPMV
Long-precision real	DTRMV and DTPMV
Short-precision complex	CTRMV and CTPMV
Long-precision complex	ZTRMV and ZTPMV

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL STRMV | DTRMV | CTRMV | ZTRMV (*uplo*, *transa*, *diag*, *n*, *a*, *lda*, *x*, *incx*) CALL STPMV | DTPMV | CTPMV | ZTPMV (*uplo*, *transa*, *diag*, *n*, *ap*, *x*, *incx*)

C and C++	strmv | dtrmv | ctrmv | ztrmv (*uplo*, *transa*, *diag*, *n*, *a*, *lda*, *x*, *incx*);
	stpmv | dtpmv | ctpmv | ztpmv (*uplo*, *transa*, *diag*, *n*, *ap*, *x*, *incx*);

CBLAS	cblas_strmv | cblas_dtrmv | cblas_ctrmv | cblas_ztrmv (*cblas_layout*, *cblas_uplo*, *cblas_transa*, *cblas_diag*, *n*, *a*, *lda*, *x*, *incx*);
	cblas_stpmv | cblas_dtpmv | cblas_ctpmv | cblas_ztpmv (*cblas_layout*, *cblas_uplo*, *cblas_transa*, *cblas_diag*, *n*, *ap*, *x*, *incx*);

On Entry

cblas_layout

indicates whether the input matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

uplo
indicates whether matrix ***A*** is an upper or lower triangular matrix, where:

If *uplo* = 'U', ***A*** is an upper triangular matrix.

If *uplo* = 'L', ***A*** is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

cblas_uplo
indicates whether matrix ***A*** is an upper or lower triangular matrix, where:

If *cblas_uplo* = CblasUpper, ***A*** is an upper triangular matrix.

If *cblas_uplo* = CblasLower, ***A*** is a lower triangular matrix.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

transa
indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation.

Specified as: a single character. It must be 'N', 'T', or 'C'.

cblas_transa
indicates the form of matrix ***A*** to use in the computation, where:

If *cblas_transa* = CblasNoTrans, ***A*** is used in the computation.

If *cblas_transa* = CblasTrans, $\boldsymbol{A}^T$ is used in the computation.

If *cblas_transa* = CblasConjTrans, $\boldsymbol{A}^H$ is used in the computation.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

cblas_diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = CblasUnit, ***A*** is a unit triangular matrix.

If *diag* = CblasNonUnit ***A*** is not a unit triangular matrix.

Specified as: an object of enumerated type CBLAS_DIAG. It must be CblasNonUnit or CblasUnit.

n
is the order of triangular matrix ***A***.

Specified as: an integer; $0 \le n \le lda$.

a
is the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular storage mode, respectively.

Note: No data should be moved to form $\boldsymbol{A}^T$ or $\boldsymbol{A}^H$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 98 on page 348.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

ap

is the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular-packed storage mode, respectively.

Specified as: a one-dimensional array of (at least) length *n*(*n*+1)/2, containing numbers of the data type indicated in Table 98 on page 348.

x

is the vector ***x*** of length *n*.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 98 on page 348.

incx

is the stride for vector ***x***.

Specified as: an integer; *incx* > 0 or *incx* < 0.

On Return

x

is the vector ***x*** of length *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 98 on page 348.

Notes

1. These subroutines accept lowercase letters for the *uplo*, *transa*, and *diag* arguments.
2. For STRMV, DTRMV, STPMV, and DTPMV if you specify 'C' for the *transa* argument, it is interpreted as though you specified 'T'.
3. Matrix ***A*** and vector ***x*** must have no common elements; otherwise, results are unpredictable.
4. ESSL assumes certain values in your array for parts of a triangular matrix. As a result, you do not have to set these values. For unit triangular matrices, the elements of the diagonal are assumed to be 1.0 for real matrices and (1.0, 0.0) for complex matrices. When using upper- or lower-triangular storage, the unreferenced elements in the lower and upper triangular part, respectively, are assumed to be zero.
5. For a description of triangular matrices and how they are stored in upper- and lower-triangular storage mode and in upper- and lower-triangular-packed storage mode, see "Triangular Matrix" on page 88.

Function

These subroutines can perform the following matrix-vector product computations, using the triangular matrix ***A***, its transpose, or its conjugate transpose, where ***A*** can be either upper or lower triangular:

$\boldsymbol{x} \leftarrow \boldsymbol{A}\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^{T}\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^{H}\boldsymbol{x}$ (for CTRMV, ZTRMV, CTPMV, and ZTPMV only)

where:

x is a vector of length *n*.

A is an upper or lower triangular matrix of order *n*. For _TRMV, it is stored in upper- or lower-triangular storage mode, respectively. For _TPMV, it is stored in upper- or lower-triangular-packed storage mode, respectively.

See references ["42" on page 1255] and ["48" on page 1255]. If *n* is 0, no computation is performed.

Error conditions

Computational Errors

None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. *uplo* ≠ 'L' or 'U'
3. *cblas_uplo* ≠ CblasLower or CblasUpper
4. *transa* ≠ 'T', 'N', or 'C'
5. *cblas_transa* ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
6. *diag* ≠ 'N' or 'U'
7. *cblas_diag* ≠ CblasNonUnit or CblasUnit
8. *n* < 0
9. *lda* ≤ 0
10. *lda* < *n*
11. *incx* = 0

Examples

Example 1

This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{A}\boldsymbol{x}$. Matrix ***A*** is a real 4 by 4 lower triangular matrix that is unit triangular, stored in lower-triangular storage mode. Vector ***x*** is a vector of length 4. Matrix ***A*** is:

```
                          ┌                    ┐
                          | 1.0   .    .    .  |
                          | 1.0  1.0   .    .  |
                          | 2.0  3.0  1.0   .  |
                          | 3.0  4.0  3.0  1.0 |
                          └                    ┘
```

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input:

```
            UPLO TRANSA DIAG  N   A  LDA  X  INCX
             |     |     |    |   |   |   |   |
CALL STRMV( 'L' , 'N' , 'U' , 4 , A , 4 , X , 1  )


                                   ┐
        |  .    .    .    . |
A    =  | 1.0   .    .    . |
        | 2.0  3.0   .    . |
        | 3.0  4.0  3.0   . |
                                   ┘

X        =  (1.0, 2.0, 3.0, 4.0)
```

Output:

```
X        =  (1.0, 3.0, 11.0, 24.0)
```

Example 2

This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{A}^T\boldsymbol{x}$. Matrix ***A*** is a real 4 by 4 upper triangular matrix that is unit triangular, stored in upper-triangular storage mode. Vector ***x*** is a vector of length 4. Matrix ***A*** is:

```
                      ┌                    ┐
                      | 1.0  2.0  3.0  2.0 |
                      |  .   1.0  2.0  5.0 |
                      |  .    .   1.0  3.0 |
                         |  .    .    .   1.0 |
                                             ┘
```

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input:

```
            UPLO TRANSA DIAG  N   A  LDA  X  INCX
             |     |     |    |   |   |   |   |
CALL STRMV( 'U' , 'T' , 'U' , 4 , A , 4 , X , 1  )


                                   ┐
        |  .   2.0  3.0  2.0 |
A    =  |  .    .   2.0  5.0 |
        |  .    .    .   3.0 |
        |  .    .    .    .  |
                                   ┘

X        =  (5.0, 4.0, 3.0, 2.0)
```

Output:

```
X        =  (5.0, 14.0, 26.0, 41.0)
```

Example 3

This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{A}^H \boldsymbol{x}$. Matrix ***A*** is a complex 4 by 4 upper triangular matrix that is unit triangular, stored in upper-triangular storage mode. Vector ***x*** is a vector of length 4. Matrix ***A*** is:

```
                ┌                                                ┐
                | (1.0, 0.0)  (2.0, 2.0)  (3.0, 3.0)  (2.0, 2.0) |
                |     .       (1.0, 0.0)  (2.0, 2.0)  (5.0, 5.0) |
                |     .           .       (1.0, 0.0)  (3.0, 3.0) |
                |     .           .           .       (1.0, 0.0) |
                                                                ┘
```

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of (1.0, 0.0) for the diagonal elements.

Call Statement and Input:

```
            UPLO TRANSA DIAG  N   A  LDA  X  INCX
             |     |     |    |   |   |   |   |
CALL CTRMV( 'U' , 'C' , 'U' , 4 , A , 4 , X , 1  )


                                                   ┐
        | .  (2.0, 2.0)  (3.0, 3.0)  (2.0, 2.0) |
A    =  | .      .       (2.0, 2.0)  (5.0, 5.0) |
        | .      .           .       (3.0, 3.0) |
        | .      .           .           .      |
                                                   ┘

X        =  ((5.0, 5.0), (4.0, 4.0), (3.0, 3.0), (2.0, 2.0))
```

Output:

```
X        =  ((5.0, 5.0), (24.0, 4.0), (49.0, 3.0), (80.0, 2.0))
```

Example 4

This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{A}\boldsymbol{x}$. Matrix ***A*** is a real 4 by 4 lower triangular matrix that is unit triangular, stored in lower-triangular-packed storage mode. Vector ***x*** is a vector of length 4. Matrix ***A*** is:

```
                          ┌                    ┐
                          | 1.0   .    .    .  |
                          | 1.0  1.0   .    .  |
                          | 2.0  3.0  1.0   .  |
                          | 3.0  4.0  3.0  1.0 |
                                              ┘
```

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input:

```
            UPLO TRANSA DIAG  N   AP   X  INCX
             |     |     |    |   |    |   |
CALL STPMV( 'L' , 'N' , 'U' , 4 , AP , X , 1  )

AP       =  ( . , 1.0, 2.0, 3.0, . , 3.0, 4.0, . , 3.0, . )
X        =  (1.0, 2.0, 3.0, 4.0)
```

Output:

```
X        =  (1.0, 3.0, 11.0, 24.0)
```

Example 5

This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{A}^T\boldsymbol{x}$. Matrix ***A*** is a real 4 by 4 upper triangular matrix that is not unit triangular, stored in upper-triangular-packed storage mode. Vector ***x*** is a vector of length 4. Matrix ***A*** is:

```
                            ┌                     ┐
                            | 1.0  2.0  3.0  2.0 |
                            |  .   2.0  2.0  5.0 |
                            |  .    .   3.0  3.0 |
                            |  .    .    .   1.0 |
                                                 ┘
```

Call Statement and Input:

```
            UPLO TRANSA DIAG  N   AP   X  INCX
             |     |     |    |   |    |   |
CALL STPMV( 'U' , 'T' , 'N' , 4 , AP , X , 1  )

AP       =  (1.0, 2.0, 2.0, 3.0, 2.0, 3.0, 2.0, 5.0, 3.0, 1.0)
X        =  (5.0, 4.0, 3.0, 2.0)
```

Output:

```
X        =  (5.0, 18.0, 32.0, 41.0)
```

Example 6

This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{A}^H\boldsymbol{x}$. Matrix ***A*** is a complex 4 by 4 upper triangular matrix that is unit triangular, stored in upper-triangular-packed storage mode. Vector ***x*** is a vector of length 4. Matrix ***A*** is:

```
                                                             ┐
            | (1.0, 0.0) (2.0, 2.0) (3.0, 3.0) (2.0, 2.0) |
            |     .      (1.0, 0.0) (2.0, 2.0) (5.0, 5.0) |
            |     .          .      (1.0, 0.0) (3.0, 3.0) |
            |     .          .          .      (1.0, 0.0) |
                                                             ┘
```

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of (1.0, 0.0) for the diagonal elements.

Call Statement and Input:

```
            UPLO TRANSA DIAG  N   AP   X  INCX
             |     |     |    |   |    |   |
CALL CTPMV( 'U' , 'C' , 'U' , 4 , AP , X , 1  )

AP       =  ( . , (2.0, 2.0), . , (3.0, 3.0), (2.0, 2.0), . ,
             (2.0, 2.0), (5.0, 5.0), (3.0, 3.0), . )
X        =  ((5.0, 5.0), (4.0, 4.0), (3.0, 3.0), (2.0, 2.0))
```

Output:

```
X        =  ((5.0, 5.0), (24.0, 4.0), (49.0, 3.0), (80.0, 2.0))
```

STRSV, DTRSV, CTRSV, ZTRSV, STPSV, DTPSV, CTPSV, and ZTPSV (Solution of a Triangular System of Equations with a Single Right-Hand Side)

Purpose

STRSV, DTRSV, STPSV, and DTPSV perform one of the following solves for a triangular system of equations with a single right-hand side, using the vector $\boldsymbol{x}$ and triangular matrix $\boldsymbol{A}$ or its transpose:

Solution	Equation
1. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$	$\boldsymbol{Ax} = \boldsymbol{b}$
2. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$	$\boldsymbol{A}^{T}\boldsymbol{x} = \boldsymbol{b}$

CTRSV, ZTRSV, CTPSV, and ZTPSV perform one of the following solves for a triangular system of equations with a single right-hand side, using the vector $\boldsymbol{x}$ and and triangular matrix $\boldsymbol{A}$, its transpose, or its conjugate transpose:

Solution	Equation
1. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$	$\boldsymbol{Ax} = \boldsymbol{b}$
2. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$	$\boldsymbol{A}^{T}\boldsymbol{x} = \boldsymbol{b}$
3. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-H}\boldsymbol{x}$	$\boldsymbol{A}^{H}\boldsymbol{x} = \boldsymbol{b}$

Matrix $\boldsymbol{A}$ can be either upper or lower triangular, where:

- For the _TRSV subroutines, it is stored in upper- or lower-triangular storage mode, respectively.
- For the _TPSV subroutines, it is stored in upper- or lower-triangular-packed storage mode, respectively.

Note: The term $\boldsymbol{b}$ used in the systems of equations listed above represents the right-hand side of the system. It is important to note that in these subroutines the right-hand side of the equation is actually provided in the input-output argument *x*.

Table 99: Data Types.

Data Types	
A*, *x	**Subroutine**
Short-precision real	STRSV and STPSV
Long-precision real	DTRSV and DTPSV
Short-precision complex	CTRSV and CTPSV
Long-precision complex	ZTRSV and ZTPSV

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL STRSV | DTRSV | CTRSV | ZTRSV (*uplo*, *transa*, *diag*, *n*, *a*, *lda*, *x*, *incx*) CALL STPSV | DTPSV | CTPSV | ZTPSV (*uplo*, *transa*, *diag*, *n*, *ap*, *x*, *incx*)

C and C++	strsv | dtrsv | ctrsv | ztrsv (*uplo*, *transa*, *diag*, *n*, *a*, *lda*, *x*, *incx*);
	stpsv | dtpsv | ctpsv | ztpsv (*uplo*, *transa*, *diag*, *n*, *ap*, *x*, *incx*);

CBLAS	cblas_strsv | cblas_dtrsv | cblas_ctrsv | cblas_ztrsv (*cblas_layout*, *cblas_uplo*, *cblas_transa*, *cblas_diag*, *n*, *a*, *lda*, *x*, *incx*);
	cblas_stpsv | cblas_dtpsv | cblas_ctpsv | cblas_ztpsv (*cblas_layout*, *cblas_uplo*, *cblas_transa*, *cblas_diag*, *n*, *ap*, *x*, *incx*);

On Entry

cblas_layout

indicates whether the input matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

uplo

indicates whether matrix ***A*** is an upper or lower triangular matrix, where:

If *uplo* = 'U', ***A*** is an upper triangular matrix.

If *uplo* = 'L', ***A*** is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

cblas_uplo

indicates whether matrix ***A*** is an upper or lower triangular matrix, where:

If *cblas_uplo* = CblasUpper, ***A*** is an upper triangular matrix.

If *cblas_uplo* = CblasLower, ***A*** is a lower triangular matrix.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

transa

indicates the form of matrix ***A*** used in the system of equations, where:

If *transa* = 'N', ***A*** is used, resulting in solution 1.

If *transa* = 'T', $\boldsymbol{A}^T$ is used, resulting in solution 2.

If *transa* = 'C', $\boldsymbol{A}^H$ is used, resulting in solution 3.

Specified as: a single character. It must be 'N', 'T', or 'C'.

cblas_transa

indicates the form of matrix ***A*** to use in the computation, where:

If *cblas_transa* = CblasNoTrans, ***A*** is used, resulting in solution 1.

If *cblas_transa* = CblasTrans, $\boldsymbol{A}^T$ is used, resulting in solution 2.

If *cblas_transa* = CblasConjTrans, $\boldsymbol{A}^H$ is used, resulting in solution 3.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

cblas_diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = CblasUnit, ***A*** is a unit triangular matrix.

If *diag* = CblasNonUnit ***A*** is not a unit triangular matrix.

Specified as: an object of enumerated type CBLAS_DIAG. It must be CblasNonUnit or CblasUnit.

n
is the order of triangular matrix ***A***.

Specified as: an integer; $n \geq 0$ and $n \leq lda$.

a
is the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular storage mode, respectively. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 99 on page 354.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

ap
is the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular-packed storage mode, respectively.

Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 99 on page 354.

x
is the vector ***x*** of length *n*, containing the right-hand side of the triangular system to be solved.

Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 99 on page 354.

incx
is the stride for vector ***x***.

Specified as: an integer; $incx > 0$ or $incx < 0$.

On Return

x
is the solution vector ***x*** of length *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 99 on page 354.

Notes

1. These subroutines accept lowercase letters for the *uplo*, *transa*, and *diag* arguments.
2. For STRSV, DTRSV, STPSV, and DTPSV, if you specify 'C' for the *transa* argument, it is interpreted as though you specified 'T'.
3. Matrix ***A*** and vector ***x*** must have no common elements; otherwise, results are unpredictable.
4. ESSL assumes certain values in your array for parts of a triangular matrix. As a result, you do not have to set these values. For unit diagonal matrices, the elements of the diagonal are assumed to be 1.0 for real matrices and (1.0, 0.0) for complex matrices. When using upper- or lower-triangular storage, the unreferenced elements in the lower and upper triangular part, respectively, are assumed to be zero.

5. For a description of triangular matrices and how they are stored in upper- and lower-triangular storage mode and in upper- and lower-triangular-packed storage mode, see "Triangular Matrix" on page 88.

Function

These subroutines solve a triangular system of equations with a single right-hand side. The solution ***x*** may be any of the following, where triangular matrix ***A***, its transpose, or its conjugate transpose is used, and where ***A*** can be either upper- or lower-triangular:

$\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^{-H}\boldsymbol{x}$ (only for CTRSV, ZTRSV, CTPSV, and ZTPSV)

where:

x is a vector of length *n*.

A is an upper or lower triangular matrix of order *n*. For _TRSV, it is stored in upper- or lower-triangular storage mode, respectively. For _TPSV, it is stored in upper- or lower-triangular-packed storage mode, respectively.

If *n* is 0, no computation is performed. See references ["42" on page 1255], ["46" on page 1255], and ["48" on page 1255].

Error conditions

Computational Errors

None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. *uplo* ≠ 'L' or 'U'
3. *cblas_uplo* ≠ CblasLower or CblasUpper
4. *transa* ≠ 'T', 'N', or 'C'
5. *cblas_transa* ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
6. *diag* ≠ 'N' or 'U'
7. *cblas_diag* ≠ CblasNonUnit or CblasUnit
8. *n* < 0
9. *lda* ≤ 0
10. *lda* < *n*
11. *incx* = 0

Examples

Example 1

This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$. Matrix ***A*** is a real 4 by 4 lower unit triangular matrix, stored in lower-triangular storage mode. Vector ***x*** is a vector of length 4.

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input:

```
             UPLO TRANSA DIAG  N   A  LDA  X  INCX
              |     |     |    |   |   |   |   |
CALL STRSV( 'L' , 'N' , 'U' , 4 , A , 4 , X , 1  )
```

```
                    ┐
        |  .    .    .   . |
        | 1.0   .    .   . |
A    =  | 2.0  3.0   .   . |
        | 3.0  4.0  3.0  . |
                           ┘

X        =  (1.0, 3.0, 11.0, 24.0)
```

Output:

```
X        =  (1.0, 2.0, 3.0, 4.0)
```

Example 2

This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$. Matrix ***A*** is a real 4 by 4 upper nonunit triangular matrix, stored in upper-triangular storage mode. Vector ***x*** is a vector of length 4.

Call Statement and Input:

```
            UPLO TRANSA DIAG  N   A  LDA  X  INCX
             |     |     |    |   |   |   |   |
CALL STRSV( 'U' , 'T' , 'N' , 4 , A , 4 , X , 1  )


                              ┐
        | 1.0  2.0  3.0  2.0 |
A    =  |  .   2.0  2.0  5.0 |
        |  .    .   3.0  3.0 |
        |  .    .    .   1.0 |
                              ┘

X        =  (5.0, 18.0, 32.0, 41.0)
```

Output:

```
X        =  (5.0, 4.0, 3.0, 2.0)
```

Example 3

This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-H}\boldsymbol{x}$. Matrix ***A*** is a complex 4 by 4 upper unit triangular matrix, stored in upper-triangular storage mode. Vector ***x*** is a vector of length 4.

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of (1.0, 0.0) for the diagonal elements.

Call Statement and Input:

```
            UPLO TRANSA DIAG  N   A  LDA  X  INCX
             |     |     |    |   |   |   |   |
CALL CTRSV( 'U' , 'C' , 'U' , 4 , A , 4 , X , 1  )


                                               ┐
        | . (2.0, 2.0) (3.0,  3.0) (2.0, 2.0) |
A    =  | .     .      (2.0,  2.0) (5.0, 5.0) |
        | .     .           .      (3.0, 3.0) |
        | .     .           .           .     |
                                               ┘

X        =  ((5.0, 5.0), (24.0, 4.0), (49.0, 3.0), (80.0, 2.0))
```

Output:

```
X        =  ((5.0, 5.0), (4.0, 4.0), (3.0, 3.0), (2.0, 2.0))
```

Example 4

This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$. Matrix ***A*** is a real 4 by 4 lower unit triangular matrix, stored in lower-triangular-packed storage mode. Vector ***x*** is a vector of length 4. Matrix ***A*** is:

```
                         ┌                    ┐
                         | 1.0   .    .    .  |
                         | 1.0  1.0   .    .  |
                        | 2.0  3.0  1.0   .  |
                        | 3.0  4.0  3.0  1.0 |
                                            ┘
```

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input:

```
            UPLO TRANSA DIAG  N   AP   X  INCX
             |     |     |    |   |    |   |
CALL STPSV( 'L' , 'N' , 'U' , 4 , AP , X , 1  )

AP       =  ( . , 1.0, 2.0, 3.0, . , 3.0, 4.0, . , 3.0, . )
X        =  (1.0, 3.0, 11.0, 24.0)
```

Output:

```
X        =  (1.0, 2.0, 3.0, 4.0)
```

Example 5

This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$. Matrix ***A*** is a real 4 by 4 upper nonunit triangular matrix, stored in upper-triangular-packed storage mode. Vector ***x*** is a vector of length 4. Matrix ***A*** is:

```
                       ┌                    ┐
                       | 1.0  2.0  3.0  2.0 |
                       |  .   2.0  2.0  5.0 |
                       |  .    .   3.0  3.0 |
                       |  .    .    .   1.0 |
                                           ┘
```

Call Statement and Input:

```
            UPLO TRANSA DIAG  N   AP   X  INCX
             |     |     |    |   |    |   |
CALL STPSV( 'U' , 'T' , 'N' , 4 , AP , X , 1  )

AP       =  (1.0, 2.0, 2.0, 3.0, 2.0, 3.0, 2.0, 5.0, 3.0, 1.0)
X        =  (5.0, 18.0, 32.0, 41.0)
```

Output:

```
X        =  (5.0, 4.0, 3.0, 2.0)
```

Example 6

This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-H}\boldsymbol{x}$. Matrix ***A*** is a complex 4 by 4 upper unit triangular matrix, stored in upper-triangular-packed storage mode. Vector ***x*** is a vector of length 4. Matrix ***A*** is:

```
                 ┌                                          ┐
                 | (1.0, 0.0)  (2.0, 2.0)  (3.0, 3.0)  (2.0, 2.0) |
                 |     .       (1.0, 0.0)  (2.0, 2.0)  (5.0, 5.0) |
                 |     .           .       (1.0, 0.0)  (3.0, 3.0) |
                 |     .           .           .       (1.0, 0.0) |
                                                                 ┘
```

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of (1.0, 0.0) for the diagonal elements.

Call Statement and Input:

```
            UPLO TRANSA DIAG  N   AP   X  INCX
             |     |     |    |   |    |   |
CALL CTPSV( 'U' , 'C' , 'U' , 4 , AP , X , 1  )

AP       =  ( . , (2.0, 2.0), . , (3.0, 3.0), (2.0, 2.0), . ,
             (2.0, 2.0), (5.0, 5.0), (3.0, 3.0), . )
X        =  ((5.0, 5.0), (24.0, 4.0), (49.0, 3.0), (80.0, 2.0))
```

Output:

```
X        =  ((5.0, 5.0), (4.0, 4.0), (3.0, 3.0), (2.0, 2.0))
```

STBMV, DTBMV, CTBMV, and ZTBMV (Matrix-Vector Product for a Triangular Band Matrix, Its Transpose, or Its Conjugate Transpose)

Purpose

STBMV and DTBMV compute one of the following matrix-vector products, using the vector ***x*** and triangular band matrix ***A*** or its transpose:

$\boldsymbol{x} \leftarrow \boldsymbol{Ax}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^T\boldsymbol{x}$

CTBMV and ZTBMV compute one of the following matrix-vector products, using the vector ***x*** and triangular band matrix ***A***, its transpose, or its conjugate transpose:

$\boldsymbol{x} \leftarrow \boldsymbol{Ax}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^T\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^H\boldsymbol{x}$

Matrix ***A*** can be either upper or lower triangular and is stored in upper- or lower-triangular-band-packed storage mode, respectively.

Table 100: Data Types.

Data Types	
A, ***x***	**Subprogram**
Short-precision real	STBMV
Long-precision real	DTBMV
Short-precision complex	CTBMV
Long-precision complex	ZTBMV

Syntax

Language	Syntax
Fortran	CALL STBMV | DTBMV | CTBMV | ZTBMV (*uplo*, *transa*, *diag*, *n*, *k*, *a*, *lda*, *x*, *incx*)
C and C++	stbmv | dtbmv | ctbmv | ztbmv (*uplo*, *transa*, *diag*, *n*, *k*, *a*, *lda*, *x*, *incx*);
CBLAS	cblas_stbmv | cblas_dtbmv | cblas_ctbmv | cblas_ztbmv (*cblas_layout*, *cblas_uplo*, *cblas_transa*, *cblas_diag*, *n*, *k*, *a*, *lda*, *x*, *incx*);

On Entry

cblas_layout

indicates whether the input matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

uplo
indicates whether matrix ***A*** is an upper or lower triangular band matrix, where:

If *uplo* = 'U', ***A*** is an upper triangular matrix.

If *uplo* = 'L', ***A*** is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

cblas_uplo
indicates whether matrix ***A*** is an upper or lower triangular matrix, where:

If *cblas_uplo* = CblasUpper, ***A*** is an upper triangular matrix.

If *cblas_uplo* = CblasLower, ***A*** is a lower triangular matrix.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

transa
indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation.

Specified as: a single character. It must be 'N', 'T', or 'C'.

cblas_transa
indicates the form of matrix ***A*** to use in the computation, where:

If *cblas_transa* = CblasNoTrans, ***A*** is used in the computation.

If *cblas_transa* = CblasTrans, $\boldsymbol{A}^T$ is used in the computation.

If *cblas_transa* = CblasConjTrans, $\boldsymbol{A}^H$ is used in the computation.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

cblas_diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = CblasUnit, ***A*** is a unit triangular matrix.

If *diag* = CblasNonUnit ***A*** is not a unit triangular matrix.

Specified as: an object of enumerated type CBLAS_DIAG. It must be CblasNonUnit or CblasUnit.

n

is the order of triangular band matrix ***A***. Specified as: an integer; $n \geq 0$.

k

is the upper or lower band width k of the matrix ***A***.

Specified as: an integer; $k \geq 0$.

a

is the upper or lower triangular band matrix ***A*** of order *n*, stored in upper- or lower-triangular-band-packed storage mode, respectively.

Note: No data should be moved to form $\boldsymbol{A}^T$ or $\boldsymbol{A}^H$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 100 on page 360.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and $lda \geq k+1$.

x
is the vector ***x*** of length *n*.

Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 100 on page 360.

incx
is the stride for vector ***x***.

Specified as: an integer; $incx > 0$ or $incx < 0$.

On Return

x
is the vector ***x*** of length *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 100 on page 360.

Notes

1. These subroutines accept lowercase letters for the *uplo*, *transa*, and *diag* arguments.
2. For STBMV and DTBMV, if you specify 'C' for the *transa* argument, it is interpreted as though you specified 'T'.
3. Matrix ***A*** and vector ***x*** must have no common elements; otherwise, results are unpredictable.
4. To achieve optimal performance in these subroutines, use $lda = k+1$.
5. For unit triangular matrices, the elements of the diagonal are assumed to be 1.0 for real matrices and (1.0, 0.0) for complex matrices. As a result, you do not have to set these values.
6. For both upper and lower triangular band matrices, if you specify $k \geq n$, ESSL assumes, **only for purposes of the computation**, that the upper or lower band width of matrix ***A*** is n-1; that is, it processes matrix ***A***, of order *n*, as though it is a (nonbanded) triangular matrix. However, ESSL uses the original value for *k* **for the purposes of finding the locations** of element a_{11} and all other elements in the array specified for ***A***, as described in "Triangular Band Matrix" on page 102. For an illustration of this technique, see Example 4.
7. For a description of triangular band matrices and how they are stored in upper- and lower-triangular-band-packed storage mode, see "Triangular Band Matrix" on page 102.
8. If you are using a lower triangular band matrix, you may want to use this alternate approach instead of using lower-triangular-band-packed storage mode. Leave matrix ***A*** in full-matrix storage mode when you pass it to ESSL and specify the *lda* argument to be *lda*+1, which is the leading dimension of matrix ***A*** plus 1. ESSL then processes the matrix elements in the same way as though you had set them up in lower-triangular-band-packed storage mode.

Function

These subroutines can perform the following matrix-vector product computations, using the triangular band matrix ***A***, its transpose, or its conjugate transpose, where ***A*** can be either upper or lower triangular:

$\mathbf{x} \leftarrow \mathbf{A}\mathbf{x}$
$\mathbf{x} \leftarrow \mathbf{A}^{T}\mathbf{x}$
$\mathbf{x} \leftarrow \mathbf{A}^{H}\mathbf{x}$ (for CTBMV and ZTBMV only)

where:

x is a vector of length *n*.

A is an upper or lower triangular band matrix of order *n*, stored in upper- or lower-triangular-band-packed storage mode, respectively.

See references [“44” on page 1255], [“56” on page 1256], and [“48” on page 1255]. If *n* is 0, no computation is performed.

Error conditions

Computational Errors

None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. *uplo* ≠ 'L' or 'U'
3. *cblas_uplo* ≠ CblasLower or CblasUpper
4. *transa* ≠ 'T', 'N', or 'C'
5. *cblas_transa* ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
6. *diag* ≠ 'N' or 'U'
7. *cblas_diag* ≠ CblasNonUnit or CblasUnit
8. *n* < 0
9. *k* < 0
10. *lda* ≤ 0
11. *lda* < *k*+1
12. *incx* = 0

Examples

Example 1

This example shows the computation ***x***←***Ax***. Matrix ***A*** is a real 7 by 7 upper triangular band matrix with a half band width of 3 that is not unit triangular, stored in upper-triangular-band-packed storage mode. Vector ***x*** is a vector of length 7. Matrix ***A*** is:

```
                    ┌                                    ┐
                    | 1.0  1.0  1.0  1.0  0.0  0.0  0.0 |
                    | 0.0  2.0  2.0  2.0  2.0  0.0  0.0 |
                    | 0.0  0.0  3.0  3.0  3.0  3.0  0.0 |
                    | 0.0  0.0  0.0  4.0  4.0  4.0  4.0 |
                    | 0.0  0.0  0.0  0.0  5.0  5.0  5.0 |
                    | 0.0  0.0  0.0  0.0  0.0  6.0  6.0 |
                    | 0.0  0.0  0.0  0.0  0.0  0.0  7.0 |
                    └                                    ┘
```

Call Statement and Input:

```
            UPLO TRANSA DIAG   N   K   A  LDA  X  INCX
             |     |     |     |   |   |   |   |   |
CALL STBMV( 'U' , 'N' , 'N' , 7 , 3 , A , 5 , X , 1  )

                                        ┐
        |  .    .    .   1.0  2.0  3.0  4.0 |
        |  .    .   1.0  2.0  3.0  4.0  5.0 |
A    =  |  .   1.0  2.0  3.0  4.0  5.0  6.0 |
        | 1.0  2.0  3.0  4.0  5.0  6.0  7.0 |
        |  .    .    .    .    .    .    .  |
                                        ┘

X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)
```

Output:

```
X        =  (10.0, 28.0, 54.0, 88.0, 90.0, 78.0, 49.0)
```

Example 2

This example shows the computation ***x***←***A***^{T}***x***. Matrix ***A*** is a real 7 by 7 lower triangular band matrix with a half band width of 3 that is not unit triangular, stored in lower-triangular-band-packed storage mode. Vector ***x*** is a vector of length 7. Matrix ***A*** is:

```
                    ┌                                   ┐
                    | 1.0  0.0  0.0  0.0  0.0  0.0  0.0 |
                    | 1.0  2.0  0.0  0.0  0.0  0.0  0.0 |
                    | 1.0  2.0  3.0  0.0  0.0  0.0  0.0 |
                    | 1.0  2.0  3.0  4.0  0.0  0.0  0.0 |
                    | 0.0  2.0  3.0  4.0  5.0  0.0  0.0 |
                    | 0.0  0.0  3.0  4.0  5.0  6.0  0.0 |
                    | 0.0  0.0  0.0  4.0  5.0  6.0  7.0 |
                    └                                   ┘
```

Call Statement and Input:

```
             UPLO TRANSA DIAG  N   K   A  LDA  X  INCX
              |     |     |    |   |   |   |   |   |
CALL STBMV( 'L' , 'T' , 'N' , 7 , 3 , A , 5 , X , 1  )

                                                 ┐
          | 1.0  2.0  3.0  4.0  5.0  6.0  7.0 |
          | 1.0  2.0  3.0  4.0  5.0  6.0   .  |
A    =    | 1.0  2.0  3.0  4.0  5.0   .    .  |
          | 1.0  2.0  3.0  4.0   .    .    .  |
          |  .    .    .    .    .    .    .  |
                                                 ┘

X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)
```

Output:

```
X        =  (10.0, 28.0, 54.0, 88.0, 90.0, 78.0, 49.0)
```

Example 3

This example shows the computation ***x***←***A***^{H}***x***. Matrix ***A*** is a complex 7 by 7 upper triangular band matrix with a half band width of 3 that is not unit triangular, stored in upper-triangular-band-packed storage mode. Vector ***x*** is a vector of length 7. Matrix ***A*** is:

```
                                                                                   ┐
          | (1.0, 1.0) (1.0, 1.0) (1.0, 1.0) (1.0, 1.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) |
          | (0.0, 0.0) (2.0, 2.0) (2.0, 2.0) (2.0, 2.0) (2.0, 2.0) (0.0, 0.0) (0.0, 0.0) |
          | (0.0, 0.0) (0.0, 0.0) (3.0, 3.0) (3.0, 3.0) (3.0, 3.0) (3.0, 3.0) (0.0, 0.0) |
          | (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (4.0, 4.0) (4.0, 4.0) (4.0, 4.0) (4.0, 4.0) |
          | (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (5.0, 5.0) (5.0, 5.0) (5.0, 5.0) |
          | (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (6.0, 6.0) (6.0, 6.0) |
          | (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (7.0, 7.0) |
                                                                                   ┘
```

Call Statement and Input:

```
            UPLO TRANSA DIAG  N   K   A  LDA  X  INCX
             |     |     |    |   |   |   |   |   |
CALL CTBMV( 'U' , 'C' , 'N' , 7 , 3 , A , 5 , X , 1  )
```

```

┐
        |     .           .           .       (1.0, 1.0)  (2.0, 2.0)  (3.0, 3.0)  (4.0,
4.0) |
        |     .           .       (1.0, 1.0)  (2.0, 2.0)  (3.0, 3.0)  (4.0, 4.0)  (5.0,
5.0) |
A    =  |     .       (1.0, 1.0)  (2.0, 2.0)  (3.0, 3.0)  (4.0, 4.0)  (5.0, 5.0)  (6.0,
6.0) |
        | (1.0, 1.0)  (2.0, 2.0)  (3.0, 3.0)  (4.0, 4.0)  (5.0, 5.0)  (6.0, 6.0)  (7.0,
7.0) |

|     .           .           .           .           .           .           .      |

┘
```

```
X        =  ((1.0, 2.0), (2.0, 4.0), (3.0, 6.0), (4.0, 8.0),
             (5.0, 10.0), (6.0, 12.0), (7.0, 14.0))
```

Output:

```
X        =  ((1.0, 2.0), (7.0, 9.0), (24.0, 23.0), (58.0, 46.0),
             (112.0, 79.0), (186.0, 122.0), (280.0, 175.0))
```

Example 4

This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{A}^T\boldsymbol{x}$, where $k > n$. Matrix ***A*** is a real 4 by 4 upper triangular band matrix with a half band width of 5 that is not unit triangular, stored in upper-triangular-band-packed storage mode. Vector ***x*** is a vector of length 4. Matrix ***A*** is:

```
                          ┌                   ┐
                          | 1.0  1.0  1.0  1.0 |
                          |  .   2.0  2.0  2.0 |
                          |  .    .   3.0  3.0 |
                          |  .    .    .   4.0 |
                                              ┘
```

Call Statement and Input:

```
            UPLO TRANSA DIAG  N   K   A  LDA  X  INCX
             |     |     |    |   |   |   |   |   |
CALL STBMV( 'U' , 'T' , 'N' , 4 , 5 , A , 6 , X , 1  )

                                  ┐
        |  .    .    .    .      |
A    =  |  .    .    .    .      |
        |  .    .    .   1.0     |
        |  .    .   1.0  2.0     |
        |  .   1.0  2.0  3.0     |
        | 1.0  2.0  3.0  4.0     |
                                  ┘

X        =  (1.0, 2.0, 3.0, 4.0)
```

Output:

```
X        =  (1.0, 5.0, 14.0, 30.0)
```

STBSV, DTBSV, CTBSV, and ZTBSV (Triangular Band Equation Solve)

Purpose

STBSV and DTBSV solve one of the following triangular banded systems of equations with a single right-hand side, using the vector ***x*** and triangular band matrix ***A*** or its transpose:

Solution	Equation
1. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$	$\boldsymbol{Ax} = \boldsymbol{b}$
2. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$	$\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$

CTBSV and ZTBSV solve one of the following triangular banded systems of equations with a single right-hand side, using the vector ***x*** and triangular band matrix ***A***, its transpose, or its conjugate transpose:

Solution	Equation
1. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$	$\boldsymbol{Ax} = \boldsymbol{b}$
2. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$	$\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$
3. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-H}\boldsymbol{x}$	$\boldsymbol{A}^H\boldsymbol{x} = \boldsymbol{b}$

Matrix ***A*** can be either upper or lower triangular and is stored in upper- or lower-triangular-band-packed storage mode, respectively.

Table 101: Data Types.

Data Types	
A, ***x***	**Subprogram**
Short-precision real	STBSV
Long-precision real	DTBSV
Short-precision complex	CTBSV
Long-precision complex	ZTBSV

Syntax

Language	Syntax
Fortran	CALL STBSV | DTBSV | CTBSV | ZTBSV (*uplo*, *trans*, *diag*, *n*, *k*, *a*, *lda*, *x*, *incx*)
C and C++	stbsv | dtbsv | ctbsv | ztbsv (*uplo*, *trans*, *diag*, *n*, *k*, *a*, *lda*, *x*, *incx*);f
CBLAS	cblas_stbsv | cblas_dtbsv | cblas_ctbsv | cblas_ztbsv (*cblas_layout*, *cblas_uplo*, *cblas_trans*, *cblas_diag*, *n*, *k*, *a*, *lda*, *x*, *incx*);

On Entry

cblas_layout

indicates whether the input matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

uplo

indicates whether matrix ***A*** is an upper or lower triangular band matrix, where:

If *uplo* = 'U', ***A*** is an upper triangular matrix.

If *uplo* = 'L', ***A*** is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

cblas_uplo
indicates whether matrix ***A*** is an upper or lower triangular matrix, where:

If *cblas_uplo* = CblasUpper, ***A*** is an upper triangular matrix.

If *cblas_uplo* = CblasLower, ***A*** is a lower triangular matrix.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

trans
indicates the form of matrix ***A*** used in the system of equations, where:

If *trans* = 'N', ***A*** is used, resulting in solution 1.

If *trans* = 'T', $\boldsymbol{A}^T$ is used, resulting in solution 2.

If *trans* = 'C', $\boldsymbol{A}^H$ is used, resulting in solution 3.

Specified as: a single character. It must be 'N', 'T', or 'C'.

cblas_transa
indicates the form of matrix ***A*** to use in the computation, where:

If *cblas_transa* = CblasNoTrans, ***A*** is used, resulting in solution 1.

If *cblas_transa* = CblasTrans, $\boldsymbol{A}^T$ is used, resulting in solution 2.

If *cblas_transa* = CblasConjTrans, $\boldsymbol{A}^H$is used, resulting in solution 3.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

cblas_diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = CblasUnit, ***A*** is a unit triangular matrix.

If *diag* = CblasNonUnit ***A*** is not a unit triangular matrix.

Specified as: an object of enumerated type CBLAS_DIAG. It must be CblasNonUnit or CblasUnit.

n
is the order of triangular band matrix ***A***. Specified as: an integer; $n \geq 0$.

k
is the upper or lower band width *k* of the matrix ***A***. Specified as: an integer; $k \geq 0$.

a
is the upper or lower triangular band matrix ***A*** of order *n*, stored in upper- or lower-triangular-band-packed storage mode, respectively. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 101 on page 366.

lda
is the leading dimension of the array specified for *a*. Specified as: an integer; $lda > 0$ and $lda \geq k+1$.

x
is the vector ***x*** of length *n*, containing the right-hand side of the triangular system to be solved. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 101 on page 366.

incx
is the stride for vector ***x***. Specified as: an integer; *incx* > 0 or *incx* < 0.

On Return

x
is the solution vector ***x*** of length *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 101 on page 366.

Notes

1. These subroutines accept lowercase letters for the *uplo*, *trans*, and *diag* arguments.
2. For STBSV and DTBSV, if you specify 'C' for the *trans* argument, it is interpreted as though you specified 'T'.
3. Matrix ***A*** and vector ***x*** must have no common elements; otherwise, results are unpredictable.
4. For unit triangular matrices, the elements of the diagonal are assumed to be 1.0 for real matrices and (1.0, 0.0) for complex matrices, and you do not need to set these values in the array.
5. For both upper and lower triangular band matrices, if you specify $k \geq n$, ESSL assumes, **for purposes of the computation only**, that the upper or lower band width of matrix ***A*** is *n*-1; that is, it processes matrix ***A*** of order *n*, as though it is a (nonbanded) triangular matrix. However, ESSL uses the original value for *k* **for the purposes of finding the locations** of element a_{11} and all other elements in the array specified for ***A***, as described in "Triangular Band Matrix" on page 102. For an illustration of this technique, see Example 3.
6. For a description of triangular band matrices and how they are stored in upper- and lower-triangular-band-packed storage mode, see "Triangular Band Matrix" on page 102.
7. If you are using a lower triangular band matrix, it may save your program some time if you use this alternate approach instead of using lower-triangular-band-packed storage mode. Leave matrix ***A*** in full-matrix storage mode when you pass it to ESSL and specify the *lda* argument to be *lda*+1, which is the leading dimension of matrix ***A*** plus 1. ESSL then processes the matrix elements in the same way as though you had set them up in lower-triangular-band-packed storage mode.

Function

These subroutines solve a triangular banded system of equations with a single right-hand side. The solution, ***x***, may be any of the following, where triangular band matrix ***A***, its transpose, or its conjugate transpose is used, and where ***A*** can be either upper- or lower-triangular:

1. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$
2. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$
3. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-H}\boldsymbol{x}$ (for CTBSV and ZTBSV only)

where:

x is a vector of length *n*.

A is an upper or lower triangular band matrix of order *n*, stored in upper- or lower-triangular-band-packed storage mode, respectively.

See references ["44" on page 1255], ["56" on page 1256], and ["48" on page 1255]. If *n* is 0, no computation is performed.

Error conditions

Computational Errors
None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. $n < 0$
3. $k < 0$
4. $lda \le 0$
5. $lda < k+1$
6. $incx = 0$
7. *uplo* ≠ 'L' or 'U'
8. *cblas_uplo* ≠ CblasLower or CblasUpper
9. *trans* ≠ 'T', 'N', or 'C'
10. *cblas_transa* ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
11. *diag* ≠ 'N' or 'U'
12. *cblas_diag* ≠ CblasNonUnit or CblasUnit

Examples

Example 1

This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$. Matrix ***A*** is a real 9 by 9 upper triangular band matrix with an upper band width of 2 that is not unit triangular, stored in upper-triangular-band-packed storage mode. Vector ***x*** is a vector of length 9, where matrix ***A*** is:

```
              ┌                                             ┐
              | 1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0 |
              | 0.0  4.0  2.0  3.0  0.0  0.0  0.0  0.0  0.0 |
              | 0.0  0.0  4.0  1.0  1.0  0.0  0.0  0.0  0.0 |
              | 0.0  0.0  0.0  4.0  2.0  2.0  0.0  0.0  0.0 |
              | 0.0  0.0  0.0  0.0  3.0  1.0  1.0  0.0  0.0 |
              | 0.0  0.0  0.0  0.0  0.0  3.0  2.0  2.0  0.0 |
              | 0.0  0.0  0.0  0.0  0.0  0.0  3.0  1.0  0.0 |
              | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  2.0  2.0 |
              | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0 |
              └                                             ┘
```

Call Statement and Input:

```
            UPLO TRANS  DIAG  N   K   A  LDA  X  INCX
             |     |     |    |   |   |   |   |   |
CALL STBSV( 'U' , 'N' , 'N' , 9 , 2 , A , 3 , X , 1  )
```

```
         ┌                                             ┐
         |  .    .   1.0  3.0  1.0  2.0  1.0  2.0  0.0 |
A    =   |  .   1.0  2.0  1.0  2.0  1.0  2.0  1.0  2.0 |
         | 1.0  4.0  4.0  4.0  3.0  3.0  3.0  2.0  1.0 |
         └                                             ┘

X         =  (2.0, 7.0, 1.0, 8.0, 2.0, 8.0, 1.0, 8.0, 3.0)
```

Output:

```
X         =  (1.0, 1.0, 0.0, 1.0, 0.0, 2.0, 0.0, 1.0, 3.0)
```

Example 2

This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$, solving the same system as in Example 1. Matrix ***A*** is a real 9 by 9 lower triangular band matrix with a lower band width of 2 that is not unit triangular, stored in lower-triangular-band-packed storage mode. Vector ***x*** is a vector of length 9 where matrix ***A*** is:

```
              ┌                                             ┐
              | 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
              | 1.0  4.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
              | 1.0  2.0  4.0  0.0  0.0  0.0  0.0  0.0  0.0 |
              | 0.0  3.0  1.0  4.0  0.0  0.0  0.0  0.0  0.0 |
              | 0.0  0.0  1.0  2.0  3.0  0.0  0.0  0.0  0.0 |
              | 0.0  0.0  0.0  2.0  1.0  3.0  0.0  0.0  0.0 |
              | 0.0  0.0  0.0  0.0  1.0  2.0  3.0  0.0  0.0 |
              | 0.0  0.0  0.0  0.0  0.0  2.0  1.0  2.0  0.0 |
              | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  2.0  1.0 |
              └                                             ┘
```

Call Statement and Input:

```
            UPLO TRANS  DIAG  N   K   A  LDA  X  INCX
             |     |     |    |   |   |   |   |   |
CALL STBSV( 'L' , 'T' , 'N' , 9 , 2 , A , 3 , X , 1  )
```

```
         ┌                                             ┐
         | 1.0  4.0  4.0  4.0  3.0  3.0  3.0  2.0  1.0 |
A    =   | 1.0  2.0  1.0  2.0  1.0  2.0  1.0  2.0   .  |
         | 1.0  3.0  1.0  2.0  1.0  2.0  0.0   .    .  |
         └                                             ┘
```

X = (same as input X in Example 1)

Output:

X = (same as output X in Example 1)

Example 3

This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$, where $k > n$. Matrix ***A*** is a real 4 by 4 upper triangular band matrix with an upper band width of 3, even though *k* is specified as 5. It is not unit triangular and is stored in upper-triangular-band-packed storage mode. Vector ***x*** is a vector of length 4 where matrix ***A*** is:

```
                          ┌                    ┐
                          | 1.0  2.0  3.0  2.0 |
                          | 0.0  2.0  2.0  5.0 |
                          | 0.0  0.0  3.0  3.0 |
                          | 0.0  0.0  0.0  1.0 |
                          └                    ┘
```

Call Statement and Input:

```
            UPLO TRANS  DIAG  N   K   A  LDA  X  INCX
             |     |     |    |   |   |   |   |   |
CALL STBSV( 'U' , 'T' , 'N' , 4 , 5 , A , 6 , X , 1  )
```

```
         ┌                    ┐
         |  .    .    .    .  |
         |  .    .    .    .  |
A    =   |  .    .    .   2.0 |
         |  .    .   3.0  5.0 |
         |  .   2.0  2.0  3.0 |
         | 1.0  2.0  3.0  1.0 |
         └                    ┘

X        =  (5.0, 18.0, 32.0, 41.0)
```

Output:

```
X        =  (5.0, 4.0, 3.0, 2.0)
```

Example 4

This example shows the solution ***x***←***A***$^{-T}$***x***. Matrix ***A*** is a complex 7 by 7 lower triangular band matrix with a lower band width of 3 that is not unit triangular, stored in lower-triangular-band-packed storage mode. Vector ***x*** is a vector of length 7. Matrix ***A*** is:

```
       ┌                                                                      ┐
       | (1.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) |
       | (1.0, 2.0) (2.0, 1.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) |
       | (1.0, 3.0) (2.0, 2.0) (3.0, 1.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) |
       | (1.0, 4.0) (2.0, 3.0) (3.0, 3.0) (4.0, 1.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) |
       | (0.0, 0.0) (2.0, 4.0) (3.0, 3.0) (4.0, 2.0) (2.0, 1.0) (0.0, 0.0) (0.0, 0.0) |
       | (0.0, 0.0) (0.0, 0.0) (3.0, 3.0) (4.0, 3.0) (5.0, 1.0) (3.0, 1.0) (0.0, 0.0) |
       | (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (4.0, 4.0) (5.0, 2.0) (6.0, 1.0) (2.0, 1.0) |
       └                                                                      ┘
```

Call Statement and Input:

```
             UPLO TRANS  DIAG  N   K   A  LDA  X  INCX
              |     |     |    |   |   |   |   |   |
CALL CTBSV( 'L' , 'T' , 'N' , 7 , 3 , A , 4 , X , 1  )
```

```
         ┌                                                                      ┐
         | (1.0, 0.0) (2.0, 1.0) (3.0, 1.0) (4.0, 1.0) (2.0, 1.0) (3.0, 1.0) (2.0, 1.0) |
A    =   | (1.0, 2.0) (2.0, 2.0) (3.0, 3.0) (4.0, 2.0) (5.0, 1.0) (6.0, 1.0)     .      |
         | (1.0, 3.0) (2.0, 3.0) (3.0, 3.0) (4.0, 3.0) (5.0, 2.0)     .          .      |
         | (1.0, 4.0) (2.0, 4.0) (3.0, 3.0) (4.0, 4.0)     .          .          .      |
         └                                                                      ┘
```

```
X        =  ((2.0, 2.0), (7.0, 1.0), (1.0, 1.0), (8.0, 1.0),
             (2.0, 0.0), (8.0, 1.0), (1.0, 2.0))
```

Output:

```
X        =  ((-12.048, -13.136), (6.304, -1.472), (-1.880, 1.040),
             (2.600, -1.800), (-2.160, 1.880), (0.800, -1.400),
             (0.800, 0.600))
```

Sparse Matrix-Vector Subprograms

This contains the sparse matrix-vector subprogram descriptions.

DSMMX (Matrix-Vector Product for a Sparse Matrix in Compressed-Matrix Storage Mode)

Purpose

This subprogram computes the matrix-vector product for sparse matrix ***A***, stored in compressed-matrix storage mode, using the matrix and vectors ***x*** and ***y***:

y←***Ax***

where ***A***, ***x***, and ***y*** contain long-precision real numbers. You can use DSMTM to transpose matrix ***A*** before calling this subroutine. The resulting computation performed by this subroutine is then ***y***←***A***^{T}***x***.

Syntax

Language	**Syntax**
Fortran	CALL DSMMX (*m*, *nz*, *ac*, *ka*, *lda*, *x*, *y*)
C and C++	dsmmx (*m*, *nz*, *ac*, *ka*, *lda*, *x*, *y*);

On Entry

m

is the number of rows in sparse matrix ***A*** and the number of elements in vector ***y***. Specified as: an integer; $m \geq 0$.

nz

is the maximum number of nonzero elements in each row of sparse matrix ***A***. Specified as: an integer; $nz \geq 0$.

ac

is the *m* by *n* sparse matrix ***A***, stored in compressed-matrix storage mode in an array, referred to as AC. Specified as: an *lda* by (at least) *nz* array, containing long-precision real numbers.

ka

is the array, referred to as KA, containing the column numbers of the matrix ***A*** elements stored in the corresponding positions in array AC. Specified as: an *lda* by (at least) *nz* array, containing integers, where $1 \leq$ (elements of KA) $\leq n$.

lda

is the size of the leading dimension of the arrays specified for *ac* and *ka*. Specified as: an integer; $lda > 0$ and $lda \geq m$.

x

is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length *n*, containing long-precision real numbers.

y

See On Return.

On Return

y

is the vector ***y*** of length *m*, containing the result of the computation. Returned as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

Notes

1. Matrix ***A*** must have no common elements with vectors ***x*** and ***y***; otherwise, results are unpredictable.
2. For the KA array, where there are no corresponding nonzero elements in AC, you must still fill in a number between 1 and *n*. See the Example.
3. For a description of how sparse matrices are stored in compressed-matrix storage mode, see "Compressed-Matrix Storage Mode" on page 110.
4. If your sparse matrix is stored by rows, as defined in "Storage-by-Rows" on page 115, you should first use the DSRSM utility subroutine, described in "DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)" on page 1212, to convert your sparse matrix to compressed-matrix storage mode.

Function

The matrix-vector product is computed for a sparse matrix, stored in compressed matrix mode:

$\boldsymbol{y} \leftarrow \boldsymbol{A}\boldsymbol{x}$

where:

A is an *m* by *n* sparse matrix, stored in compressed-matrix storage mode in arrays AC and KA.

x is a vector of length *n*.

y is a vector of length *m*.

It is expressed as follows:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_m \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

See reference [“89” on page 1258]. If *m* is 0, no computation is performed; if *nz* is 0, output vector ***y*** is set to zero, because matrix ***A*** contains all zeros.

If your program uses a sparse matrix stored by rows and you want to use this subroutine, you should first convert your sparse matrix to compressed-matrix storage mode by using the DSRSM utility subroutine described in “DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)” on page 1212.

Error conditions

Computational Errors
None

Input-Argument Errors

1. $m < 0$
2. $lda \le 0$
3. $m > lda$
4. $nz < 0$

Examples

Example

This example shows the matrix-vector product computed for the following sparse matrix ***A***, which is stored in compressed-matrix storage mode in arrays AC and KA. Matrix ***A*** is:

```
                     | 4.0  0.0  7.0  0.0  0.0  0.0 |
                     | 3.0  4.0  0.0  2.0  0.0  0.0 |
                     | 0.0  2.0  4.0  0.0  4.0  0.0 |
                     | 0.0  0.0  7.0  4.0  0.0  1.0 |
                     | 1.0  0.0  0.0  3.0  4.0  0.0 |
                     | 1.0  1.0  0.0  0.0  3.0  4.0 |
```

Call Statement and Input:

```
           M   NZ  AC   KA  LDA  X   Y
           |   |   |    |    |   |   |
CALL DSMMX( 6 , 4 , AC , KA , 6 , X , Y )
```

```
        | 4.0  7.0  0.0  0.0 |
        | 4.0  3.0  2.0  0.0 |
AC   =  | 4.0  2.0  4.0  0.0 |
        | 4.0  7.0  1.0  0.0 |
        | 4.0  1.0  3.0  0.0 |
        | 4.0  1.0  1.0  3.0 |
```

```
        | 1  3  1  1 |
        | 2  1  4  1 |
KA   =  | 3  2  5  1 |
        | 4  3  6  1 |
        | 5  1  4  1 |
        | 6  1  2  5 |

X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
```

Output:

```
Y        =  (25.0, 19.0, 36.0, 43.0, 33.0, 42.0)
```

DSDMX (Matrix-Vector Product for a Sparse Matrix or Its Transpose in Compressed-Diagonal Storage Mode)

Purpose

This subprogram computes the matrix-vector product for square sparse matrix ***A***, stored in compressed-diagonal storage mode, using either the matrix or its transpose, and vectors ***x*** and ***y***:

$\boldsymbol{y} \leftarrow \boldsymbol{A}\boldsymbol{x}$
$\boldsymbol{y} \leftarrow \boldsymbol{A}^{T}\boldsymbol{x}$

where ***A***, ***x***, and ***y*** contain long-precision real numbers.

Syntax

Language	**Syntax**
Fortran	CALL DSDMX (*iopt*, *n*, *nd*, *ad*, *lda*, *trans*, *la*, *x*, *y*)
C and C++	dsdmx (*iopt*, *n*, *nd*, *ad*, *lda*, *trans*, *la*, *x*, *y*);

On Entry

iopt

indicates the storage variation used for sparse matrix ***A***, stored in compressed-diagonal storage mode, where:

If *iopt* = 0, matrix ***A*** is a general sparse matrix, where all the nonzero diagonals in matrix ***A*** are used to set up the storage arrays.

If *iopt* = 1, matrix ***A*** is a symmetric sparse matrix, where only the nonzero main diagonal and one of each of the unique nonzero diagonals are used to set up the storage arrays.

Specified as: an integer; *iopt* = 0 or 1.

n

is the order of sparse matrix ***A*** and the number of elements in vectors ***x*** and ***y***. Specified as: an integer; $n \geq 0$.

nd

is the number of diagonals stored in the columns of array AD, as well as the number of columns in AD and the number of elements in array LA. Specified as: an integer; $nd \geq 0$.

ad

is the sparse matrix ***A*** of order *n*, stored in compressed diagonal storage in an array, referred to as AD. The *iopt* argument indicates the storage variation used for storing matrix ***A***. The *trans* argument indicates the following:

If *trans* = 'N', ***A*** is used in the computation.

If *trans* = 'T', $\boldsymbol{A}^{T}$ is used in the computation.

Note: No data should be moved to form $\boldsymbol{A}^{T}$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: an *lda* by (at least) *nd* array, containing long-precision real numbers; $lda \geq n$.

lda
is the size of the leading dimension of the array specified for *ad*. Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

trans
indicates the form of matrix ***A*** to use in the computation, where:

If *trans* = 'N', ***A*** is used in the computation.

If *trans* = 'T', $\boldsymbol{A}^T$ is used in the computation.

Specified as: a single character; *trans* = 'N' or 'T'.

la
is the array, referred to as LA, containing the diagonal numbers *k* for the diagonals stored in each corresponding column in array AD. (For an explanation of how diagonal numbers are assigned, see "Compressed-Diagonal Storage Mode" on page 111.)

Specified as: a one-dimensional array of (at least) length *nd*, containing integers; 1-*n* ≤ LA(*i*) ≤ *n*-1.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array, containing long-precision real numbers.

y
See On Return.

On Return

y
is the vector ***y*** of length *n*, containing the result of the computation. Returned as: a one-dimensional array, containing long-precision real numbers.

Notes

1. All subroutines accept lowercase letters for the *trans* argument.
2. Matrix ***A*** must have no common elements with vectors ***x*** and ***y***; otherwise, results are unpredictable.
3. For a description of how sparse matrices are stored in compressed-diagonal storage mode, see "Compressed-Diagonal Storage Mode" on page 111.

Function

The matrix-vector product of a square sparse matrix or its transpose, is computed for a matrix stored in compressed-diagonal storage mode:

$\boldsymbol{y} \leftarrow \boldsymbol{A}\boldsymbol{x}$
$\boldsymbol{y} \leftarrow \boldsymbol{A}^T\boldsymbol{x}$

where:

A is a sparse matrix of order *n*, stored in compressed-diagonal storage mode in AD and LA, using the storage variation for either general or symmetric sparse matrices, as indicated by the *iopt* argument.

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

It is expressed as follows for $\boldsymbol{y} \leftarrow \boldsymbol{A}\boldsymbol{x}$:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

It is expressed as follows for $\boldsymbol{y} \leftarrow \boldsymbol{A}^T \boldsymbol{x}$:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{n1} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{1n} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

If *n* is 0, no computation is performed; if *nd* is 0, output vector ***y*** is set to zero, because matrix ***A*** contains all zeros.

Error conditions

Computational Errors

None

Input-Argument Errors

1. *iopt* ≠ 0 or 1
2. *n* < 0
3. *lda* ≤ 0
4. *n* > *lda*
5. *trans* ≠ 'N' or 'T'
6. *nd* < 0
7. LA(*j*) ≤ -*n* or LA(*j*) ≥ *n*, for any *j* = 1, *n*

Examples

Example 1

This example shows the matrix-vector product using *trans* = 'N', which is computed for the following sparse matrix ***A*** of order 6. The matrix is stored in compressed-matrix storage mode in arrays AD and LA using the storage variation for general sparse matrices, storing all nonzero diagonals. Matrix ***A*** is:

```
                     | 4.0  0.0  7.0  0.0  0.0  0.0 |
                     | 3.0  4.0  0.0  2.0  0.0  0.0 |
                     | 0.0  2.0  4.0  0.0  4.0  0.0 |
                     | 0.0  0.0  7.0  4.0  0.0  1.0 |
                     | 1.0  0.0  0.0  3.0  4.0  0.0 |
                     | 1.0  1.0  0.0  0.0  3.0  4.0 |
```

Call Statement and Input:

```
           IOPT  N   ND  AD  LDA TRANS  LA   X   Y
             |   |   |   |    |    |    |    |   |
CALL DSDMX( 0  , 6 , 5 , AD , 6 , 'N' , LA , X , Y )
```

```
        | 4.0  0.0  0.0  0.0  7.0 |
        | 4.0  0.0  0.0  3.0  2.0 |
AD   =  | 4.0  0.0  0.0  2.0  4.0 |
        | 4.0  0.0  0.0  7.0  1.0 |
        | 4.0  0.0  1.0  3.0  0.0 |
        | 4.0  1.0  1.0  3.0  0.0 |
```

```
LA       =  (0, -5, -4, -1, 2)
X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
```

Output:

```
Y        =  (25.0, 19.0, 36.0, 43.0, 33.0, 42.0)
```

Example 2

This example shows the matrix-vector product using *trans* = 'N', which is computed for the following sparse matrix ***A*** of order 6. The matrix is stored in compressed-matrix storage mode in arrays AD and LA using the storage variation for symmetric sparse matrices, storing the nonzero main diagonal and one of each of the unique nonzero diagonals. Matrix ***A*** is:

```
                  | 11.0   0.0  13.0   0.0  15.0   0.0 |
                  |  0.0  22.0   0.0  24.0   0.0  26.0 |
                  | 13.0   0.0  33.0   0.0  35.0   0.0 |
                  |  0.0  24.0   0.0  44.0   0.0  46.0 |
                  | 15.0   0.0  35.0   0.0  55.0   0.0 |
                  |  0.0  26.0   0.0  46.0   0.0  66.0 |
```

Call Statement and Input:

```
           IOPT  N   ND  AD  LDA TRANS  LA   X   Y
             |   |   |   |    |    |    |    |   |
CALL DSDMX( 1  , 6 , 3 , AD , 6 , 'N' , LA , X , Y )
```

```
        | 11.0  13.0   0.0 |
        | 22.0  24.0   0.0 |
AD   =  | 33.0  35.0   0.0 |
        | 44.0  46.0   0.0 |
        | 55.0   0.0  15.0 |
        | 66.0   0.0  26.0 |
```

```
LA       =  (0, 2, -4)
X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
```

Output:

```
Y        =  (125.0, 296.0, 287.0, 500.0, 395.0, 632.0)
```

Example 3

This example is the same as Example 1 except that it shows the matrix-vector product for the transpose of a matrix, using *trans* = 'T'. It is computed using the transpose of the following sparse matrix ***A*** of order 6, which is stored in compressed-matrix storage mode in arrays AD and LA, using the storage variation for general sparse matrices, storing all nonzero diagonals. It uses the same matrix ***A*** as in Example 1.

Call Statement and Input:

```
            IOPT  N   ND  AD  LDA TRANS  LA   X   Y
             |    |   |   |    |    |    |    |   |
CALL DSDMX( 0   , 6 , 5 , AD , 6 , 'T' , LA , X , Y )
```

AD =(same as input AD in Example 1)
LA =(same as input LA in Example 1)
X =(same as input X in Example 1)

Output:

```
Y        =  (21.0, 20.0, 47.0, 35.0, 50.0, 28.0)
```

DSMTM (Transpose a Sparse Matrix in Compressed-Matrix Storage Mode)

Purpose

This subprogram transposes sparse matrix ***A***, stored in compressed-matrix storage mode, where ***A*** contains long-precision real numbers.

Syntax

Language	Syntax
Fortran	CALL DSMTM (*m*, *nz*, *ac*, *ka*, *lda*, *n*, *nt*, *at*, *kt*, *ldt*, *aux*, *naux*)
C and C++	dsmtm (*m*, *nz*, *ac*, *ka*, *lda*, *n*, *nt*, *at*, *kt*, *ldt*, *aux*, *naux*);

On Entry

m
is the number of rows in sparse matrix ***A***. Specified as: an integer; $m \geq 0$.

nz
is the maximum number of nonzero elements in each row of sparse matrix ***A***. Specified as: an integer; $nz \geq 0$.

ac
is the *m* by *n* sparse matrix ***A***, stored in compressed-matrix storage mode in an array, referred to as AC. Specified as: an *lda* by (at least) *nz* array, containing long-precision real numbers.

ka
is the array, referred to as KA, containing the column numbers of the matrix ***A*** elements stored in the corresponding positions in array AC. Specified as: an *lda* by (at least) *nz* array, containing integers, where 1 ≤ (elements of KA) ≤ *n*.

lda
is the size of the leading dimension of the arrays specified for *ac* and *ka*. Specified as: an integer; $lda > 0$ and $lda \geq m$.

n
is the number of columns in sparse matrix ***A***. Specified as: an integer; $0 \leq n \leq ldt$ and *n* ≥ (maximum column index in KA).

nt
is the number of columns in output arrays AT and KT that are available for use. Specified as: an integer; $nt > 0$.

at
See On Return.

kt
See On Return.

ldt
is the size of the leading dimension of the arrays specified for *at* and *kt*. Specified as: an integer; $ldt > 0$ and $ldt \geq n$.

aux
has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is a storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing long-precision real numbers. They can have any value.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, DSMTM dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, $naux \geq n$.

On Return

n
is the number of rows in the transposed matrix $\boldsymbol{A}^T$. Returned as: an integer; *n* = (maximum column index in KA).

nt
is the maximum number of nonzero elements, *nt*, in each row of the transposed matrix $\boldsymbol{A}^T$. Returned as: an integer; $nt \leq m$.

at
is the *n* by (at least) *m* sparse matrix transpose $\boldsymbol{A}^T$, stored in compressed-matrix storage mode in an array, referred to as AT. Returned as: an *ldt* by (at least) *nt* array, containing long-precision real numbers.

kt
is the array, referred to as KT, containing the column numbers of the transposed matrix $\boldsymbol{A}^T$ elements, stored in the corresponding positions in array AT. Returned as: an *ldt* by (at least) *nt* array, containing integers, where $1 \leq$ (elements of KT) $\leq m$.

Notes

1. In your C program, arguments *n* and *nt* must be passed by reference.
2. The value specified for input argument *nt* should be greater than or equal to the number of nonzero elements you estimate to be in each row of the transposed sparse matrix $\boldsymbol{A}^T$. The output value is less than or equal to the input value you specify.
3. For the KA array, where there are no corresponding nonzero elements in AC, you must still fill in a number between 1 and *n*. See the Example.
4. For a description of how sparse matrices are stored in compressed-matrix storage mode, see "Compressed-Matrix Storage Mode" on page 110.
5. If your sparse matrix is stored by rows, as defined in "Storage-by-Rows" on page 115, you should first use the DSRSM utility subroutine, described in "DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)" on page 1212, to convert your sparse matrix to compressed-matrix storage mode.
6. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

A sparse matrix ***A***, stored in arrays AC and KA in compressed-matrix storage mode, is transposed, forming $\boldsymbol{A}^T$, and is stored in arrays AT and KT in compressed-matrix storage mode. See reference [“89” on page 1258]. This subroutine is provided for when you want to do a matrix-vector product using a transposed matrix, $\boldsymbol{A}^T$. First, you transpose a matrix, ***A***, using this subroutine, then you call DSMMX with the transposed matrix $\boldsymbol{A}^T$. This results in the following computation being performed: $\boldsymbol{y} \leftarrow \boldsymbol{A}^T\boldsymbol{x}$.

If your program uses a sparse matrix stored by rows and you want to use this subroutine, you should first convert your sparse matrix to compressed-matrix storage mode by using the DSRSM utility subroutine described in “DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)” on page 1212.

Error conditions

Resource Errors

Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors

None

Input-Argument Errors

1. *m*, *n* < 0
2. *lda*, *ldt* < 1
3. *lda* < *m*
4. *ldt* < *n*
5. *nz* < 0
6. *n* is less than the maximum column index in KA.
7. *nt* or *ldt* are too small.
8. When the following two errors occur, arrays AT, KT, and AUX are overwritten:

   naux < *n*
   nt ≤ 0
9. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example

This example shows how to transpose the following 5 by 4 sparse matrix ***A***, which is stored in compressed-matrix storage mode in arrays AC and KA. Matrix ***A*** is:

```
| 11.0   0.0   0.0   0.0 |
| 21.0   0.0  23.0   0.0 |
|  0.0   0.0  33.0  34.0 |
|  0.0  42.0   0.0  44.0 |
| 51.0   0.0  53.0   0.0 |
```

The resulting 4 by 5 matrix transpose $\boldsymbol{A}^T$, stored in compressed-matrix storage mode in arrays AT and KT, is as follows. Matrix $\boldsymbol{A}^T$ is:

```
| 11.0  21.0   0.0   0.0  51.0 |
|  0.0   0.0   0.0  42.0   0.0 |
|  0.0  23.0  33.0   0.0  53.0 |
|  0.0   0.0  34.0  44.0   0.0 |
```

As shown here, the value of N is larger than the actual number of columns in the matrix ***A***. On output, the exact number of rows in the transposed matrix is returned in the output argument N.

On output, row 6 of AT and KT is is not accessed or modified by the subroutine. Column 4 and row 5 are accessed and modified. They are of no use in further computations and will not be used, because NT = 3 and M = 4.

Call Statement and Input:

```
            M   NZ  AC   KA  LDA  N   NT  AT   KT   LDT  AUX  NAUX
            |   |   |    |    |   |   |   |    |     |    |    |
CALL DSMTM( 5 , 2 , AC , KA , 5 , 5 , 4 , AT , KT ,  6 , AUX , 5  )
```

```
        | 11.0   0.0 |
        | 21.0  23.0 |
AC   =  | 33.0  34.0 |
        | 42.0  44.0 |
        | 51.0  53.0 |
```

```
        | 1  1 |
        | 1  3 |
KA   =  | 3  4 |
        | 2  4 |
        | 1  3 |
```

Output:

```
N        =  4
NT       =  3
```

```
        | 11.0  21.0  51.0  0.0 |
        | 42.0   0.0   0.0  0.0 |
AT   =  | 33.0  23.0  53.0  0.0 |
        | 34.0  44.0   0.0  0.0 |
        |  0.0   0.0   0.0  0.0 |
        |   .     .     .    .  |
```

```
        | 1  2  5  1 |
        | 4  1  1  1 |
KT   =  | 3  2  5  1 |
        | 3  4  1  1 |
        | 1  1  1  1 |
        | .  .  .  . |
```

Chapter 9. Matrix operations subroutines

The matrix operations subroutines are described here.

Overview of the Matrix Operation Subroutines

Some of the matrix operation subroutines were designed in accordance with the Level 3 BLAS de facto standard. If these subroutines do not comply with the standard as approved, IBM will consider updating them to do so. If IBM updates these subroutines, the updates could require modifications of the calling application program. For details on the Level 3 BLAS, see reference [“42” on page 1255]. The matrix operation subroutines also include the commonly used matrix operations: addition, subtraction, multiplication, and transposition.

Table 102: List of Matrix Operation Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SGEADD CGEADD	DGEADD ZGEADD	“SGEADD, DGEADD, CGEADD, and ZGEADD (Matrix Addition for General Matrices or Their Transposes)” on page 386
SGESUB CGESUB	DGESUB ZGESUB	“SGESUB, DGESUB, CGESUB, and ZGESUB (Matrix Subtraction for General Matrices or Their Transposes)” on page 392
SGEMUL CGEMUL	DGEMUL ZGEMUL DGEMLP§	“SGEMUL, DGEMUL, CGEMUL, and ZGEMUL (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes)” on page 398
SGEMMS CGEMMS	DGEMMS ZGEMMS	“SGEMMS, DGEMMS, CGEMMS, and ZGEMMS (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm)” on page 408
SGEMM♦GPU CGEMM♦GPU cblas_sgemm♦ cblas_cgemm♦	DGEMM♦GPU ZGEMM♦GPU cblas_dgemm♦ cblas_zgemm♦	“SGEMM, DGEMM, CGEMM, and ZGEMM (Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes)” on page 413
SSYMM♦GPU CSYMM♦GPU CHEMM♦GPU cblas_ssymm♦ cblas_csymm♦ cblas_chemm♦	DSYMM♦GPU ZSYMM♦GPU ZHEMM♦GPU cblas_dsymm♦ cblas_zsymm♦ cblas_zhemm♦	“SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM (Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian)” on page 422
STRMM♦GPU CTRMM♦GPU cblas_strmm♦ cblas_ctrmm♦	DTRMM♦GPU ZTRMM♦GPU cblas_dtrmm♦ cblas_ztrmm♦	“STRMM, DTRMM, CTRMM, and ZTRMM (Triangular Matrix-Matrix Product)” on page 430

Table 102: List of Matrix Operation Subroutines. (continued)

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
STRSM♦GPU CTRSM♦GPU cblas_strsm♦ cblas_ctrsm♦	DTRSM♦GPU ZTRSM♦GPU cblas_dtrsm♦ cblas_ztrsm♦	"STRSM, DTRSM, CTRSM, and ZTRSM (Solution of Triangular Systems of Equations with Multiple Right-Hand Sides)" on page 437
SSYRK♦GPU CSYRK♦GPU CHERK♦GPU cblas_ssyrk♦ cblas_csyrk♦ cblas_cherk♦	DSYRK♦GPU ZSYRK♦GPU ZHERK♦GPU cblas_dsyrk♦ cblas_zsyrk♦ cblas_zherk♦	"SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK (Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)" on page 445
SSYR2K♦GPU CSYR2K♦GPU CHER2K♦GPU cblas_ssyr2k♦ cblas_csyr2k♦ cblas_cher2k♦	DSYR2K♦GPU ZSYR2K♦GPU ZHER2K♦GPU cblas_dsyr2k♦ cblas_zsyr2k♦ cblas_zher2k♦	"SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K (Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)" on page 451
SGETMI CGETMI CGECMI	DGETMI ZGETMI ZGECMI	"SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI and ZGECMI (General Matrix Transpose or Conjugate Transpose [In-Place])" on page 459
SGETMO CGETMO CGECMO	DGETMO ZGETMO ZGECMO	"SGETMO, DGETMO, CGETMO, ZGETMO, CGECMO, and ZGECMO (General Matrix Transpose or Conjugate Transpose [Out-of-Place])" on page 462

♦ Level 3 BLAS

§ This subroutine is provided only for migration from earlier release of ESSL and is not intended for use in new programs.

>|GPU This subroutine can offload computations to the GPUs when the ESSL SMP CUDA Library is used.|<

Use Considerations

This describes some key points about using the matrix operations subroutines.

Specifying Normal, Transposed, or Conjugate Transposed Input Matrices

On each invocation, the matrix operation subroutines can perform one of several possible computations, using different forms of the input matrices ***A*** and ***B***. For the real and complex versions of the subroutines, there are four and nine combinations, respectively, depending on the characters specified for the *transa* and *transb* arguments:

'N'
: Normal form

'T'
: Transposed form

'C'
Conjugate transposed form

The four and nine possible combinations are defined as follows:

Real Combinations	Complex Combinations
$\boldsymbol{AB}$	$\boldsymbol{AB}$
$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}$	$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}$
	$\boldsymbol{A}^{\mathrm{H}}\boldsymbol{B}$
$\boldsymbol{AB}^{\mathrm{T}}$	$\boldsymbol{AB}^{\mathrm{T}}$
$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}^{\mathrm{T}}$	$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}^{\mathrm{T}}$
	$\boldsymbol{A}^{\mathrm{H}}\boldsymbol{B}^{\mathrm{T}}$
	$\boldsymbol{AB}^{\mathrm{H}}$
	$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}^{\mathrm{H}}$
	$\boldsymbol{A}^{\mathrm{H}}\boldsymbol{B}^{\mathrm{H}}$

Transposing or Conjugate Transposing:

This describes some key points about using transposed and conjugate transposed matrices.

On Input

In every case, the input arrays for the matrix, its transpose, or its conjugate transpose should be stored in the original untransposed form. You then specify the desired form of the matrix to be used in the computation in the *transa* or *transb* arguments. For a description of matrix transpose and matrix conjugate transpose, see “Matrices” on page 77.

On Output

If you want to compute the transpose or the conjugate transpose of a matrix operation—that is, the output stored in matrix ***C***—you should use the matrix identities described in “Special Usage” on page 388 for each subroutine description. Examples are provided in the subroutine descriptions to show the use of these matrix identities. This accomplishes the transpose or conjugate transpose as part of the multiply operation.

Performance and Accuracy Considerations

This describes some key points about performance and accuracy in the matrix operations subroutines.

In General

1. The matrix operation subroutines use algorithms that are tuned specifically to the workstation processors they run on. The techniques involve using any one of several computational methods, based on certain operation counts and sizes of data.
2. The short-precision multiplication subroutines provide increased accuracy by partially accumulating results in long precision when the VSX unit is not used.
3. Strassen's method is not stable for certain row or column scalings of the input matrices ***A*** and ***B***. Therefore, for matrices ***A*** and ***B*** with divergent exponent values, Strassen's method may give inaccurate results. For these cases, you should use the _GEMUL or _GEMM subroutines.
4. There are ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see “What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?” on page 61.

For Large Matrices

If you are using large square matrices in your matrix multiplication operations, you may get better performance by using SGEMMS, DGEMMS, CGEMMS, and ZGEMMS. These subroutines use Winograd's variation of Strassen's algorithm for both real and complex matrices.

For Combined Operations

If you want to perform a combined matrix multiplication and addition with scaling, SGEMM, DGEMM, CGEMM, and ZGEMM provide better performance than if you perform the parts of the computation separately in your program. See references [“42” on page 1255] and [“45” on page 1255].

Matrix Operation Subroutines

This contains the matrix operation subroutine descriptions.

SGEADD, DGEADD, CGEADD, and ZGEADD (Matrix Addition for General Matrices or Their Transposes)

Purpose

These subroutines can perform any one of the following matrix additions, using matrices ***A*** and ***B*** or their transposes, and matrix ***C***:

$\boldsymbol{C} \leftarrow \boldsymbol{A}+\boldsymbol{B}$
$\boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}+\boldsymbol{B}$
$\boldsymbol{C} \leftarrow \boldsymbol{A}+\boldsymbol{B}^{\mathrm{T}}$
$\boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}+\boldsymbol{B}^{\mathrm{T}}$

Table 103: Data Types.

Data Types	
A, ***B***, ***C***	**Subroutine**
Short-precision real	SGEADD
Long-precision real	DGEADD
Short-precision complex	CGEADD
Long-precision complex	ZGEADD

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 34.

Syntax

Language	Syntax
Fortran	CALL SGEADD | DGEADD | CGEADD | ZGEADD (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *m*, *n*)
C and C++	sgeadd | dgeadd | cgeadd | zgeadd (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *m*, *n*);

On Entry

a

is the matrix ***A***, where:

If *transa* = 'N', ***A*** is used in the computation, and ***A*** has *m* rows and *n* columns.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation, and ***A*** has *n* rows and *m* columns.

Note: No data should be moved to form $\boldsymbol{A}^T$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 103 on page 386, where:

If *transa* = 'N', its size must be *lda* by (at least) *n*.

If *transa* = 'T', its size must be *lda* by (at least) *m*.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and:

If *transa* = 'N', $lda \geq m$.

If *transa* = 'T', $lda \geq n$.

transa

indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation.

Specified as: a single character; *transa* = 'N' or 'T'.

b

is the matrix ***B***, where:

If *transb* = 'N', ***B*** is used in the computation, and ***B*** has *m* rows and *n* columns.

If *transb* = 'T', $\boldsymbol{B}^T$ is used in the computation, and ***B*** has *n* rows and *m* columns.

Note: No data should be moved to form $\boldsymbol{B}^T$; that is, the matrix ***B*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 103 on page 386, where:

If *transb* = 'N', its size must be *ldb* by (at least) *n*.

If *transb* = 'T', its size must be *ldb* by (at least) *m*.

ldb

is the leading dimension of the array specified for *b*.

Specified as: an integer; *ldb* > 0 and:

If *transb* = 'N', $ldb \geq m$.

If *transb* = 'T', $ldb \geq n$.

transb

indicates the form of matrix ***B*** to use in the computation, where:

If *transb* = 'N', ***B*** is used in the computation.

If *transb* = 'T', $\boldsymbol{B}^T$ is used in the computation.

Specified as: a single character; *transb* = 'N' or 'T'.

c

See On Return.

ldc

is the leading dimension of the array specified for *c*.

Specified as: an integer; *ldc* > 0 and *ldc* ≥ *m*.

m

is the number of rows in matrix ***C***.

Specified as: an integer; 0 ≤ *m* ≤ *ldc*.

n

is the number of columns in matrix ***C***.

Specified as: an integer; 0 ≤ *n*.

On Return

c

is the *m* by *n* matrix ***C***, containing the results of the computation. Returned as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 103 on page 386.

Notes

1. All subroutines accept lowercase letters for the *transa* and *transb* arguments.
2. Matrix ***C*** must have no common elements with matrices ***A*** or ***B***. However, ***C*** may (exactly) coincide with ***A*** if *transa* = 'N', and ***C*** may (exactly) coincide with ***B*** if *transb* = 'N'. Otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

The matrix sum is expressed as follows, where a_{ij}, b_{ij}, and c_{ij} are elements of matrices ***A, B,*** and ***C***, respectively:

$c_{ij} = a_{ij}+b_{ij}$ for $\mathbf{C}\leftarrow\mathbf{A}+\mathbf{B}$
$c_{ij} = a_{ij}+b_{ji}$ for $\mathbf{C}\leftarrow\mathbf{A}+\mathbf{B}^\mathrm{T}$
$c_{ij} = a_{ji}+b_{ij}$ for $\mathbf{C}\leftarrow\mathbf{A}^\mathrm{T}+\mathbf{B}$
$c_{ij} = a_{ji}+b_{ji}$ for $\mathbf{C}\leftarrow\mathbf{A}^\mathrm{T}+\mathbf{B}^\mathrm{T}$
for $i = 1, m$ and $j = 1, n$

If *m* or *n* is 0, no computation is performed.

Special Usage

You can compute the transpose $\mathbf{C}^\mathrm{T}$ of each of the four computations listed under "Function" on page 388 by using the following matrix identities:

$(\mathbf{A}+\mathbf{B})^\mathrm{T} = \mathbf{A}^\mathrm{T}+\mathbf{B}^\mathrm{T}$
$(\mathbf{A}+\mathbf{B}^\mathrm{T})^\mathrm{T} = \mathbf{A}^\mathrm{T}+\mathbf{B}$
$(\mathbf{A}^\mathrm{T}+\mathbf{B})^\mathrm{T} = \mathbf{A}+\mathbf{B}^\mathrm{T}$
$(\mathbf{A}^\mathrm{T}+\mathbf{B}^\mathrm{T})^\mathrm{T} = \mathbf{A}+\mathbf{B}$

Be careful that your output array receiving $\mathbf{C}^\mathrm{T}$ has dimensions large enough to hold the transposed matrix. See Example 4.

Error conditions

Input-Argument Errors

1. *lda*, *ldb*, *ldc* ≤ 0
2. *m*, *n* < 0
3. *m* > *ldc*
4. *transa*, *transb* ≠ 'N' or 'T'
5. *transa* = 'N' and *m* > *lda*
6. *transa* = 'T' and *n* > *lda*
7. *transb* = 'N' and *m* > *ldb*
8. *transb* = 'T' and *n* > *ldb*

Examples

Example 1

This example shows the computation ***C***←***A***+***B***, where ***A*** and ***C*** are contained in larger arrays A and C, respectively, and ***B*** is the same size as array B, in which it is contained.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
             |   |    |     |   |    |     |   |   |   |
CALL SGEADD( A , 6 , 'N'  , B , 4 , 'N'  , C , 5 , 4 , 3 )
```

```
         conref="../transforms/am5xx_conref.dita#am5xx_conref/cg2"/>
        | 110000.0  120000.0  130000.0 |
        | 210000.0  220000.0  230000.0 |
A    =  | 310000.0  320000.0  330000.0 |
        | 410000.0  420000.0  430000.0 |
        |       .         .         .  |
        |       .         .         .  |
         conref="../transforms/am5xx_conref.dita#am5xx_conref/cg4"/>
```

```
        | 11.0  12.0  13.0 |
B    =  | 21.0  22.0  23.0 |
        | 31.0  32.0  33.0 |
        | 41.0  42.0  43.0 |
```

Output:

```
         conref="../transforms/am5xx_conref.dita#am5xx_conref/cg2"/>
        | 110011.0  120012.0  130013.0 |
        | 210021.0  220022.0  230023.0 |
C    =  | 310031.0  320032.0  330033.0 |
        | 410041.0  420042.0  430043.0 |
        |       .         .         .  |
         conref="../transforms/am5xx_conref.dita#am5xx_conref/cg4"/>
```

Example 2

This example shows the computation ***C***←***A***^{T}+***B***, where ***A, B,*** and ***C*** are the same size as arrays A, B, and C, in which they are contained.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
             |   |    |     |   |    |     |   |   |   |
CALL SGEADD( A , 3 , 'T'  , B , 4 , 'N'  , C , 4 , 4 , 3 )
```

```
        | 110000.0  120000.0  130000.0  140000.0 |
A    =  | 210000.0  220000.0  230000.0  240000.0 |
        | 310000.0  320000.0  330000.0  340000.0 |
```

```
        | 11.0  12.0  13.0 |
B    =  | 21.0  22.0  23.0 |
        | 31.0  32.0  33.0 |
        | 41.0  42.0  43.0 |
```

Output:

```
         conref="../transforms/am5xx_conref.dita#am5xx_conref/cg2"/>
        | 110011.0  210012.0  310013.0 |
C    =  | 120021.0  220022.0  320023.0 |
        | 130031.0  230032.0  330033.0 |
        | 140041.0  240042.0  340043.0 |
         conref="../transforms/am5xx_conref.dita#am5xx_conref/cg4"/>
```

Example 3

This example shows computation $\boldsymbol{C} \leftarrow \boldsymbol{A}+\boldsymbol{B}^T$, where ***A*** is contained in a larger array A, and ***B*** and ***C*** are the same size as arrays B and C, in which they are contained.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
             |   |    |     |   |    |     |   |   |   |
CALL SGEADD( A , 5 , 'N'  , B , 3 , 'T'  , C , 4 , 4 , 3 )
```

```
         conref="../transforms/am5xx_conref.dita#am5xx_conref/cg2"/>
        | 110000.0  120000.0  130000.0 |
        | 210000.0  220000.0  230000.0 |
A    =  | 310000.0  320000.0  330000.0 |
        | 410000.0  420000.0  430000.0 |
        |       .         .         .  |
         conref="../transforms/am5xx_conref.dita#am5xx_conref/cg4"/>
```

```
        | 11.0  12.0  13.0  14.0 |
B    =  | 21.0  22.0  23.0  24.0 |
        | 31.0  32.0  33.0  34.0 |
```

Output:

```
         conref="../transforms/am5xx_conref.dita#am5xx_conref/cg2"/>
        | 110011.0  120021.0  130031.0 |
C    =  | 210012.0  220022.0  230032.0 |
        | 310013.0  320023.0  330033.0 |
        | 410014.0  420024.0  430034.0 |
         conref="../transforms/am5xx_conref.dita#am5xx_conref/cg4"/>
```

Example 4

This example shows how to produce the transpose of the result of the computation performed in Example 3, $\boldsymbol{C} \leftarrow \boldsymbol{A}+\boldsymbol{B}^T$, which uses the calling sequence:

```
      CALL SGEADD( A , 5 , 'N' , B , 3 , 'T' , C , 4 , 4 , 3 )
```

You instead code a calling sequence for $\boldsymbol{C}^T \leftarrow \boldsymbol{A}^T+\boldsymbol{B}$, as shown below, where the resulting matrix $\boldsymbol{C}^T$ in the output array CT is the transpose of the matrix in the output array C in Example 3. Note that the array CT has dimensions large enough to receive the transposed matrix. For a description of all the matrix identities, see “Special Usage” on page 388.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C   LDC  M   N
             |   |    |     |   |    |     |    |   |   |
CALL SGEADD( A , 5 , 'T'  , B , 3 , 'N'  , CT , 4 , 3 , 4 )
```

```
          conref="../transforms/am5xx_conref.dita#am5xx_conref/cg2"/>
         | 110000.0  120000.0  130000.0 |
         | 210000.0  220000.0  230000.0 |
A    =   | 310000.0  320000.0  330000.0 |
         | 410000.0  420000.0  430000.0 |
         |        .         .         .  |
          conref="../transforms/am5xx_conref.dita#am5xx_conref/cg4"/>
```

```
         | 11.0  12.0  13.0  14.0 |
B    =   | 21.0  22.0  23.0  24.0 |
         | 31.0  32.0  33.0  34.0 |
```

Output:

```
         | 110011.0  210012.0  310013.0  410014.0 |
CT   =   | 120021.0  220022.0  320023.0  420024.0 |
         | 130031.0  230032.0  330033.0  430034.0 |
         |        .         .         .         .  |
```

Example 5

This example shows the computation ***C***←***A***^{T}+***B***^{T}, where ***A, B,*** and ***C*** are the same size as the arrays A, B, and C, in which they are contained.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
             |   |    |     |   |    |     |   |   |   |
CALL SGEADD( A , 3 , 'T'  , B , 3 , 'T'  , C , 4 , 4 , 3 )
```

```
         | 110000.0  120000.0  130000.0  140000.0 |
A    =   | 210000.0  220000.0  230000.0  240000.0 |
         | 310000.0  320000.0  330000.0  340000.0 |
```

```
         | 11.0  12.0  13.0  14.0 |
B    =   | 21.0  22.0  23.0  24.0 |
         | 31.0  32.0  33.0  34.0 |
```

Output:

```
          conref="../transforms/am5xx_conref.dita#am5xx_conref/cg2"/>
         | 110011.0  210021.0  310031.0 |
C    =   | 120012.0  220022.0  320032.0 |
         | 130013.0  230023.0  330033.0 |
         | 140014.0  240024.0  340034.0 |
          conref="../transforms/am5xx_conref.dita#am5xx_conref/cg4"/>
```

Example 6

This example shows the computation ***C***←***A***+***B***, where ***A***, ***B***, and ***C*** are contained in larger arrays A, B, and C, respectively, and the arrays contain complex data.

Call Statement and Input:

```
              A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
              |   |    |     |   |    |     |   |   |   |
CALL CGEADD( A , 6 , 'N'  , B , 5 , 'N'  , C , 5 , 4 , 3 )
```

```
         | (1.0, 5.0)  (9.0, 2.0)  (1.0, 9.0) |
         | (2.0, 4.0)  (8.0, 3.0)  (1.0, 8.0) |
A    =   | (3.0, 3.0)  (7.0, 5.0)  (1.0, 7.0) |
         | (6.0, 6.0)  (3.0, 6.0)  (1.0, 4.0) |
         |     .           .           .      |
         |     .           .           .      |
```

```
         | (1.0, 8.0)  (2.0, 7.0)  (3.0, 2.0) |
         | (4.0, 4.0)  (6.0, 8.0)  (6.0, 3.0) |
B    =   | (6.0, 2.0)  (4.0, 5.0)  (4.0, 5.0) |
         | (7.0, 2.0)  (6.0, 4.0)  (1.0, 6.0) |
         |     .           .           .      |
```

Output:

```
         |  (2.0, 13.0)  (11.0,  9.0)  (4.0, 11.0) |
         |  (6.0,  8.0)  (14.0, 11.0)  (7.0, 11.0) |
C    =   |  (9.0,  5.0)  (11.0, 10.0)  (5.0, 12.0) |
         | (13.0,  8.0)   (9.0, 10.0)  (2.0, 10.0) |
         |      .             .             .      |
```

SGESUB, DGESUB, CGESUB, and ZGESUB (Matrix Subtraction for General Matrices or Their Transposes)

Purpose

These subroutines can perform any one of the following matrix subtractions, using matrices ***A*** and ***B*** or their transposes, and matrix ***C***:

$\boldsymbol{C} \leftarrow \boldsymbol{A} - \boldsymbol{B}$
$\boldsymbol{C} \leftarrow \boldsymbol{A}^T - \boldsymbol{B}$
$\boldsymbol{C} \leftarrow \boldsymbol{A} - \boldsymbol{B}^T$
$\boldsymbol{C} \leftarrow \boldsymbol{A}^T - \boldsymbol{B}^T$

Table 104: Data Types.

Data Types	
A, ***B***, ***C***	**Subroutine**
Short-precision real	SGESUB
Long-precision real	DGESUB
Short-precision complex	CGESUB
Long-precision complex	ZGESUB

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SGESUB | DGESUB | CGESUB | ZGESUB (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *m*, *n*)
C and C++	sgesub | dgesub | cgesub | zgesub (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *m*, *n*);

On Entry

a

is the matrix ***A***, where:

If *transa* = 'N', ***A*** is used in the computation, and ***A*** has *m* rows and *n* columns.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation, and ***A*** has *n* rows and *m* columns.

Note: No data should be moved to form $\boldsymbol{A}^T$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 104 on page 392, where:

If *transa* = 'N', its size must be *lda* by (at least) *n*.

If *transa* = 'T', its size must be *lda* by (at least) *m*.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and:

If *transa* = 'N', *lda* ≥ *m*.

If *transa* = 'T', *lda* ≥ *n*.

transa

indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation.

Specified as: a single character; *transa* = 'N' or 'T'.

b

is the matrix ***B***, where:

If *transb* = 'N', ***B*** is used in the computation, and ***B*** has *m* rows and *n* columns.

If *transb* = 'T', $\boldsymbol{B}^T$ is used in the computation, and ***B*** has *n* rows and *m* columns.

Note: No data should be moved to form $\boldsymbol{B}^T$; that is, the matrix ***B*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 103 on page 386, where:

If *transb* = 'N', its size must be *ldb* by (at least) *n*.

If *transb* = 'T', its size must be *ldb* by (at least) *m*.

ldb

is the leading dimension of the array specified for *b*.

Specified as: an integer; *ldb* > 0 and:

If *transb* = 'N', *ldb* ≥ *m*.

If *transb* = 'T', *ldb* ≥ *n*.

transb

indicates the form of matrix ***B*** to use in the computation, where:

If *transb* = 'N', ***B*** is used in the computation.

If *transb* = 'T', $\boldsymbol{B}^T$ is used in the computation.

Specified as: a single character; *transb* = 'N' or 'T'.

c

See On Return.

ldc

is the leading dimension of the array specified for *c*.

Specified as: an integer; $ldc > 0$ and $ldc \geq m$.

m

is the number of rows in matrix ***C***.

Specified as: an integer; $0 \leq m \leq ldc$.

n

is the number of columns in matrix ***C***.

Specified as: an integer; $0 \leq n$.

On Return

c

is the *m* by *n* matrix ***C***, containing the results of the computation. Returned as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 104 on page 392.

Notes

1. All subroutines accept lowercase letters for the *transa* and *transb* arguments.
2. Matrix ***C*** must have no common elements with matrices ***A*** or ***B***. However, ***C*** may (exactly) coincide with ***A*** if *transa* = 'N', and ***C*** may (exactly) coincide with ***B*** if *transb* = 'N'. Otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

The matrix subtraction is expressed as follows, where a_{ij}, b_{ij}, and c_{ij} are elements of matrices ***A, B,*** and ***C***, respectively:

$c_{ij} = a_{ij}-b_{ij}$ for $\boldsymbol{C}\leftarrow\boldsymbol{A}-\boldsymbol{B}$
$c_{ij} = a_{ij}-b_{ji}$ for $\boldsymbol{C}\leftarrow\boldsymbol{A}-\boldsymbol{B}^T$
$c_{ij} = a_{ji}-b_{ij}$ for $\boldsymbol{C}\leftarrow\boldsymbol{A}^T-\boldsymbol{B}$
$c_{ij} = a_{ji}-b_{ji}$ for $\boldsymbol{C}\leftarrow\boldsymbol{A}^T-\boldsymbol{B}^T$
for $i = 1, m$ and $j = 1, n$

If *m* or *n* is 0, no computation is performed.

Special Usage

You can compute the transpose $\boldsymbol{C}^T$ of each of the four computations listed under "Function" on page 394 by using the following matrix identities:

$(\boldsymbol{A}-\boldsymbol{B})^T = \boldsymbol{A}^T-\boldsymbol{B}^T$
$(\boldsymbol{A}-\boldsymbol{B}^T)^T = \boldsymbol{A}^T-\boldsymbol{B}$

$(\boldsymbol{A}^T\text{-}\boldsymbol{B})^T = \boldsymbol{A}\text{-}\boldsymbol{B}^T$
$(\boldsymbol{A}^T\text{-}\boldsymbol{B}^T)^T = \boldsymbol{A}\text{-}\boldsymbol{B}$

Be careful that your output array receiving $\boldsymbol{C}^T$ has dimensions large enough to hold the transposed matrix. See Example 5.

Error conditions

Computational Errors

None

Input-Argument Errors

1. *lda*, *ldb*, *ldc* ≤ 0
2. *m*, *n* < 0
3. *m* > *ldc*
4. *transa*, *transb* ≠ 'N' or 'T'
5. *transa* = 'N' and *m* > *lda*
6. *transa* = 'T' and *n* > *lda*
7. *transb* = 'N' and *m* > *ldb*
8. *transb* = 'T' and *n* > *ldb*

Examples

Example 1

This example shows the computation ***C***←***A***-***B***, where ***A*** and ***C*** are contained in larger arrays A and C, respectively, and ***B*** is the same size as array B, in which it is contained.

Call Statement and Input:

```
              A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
              |   |    |     |   |    |     |   |   |   |
CALL SGESUB( A , 6 , 'N'  , B , 4 , 'N'  , C , 5 , 4 , 3 )
```

```
         ┌                              ┐
         | 110000.0  120000.0  130000.0 |
         | 210000.0  220000.0  230000.0 |
A    =   | 310000.0  320000.0  330000.0 |
         | 410000.0  420000.0  430000.0 |
         |        .         .         . |
         |        .         .         . |
                                        ┘
```

```
         ┌                     ┐
         | -11.0  -12.0  -13.0 |
B    =   | -21.0  -22.0  -23.0 |
         | -31.0  -32.0  -33.0 |
         | -41.0  -42.0  -43.0 |
                               ┘
```

Output:

```
         ┌                              ┐
         | 110011.0  120012.0  130013.0 |
         | 210021.0  220022.0  230023.0 |
C    =   | 310031.0  320032.0  330033.0 |
         | 410041.0  420042.0  430043.0 |
         |        .         .         . |
                                        ┘
```

Example 2

This example shows the computation ***C***←***A***^{T}-***B***, where ***A, B,*** and ***C*** are the same size as arrays A, B, and C, in which they are contained.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
             |   |    |     |   |    |     |   |   |   |
CALL SGESUB( A , 3 , 'T'  , B , 4 , 'N'  , C , 4 , 4 , 3 )
```

```
         ┌                                        ┐
         | 110000.0  120000.0  130000.0  140000.0 |
A    =   | 210000.0  220000.0  230000.0  240000.0 |
         | 310000.0  320000.0  330000.0  340000.0 |
         └                                        ┘
```

```
         ┌                    ┐
         | -11.0  -12.0  -13.0 |
B    =   | -21.0  -22.0  -23.0 |
         | -31.0  -32.0  -33.0 |
         | -41.0  -42.0  -43.0 |
         └                    ┘
```

Output:

```
         ┌                              ┐
         | 110011.0  210012.0  310013.0 |
C    =   | 120021.0  220022.0  320023.0 |
         | 130031.0  230032.0  330033.0 |
         | 140041.0  240042.0  340043.0 |
         └                              ┘
```

Example 3

This example shows computation ***C***←***A***-***B***^{T}, where ***A*** is contained in a larger array A, and ***B*** and ***C*** are the same size as arrays B and C, in which they are contained.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
             |   |    |     |   |    |     |   |   |   |
CALL SGESUB( A , 5 , 'N'  , B , 3 , 'T'  , C , 4 , 4 , 3 )
```

```
         ┌                              ┐
         | 110000.0  120000.0  130000.0 |
         | 210000.0  220000.0  230000.0 |
A    =   | 310000.0  320000.0  330000.0 |
         | 410000.0  420000.0  430000.0 |
         |       .         .         .  |
         └                              ┘
```

```
         ┌                           ┐
         | -11.0  -12.0  -13.0  -14.0 |
B    =   | -21.0  -22.0  -23.0  -24.0 |
         | -31.0  -32.0  -33.0  -34.0 |
         └                           ┘
```

Output:

```
         ┌                              ┐
         | 110011.0  120021.0  130031.0 |
C    =   | 210012.0  220022.0  230032.0 |
         | 310013.0  320023.0  330033.0 |
         | 410014.0  420024.0  430034.0 |
         └                              ┘
```

Example 4

This example shows the computation ***C***←***A***^{T}-***B***^{T}, where ***A, B,*** and ***C*** are the same size as the arrays A, B, and C, in which they are contained.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
             |   |    |     |   |    |     |   |   |   |
CALL SGESUB( A , 3 , 'T'  , B , 3 , 'T'  , C , 4 , 4 , 3 )
```

```
        ┌                                         ┐
        | 110000.0  120000.0  130000.0  140000.0 |
A    =  | 210000.0  220000.0  230000.0  240000.0 |
        | 310000.0  320000.0  330000.0  340000.0 |
                                                 ┘
```

```
        ┌                           ┐
        | -11.0  -12.0  -13.0  -14.0 |
B    =  | -21.0  -22.0  -23.0  -24.0 |
        | -31.0  -32.0  -33.0  -34.0 |
                                     ┘
```

Output:

```
        ┌                               ┐
        | 110011.0  210021.0  310031.0 |
C    =  | 120012.0  220022.0  320032.0 |
        | 130013.0  230023.0  330033.0 |
        | 140014.0  240024.0  340034.0 |
                                       ┘
```

Example 5

This example shows how to produce the transpose of the result of the computation performed in Example 4, $\boldsymbol{C}\leftarrow\boldsymbol{A}^T\text{-}\boldsymbol{B}^T$, which uses the calling sequence:

```
      CALL SGESUB( A , 3 , 'T' , B , 3 , 'T' , C , 4 , 4 , 3 )
```

You instead code a calling sequence for $\boldsymbol{C}^T\leftarrow\boldsymbol{A}\text{-}\boldsymbol{B}$, as shown below, where the resulting matrix $\boldsymbol{C}^T$ in the output array CT is the transpose of the matrix in the output array C in Example 4. Note that the array CT has dimensions large enough to receive the transposed matrix. For a description of all the matrix identities, see "Special Usage" on page 394.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C    LDC  M   N
             |   |    |     |   |    |     |    |   |   |
CALL SGESUB( A , 3 , 'N'  , B , 3 , 'N'  , CT , 3 , 3 , 4 )
```

```
        ┌                                         ┐
        | 110000.0  120000.0  130000.0  140000.0 |
A    =  | 210000.0  220000.0  230000.0  240000.0 |
        | 310000.0  320000.0  330000.0  340000.0 |
                                                 ┘
```

```
        ┌                           ┐
        | -11.0  -12.0  -13.0  -14.0 |
B    =  | -21.0  -22.0  -23.0  -24.0 |
        | -31.0  -32.0  -33.0  -34.0 |
                                     ┘
```

Output:

```
        ┌                                         ┐
        | 110011.0  120012.0  130013.0  140014.0 |
CT   =  | 210021.0  220022.0  230023.0  240024.0 |
        | 310031.0  320032.0  330033.0  340034.0 |
                                                 ┘
```

Example 6

This example shows the computation $\boldsymbol{C}\leftarrow\boldsymbol{A}\text{-}\boldsymbol{B}$, where $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are contained in larger arrays A, B, and C, respectively, and the arrays contain complex data.

Call Statement and Input:

```
                A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
                |   |    |     |   |    |     |   |   |   |
CALL CGESUB( A , 6 , 'N'  , B , 5 , 'N'  , C , 5 , 4 , 3 )
```

```
          ┌                                       ┐
          | (1.0, 5.0)  (9.0, 2.0)  (1.0, 9.0) |
          | (2.0, 4.0)  (8.0, 3.0)  (1.0, 8.0) |
A    =    | (3.0, 3.0)  (7.0, 5.0)  (1.0, 7.0) |
          | (6.0, 6.0)  (3.0, 6.0)  (1.0, 4.0) |
          |     .           .           .      |
          |     .           .           .      |
          └                                       ┘
```

```
          ┌                                       ┐
          | (1.0, 8.0)  (2.0, 7.0)  (3.0, 2.0) |
          | (4.0, 4.0)  (6.0, 8.0)  (6.0, 3.0) |
B    =    | (6.0, 2.0)  (4.0, 5.0)  (4.0, 5.0) |
          | (7.0, 2.0)  (6.0, 4.0)  (1.0, 6.0) |
          |     .           .           .      |
          └                                       ┘
```

Output:

```
          ┌                                               ┐
          |  (0.0, -3.0)   (7.0, -5.0)  (-2.0,  7.0) |
          | (-2.0,  0.0)   (2.0, -5.0)  (-5.0,  5.0) |
C    =    | (-3.0,  1.0)   (3.0,  0.0)  (-3.0,  2.0) |
          | (-1.0,  4.0)  (-3.0,  2.0)   (0.0, -2.0) |
          |      .             .             .       |
          └                                               ┘
```

SGEMUL, DGEMUL, CGEMUL, and ZGEMUL (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes)

Purpose

SGEMUL and DGEMUL can perform any one of the following matrix multiplications, using matrices ***A*** and ***B*** or their transposes, and matrix ***C***:

SGEMUL and DGEMUL Combined Matrix Computations

$\boldsymbol{C} \leftarrow \boldsymbol{AB}$	$\boldsymbol{C} \leftarrow \boldsymbol{AB}^T$
$\boldsymbol{C} \leftarrow \boldsymbol{A}^T\boldsymbol{B}$	$\boldsymbol{C} \leftarrow \boldsymbol{A}^T\boldsymbol{B}^T$

CGEMUL and ZGEMUL can perform any one of the following matrix multiplications, using matrices ***A*** and ***B***, their transposes or their conjugate transposes, and matrix ***C***:

CGEMUL and ZGEMUL Combined Matrix Computations

$\boldsymbol{C} \leftarrow \boldsymbol{AB}$	$\boldsymbol{C} \leftarrow \boldsymbol{AB}^T$	$\boldsymbol{C} \leftarrow \boldsymbol{AB}^H$
$\boldsymbol{C} \leftarrow \boldsymbol{A}^T\boldsymbol{B}$	$\boldsymbol{C} \leftarrow \boldsymbol{A}^T\boldsymbol{B}^T$	$\boldsymbol{C} \leftarrow \boldsymbol{A}^T\boldsymbol{B}^H$
$\boldsymbol{C} \leftarrow \boldsymbol{A}^H\boldsymbol{B}$	$\boldsymbol{C} \leftarrow \boldsymbol{A}^H\boldsymbol{B}^T$	$\boldsymbol{C} \leftarrow \boldsymbol{A}^H\boldsymbol{B}^H$

Table 105: Data Types.

Data Types	
A, ***B***, ***C***	**Subroutine**
Short-precision real	SGEMUL

Table 105: Data Types. (continued)	
Data Types	
Long-precision real	DGEMUL
Short-precision complex	CGEMUL
Long-precision complex	ZGEMUL

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	**Syntax**
Fortran	CALL SGEMUL | DGEMUL | CGEMUL | ZGEMUL (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *l*, *m*, *n*)
C and C++	sgemul | dgemul | cgemul | zgemul (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *l*, *m*, *n*);

On Entry

a

is the matrix ***A***, where:

If *transa* = 'N', ***A*** is used in the computation, and ***A*** has *l* rows and *m* columns.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation, and ***A*** has *m* rows and *l* columns.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation, and ***A*** has *m* rows and *l* columns.

Note: No data should be moved to form $\boldsymbol{A}^T$ or $\boldsymbol{A}^H$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 105 on page 398, where:

If *transa* = 'N', its size must be *lda* by (at least) *m*.

If *transa* = 'T' or 'C', its size must be *lda* by (at least) *l*.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and:

If *transa* = 'N', *lda* ≥ *l*.

If *transa* = 'T' or 'C', *lda* ≥ *m*.

transa

indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation.

Specified as: a single character; *transa* = 'N' or 'T' for SGEMUL and DGEMUL; *transa* = 'N', 'T', or 'C' for CGEMUL and ZGEMUL.

b

is the matrix ***B***, where:

If *transb* = 'N', ***B*** is used in the computation, and ***B*** has *m* rows and *n* columns.

If *transb* = 'T', $\boldsymbol{B}^T$ is used in the computation, and ***B*** has *n* rows and *m* columns.

If *transb* = 'C', $\boldsymbol{B}^H$ is used in the computation, and ***B*** has *n* rows and *m* columns.

Note: No data should be moved to form $\boldsymbol{B}^T$ or $\boldsymbol{B}^H$; that is, the matrix ***B*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 105 on page 398, where:

If *transb* = 'N', its size must be *ldb* by (at least) *n*.

If *transb* = 'T' or 'C', its size must be *ldb* by (at least) *m*.

ldb

is the leading dimension of the array specified for *b*.

Specified as: an integer; *ldb* > 0 and:

If *transb* = 'N', *ldb* ≥ *m*.

If *transb* = 'T' or 'C', *ldb* ≥ *n*.

transb

indicates the form of matrix ***B*** to use in the computation, where:

If *transb* = 'N', ***B*** is used in the computation.

If *transb* = 'T', $\boldsymbol{B}^T$ is used in the computation.

If *transb* = 'C', $\boldsymbol{B}^H$ is used in the computation.

Specified as: a single character; *transb* = 'N' or 'T' for SGEMUL and DGEMUL; *transb* = 'N', 'T', or 'C' for CGEMUL and ZGEMUL.

c

See On Return.

ldc

is the leading dimension of the array specified for *c*.

Specified as: an integer; *ldc* > 0 and *ldc* ≥ *l*.

l

is the number of rows in matrix ***C***.

Specified as: an integer; 0 ≤ *l* ≤ *ldc*.

m

has the following meaning, where:

If *transa* = 'N', it is the number of columns in matrix ***A***.

If *transa* = 'T' or 'C', it is the number of rows in matrix ***A***.

In addition:

If *transb* = 'N', it is the number of rows in matrix ***B***.

If *transb* = 'T' or 'C', it is the number of columns in matrix ***B***.

Specified as: an integer; *m* ≥ 0.

n

is the number of columns in matrix ***C***.

Specified as: an integer; *n* ≥ 0.

On Return

c

is the *l* by *n* matrix ***C***, containing the results of the computation. Returned as: an *ldc* by (at least) *n* numbers of the data type indicated in Table 105 on page 398.

Notes

1. All subroutines accept lowercase letters for the *transa* and *transb* arguments.
2. Matrix ***C*** must have no common elements with matrices ***A*** or ***B***; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

The matrix multiplication is expressed as follows, where a_{ik}, b_{kj}, and c_{ij} are elements of matrices ***A, B,*** and ***C***, respectively:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}\boldsymbol{B}$$

$$c_{ij} = \sum_{k=1}^{m} a_{ki} b_{kj} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}$$

$$c_{ij} = \sum_{k=1}^{m} \bar{a}_{ki} b_{kj} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{H}}\boldsymbol{B}$$

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{jk} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}\boldsymbol{B}^{\mathrm{T}}$$

$$c_{ij} = \sum_{k=1}^{m} a_{ki} b_{jk} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}^{\mathrm{T}}$$

$$c_{ij} = \sum_{k=1}^{m} \bar{a}_{ki} b_{jk} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{H}}\boldsymbol{B}^{\mathrm{T}}$$

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \bar{b}_{jk} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}\boldsymbol{B}^{\mathrm{H}}$$

$$c_{ij} = \sum_{k=1}^{m} a_{ki} \bar{b}_{jk} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}^{\mathrm{H}}$$

$$c_{ij} = \sum_{k=1}^{m} \bar{a}_{ki} \bar{b}_{jk} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{H}}\boldsymbol{B}^{\mathrm{H}}$$

$$\text{for } \quad i = 1,\ l \ \text{ and } \ j = 1, n$$

See reference [“48” on page 1255]. If *l* or *n* is 0, no computation is performed. If *l* and *n* are greater than 0, and *m* is 0, an *l* by *n* matrix of zeros is returned.

Special Usage

Equivalence Rules

By using the following equivalence rules, you can compute the transpose $\boldsymbol{C}^{\mathrm{T}}$ or the conjugate transpose $\boldsymbol{C}^{\mathrm{H}}$ of some of the computations performed by these subroutines:

Transpose	**Conjugate Transpose**
$(\boldsymbol{AB})^T = \boldsymbol{B}^T\boldsymbol{A}^T$	$(\boldsymbol{AB})^H = \boldsymbol{B}^H\boldsymbol{A}^H$
$(\boldsymbol{A}^T\boldsymbol{B})^T = \boldsymbol{B}^T\boldsymbol{A}$	$(\boldsymbol{A}^H\boldsymbol{B})^H = \boldsymbol{B}^H\boldsymbol{A}$
$(\boldsymbol{AB}^T)^T = \boldsymbol{BA}^T$	$(\boldsymbol{AB}^H)^H = \boldsymbol{BA}^H$
$(\boldsymbol{A}^T\boldsymbol{B}^T)^T = \boldsymbol{BA}$	$(\boldsymbol{A}^H\boldsymbol{B}^H)^H = \boldsymbol{BA}$

When coding the calling sequences for these cases, be careful to code your matrix arguments and dimension arguments in the order indicated by the rule. Also, be careful that your output array, receiving $\boldsymbol{C}^T$ or $\boldsymbol{C}^H$, has dimensions large enough to hold the resulting transposed or conjugate transposed matrix. See Example 2 and Example 4.

Error conditions

Resource Errors

Unable to allocate internal work area.

Computational Errors

None

Input-Argument Errors

1. *lda*, *ldb*, *ldc* ≤ 0
2. *l*, *m*, *n* < 0
3. *l* > *ldc*
4. *transa*, *transb* ≠ 'N' or 'T' for SGEMUL and DGEMUL
5. *transa*, *transb* ≠ 'N', 'T', or 'C' for CGEMUL and ZGEMUL
6. *transa* = 'N' and *l* > *lda*
7. *transa* = 'T' or 'C' and *m* > *lda*
8. *transb* = 'N' and *m* > *ldb*
9. *transb* = 'T' or 'C' and *n* > *ldb*

Examples

Example 1

This example shows the computation $\boldsymbol{C} \leftarrow \boldsymbol{AB}$, where ***A, B,*** and ***C*** are contained in larger arrays A, B, and C, respectively.

Call Statement and Input:

```
              A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N
              |   |    |     |   |    |     |   |   |   |   |
CALL SGEMUL( A , 8 , 'N'  , B , 6 , 'N'  , C , 7 , 6 , 5 , 4 )
```

```
         |  1.0   2.0  -1.0  -1.0   4.0 |
         |  2.0   0.0   1.0   1.0  -1.0 |
         |  1.0  -1.0  -1.0   1.0   2.0 |
A    =   | -3.0   2.0   2.0   2.0   0.0 |
         |  4.0   0.0  -2.0   1.0  -1.0 |
         | -1.0  -1.0   1.0  -3.0   2.0 |
         |   .     .     .     .     .  |
         |   .     .     .     .     .  |
```

```
         |  1.0  -1.0   0.0   2.0 |
         |  2.0   2.0  -1.0  -2.0 |
B    =   |  1.0   0.0  -1.0   1.0 |
         | -3.0  -1.0   1.0  -1.0 |
         |  4.0   2.0  -1.0   1.0 |
        |   .     .     .     .  |
```

Output:

```
        | 23.0  12.0  -6.0    2.0 |
        | -4.0  -5.0   1.0    3.0 |
        |  3.0   0.0   1.0    4.0 |
C    =  | -3.0   5.0  -2.0  -10.0 |
        | -5.0  -7.0   4.0    4.0 |
        | 15.0   6.0  -5.0    6.0 |
        |   .     .     .      .  |
```

Example 2

This example shows how to produce the transpose of the result of the computation performed in Example 1, ***C***←***AB***, which uses the calling sequence:

```
       CALL  SGEMUL (A,8,'N',B,6,'N',C,7,6,5,4)
```

You instead code a calling sequence for $\boldsymbol{C}^T \leftarrow \boldsymbol{B}^T\boldsymbol{A}^T$, as shown below, where the resulting matrix $\boldsymbol{C}^T$ in the output array CT is the transpose of the matrix in the output array C in Example 1. Note that the array CT has dimensions large enough to receive the transposed matrix. For a description of all the matrix identities, see "Special Usage" on page 402.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C   LDC  L   M   N
             |   |    |     |   |    |     |    |   |   |   |
CALL SGEMUL( B , 6 , 'T'  , A , 8 , 'T'  , CT , 5 , 4 , 5 , 6 )
```

```
        |  1.0  -1.0   0.0   2.0 |
        |  2.0   2.0  -1.0  -2.0 |
B    =  |  1.0   0.0  -1.0   1.0 |
        | -3.0  -1.0   1.0  -1.0 |
        |  4.0   2.0  -1.0   1.0 |
        |   .     .     .     .  |
```

```
        |  1.0   2.0  -1.0  -1.0   4.0 |
        |  2.0   0.0   1.0   1.0  -1.0 |
        |  1.0  -1.0  -1.0   1.0   2.0 |
A    =  | -3.0   2.0   2.0   2.0   0.0 |
        |  4.0   0.0  -2.0   1.0  -1.0 |
        | -1.0  -1.0   1.0  -3.0   2.0 |
        |   .     .     .     .     .  |
        |   .     .     .     .     .  |
```

Output:

```
        | 23.0  -4.0  3.0   -3.0  -5.0  15.0 |
        | 12.0  -5.0  0.0    5.0  -7.0   6.0 |
CT   =  | -6.0   1.0  1.0   -2.0   4.0  -5.0 |
        |  2.0   3.0  4.0  -10.0   4.0   6.0 |
        |   .     .    .      .     .     .  |
```

Example 3

This example shows the computation $\boldsymbol{C} \leftarrow \boldsymbol{A}^T\boldsymbol{B}$, where ***A*** and ***C*** are contained in larger arrays A and C, respectively, and ***B*** is the same size as the

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N
             |   |    |     |   |    |     |   |   |   |   |
CALL SGEMUL( A , 4 , 'T'  , B , 3 , 'N'  , C , 5 , 3 , 3 , 6 )
```

```
A    =  | 1.0  -3.0   2.0 |
        | 2.0   4.0   0.0 |
        | 1.0  -1.0  -1.0 |
        |  .     .     .  |
```

```
        | 1.0  -3.0   2.0   2.0  -1.0   2.0 |
B    =  | 2.0   4.0   0.0   0.0   1.0  -2.0 |
        | 1.0  -1.0  -1.0  -1.0  -1.0   1.0 |
```

Output:

```
        | 6.0   4.0   1.0   1.0   0.0   -1.0 |
        | 4.0  26.0  -5.0  -5.0   8.0  -15.0 |
C    =  | 1.0  -5.0   5.0   5.0  -1.0    3.0 |
        |  .     .     .     .     .      .  |
        |  .     .     .     .     .      .  |
```

Example 4

This example shows how to produce the transpose of the result of the computation performed in Example 3, $\boldsymbol{C} \leftarrow \boldsymbol{A}^T\boldsymbol{B}$, which uses the calling sequence:

```
       CALL  SGEMUL (A,4,'T',B,3,'N',C,5,3,3,6)
```

You instead code the calling sequence for $\boldsymbol{C}^T \leftarrow \boldsymbol{B}^T\boldsymbol{A}$, as shown below, where the resulting matrix $\boldsymbol{C}^T$ in the output array CT is the transpose of the matrix in the output array C in Example 3. Note that the array CT has dimensions large enough to receive the transposed matrix. For a description of all the matrix identities, see "Special Usage" on page 402.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C   LDC  L   M   N
             |   |    |     |   |    |     |    |   |   |   |
CALL SGEMUL( B , 3 , 'T'  , A , 4 , 'N'  , CT , 8 , 6 , 3 , 3 )
```

```
        | 1.0  -3.0   2.0   2.0  -1.0   2.0 |
B    =  | 2.0   4.0   0.0   0.0   1.0  -2.0 |
        | 1.0  -1.0  -1.0  -1.0  -1.0   1.0 |
```

```
        | 1.0  -3.0   2.0 |
A    =  | 2.0   4.0   0.0 |
        | 1.0  -1.0  -1.0 |
        |  .     .     .  |
```

Output:

```
        |  6.0    4.0   1.0 |
        |  4.0   26.0  -5.0 |
        |  1.0   -5.0   5.0 |
CT   =  |  1.0   -5.0   5.0 |
        |  0.0    8.0  -1.0 |
        | -1.0  -15.0   3.0 |
        |   .      .     .  |
        |    .       .      .  |
```

Example 5

This example shows the computation ***C***←***AB***^{T}, where ***A*** and ***C*** are contained in larger arrays A and C, respectively, and ***B*** is the same size as the array B in which it is contained.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N
             |   |    |     |   |    |     |   |   |   |   |
CALL SGEMUL( A , 4 , 'N'  , B , 3 , 'T'  , C , 5 , 3 , 2 , 3 )
```

```
        | 1.0  -3.0 |
A    =  | 2.0   4.0 |
        | 1.0  -1.0 |
        |  .     .  |
```

```
        | 1.0  -3.0 |
B    =  | 2.0   4.0 |
        | 1.0  -1.0 |
```

Output:

```
        |  10.0  -10.0   4.0 |
        | -10.0   20.0  -2.0 |
C    =  |   4.0   -2.0   2.0 |
        |    .      .     .  |
        |    .      .     .  |
```

Example 6

This example shows the computation ***C***←***A***^{T}***B***^{T}, where ***A, B,*** and ***C*** are the same size as the arrays A, B, and C in which they are contained. (Based on the dimensions of the matrices, ***A*** is actually a column vector, and ***C*** is actually a row vector.)

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N
             |   |    |     |   |    |     |   |   |   |   |
CALL SGEMUL( A , 3 , 'T'  , B , 3 , 'T'  , C , 1 , 1 , 3 , 3 )
```

```
        | 1.0 |
A    =  | 2.0 |
        | 1.0 |
```

```
        | 1.0  -3.0   2.0 |
B    =  | 2.0   4.0   0.0 |
        | 1.0  -1.0  -1.0 |
```

Output:

```
B    =  | -3.0  10.0  -2.0 |
```

Example 7

This example shows the computation ***C***←***A***^{T}***B*** using complex data, where ***A***, ***B***, and ***C*** are contained in larger arrays A, B, and C, respectively.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N
             |   |    |     |   |    |     |   |   |   |   |
CALL CGEMUL( A , 6 , 'T'  , B , 7 , 'N'  , C , 3 , 2 , 3 , 3 )
```

```
        | (1.0, 2.0)  (3.0, 4.0) |
        | (4.0, 6.0)  (7.0, 1.0) |
A    =  | (6.0, 3.0)  (2.0, 5.0) |
        |     .           .      |
        |     .           .      |
        |     .           .      |
```

```
        | (1.0, 9.0)  (2.0, 6.0)  (5.0, 6.0) |
        | (2.0, 5.0)  (6.0, 2.0)  (6.0, 4.0) |
        | (2.0, 6.0)  (5.0, 4.0)  (2.0, 6.0) |
B    =  |     .           .           .      |
        |     .           .           .      |
        |     .           .           .      |
        |     .           .           .      |
```

Output:

```
        | (-45.0, 85.0)  (20.0, 93.0)  (-13.0, 110.0) |
C    =  | (-50.0, 90.0)  (12.0, 79.0)    (3.0,  94.0) |
        |       .              .              .       |
```

Example 8

This example shows the computation ***C***←***AB***^{H} using complex data, where ***A*** and ***C*** are contained in larger arrays A and C, respectively, and ***B*** is the same size as the array B in which it is contained.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N
             |   |    |     |   |    |     |   |   |   |   |
CALL CGEMUL( A , 4 , 'N'  , B , 3 , 'C'  , C , 4 , 3 , 2 , 3 )
```

```
        | (1.0, 2.0)  (-3.0, 2.0) |
A    =  | (2.0, 6.0)   (4.0, 5.0) |
        | (1.0, 2.0)  (-1.0, 8.0) |
        |     .            .      |
```

```
        | (1.0, 3.0)  (-3.0, 2.0) |
B    =  | (2.0, 5.0)   (4.0, 6.0) |
        | (1.0, 1.0)  (-1.0, 9.0) |
```

Output:

```
        | (20.0,  -1.0)  (12.0, 25.0)  (24.0,  26.0) |
C    =  | (18.0, -23.0)  (80.0, -2.0)  (49.0, -37.0) |
        | (26.0, -23.0)  (56.0, 37.0)  (76.0,   2.0) |
        |       .              .             .       |
```

SGEMMS, DGEMMS, CGEMMS, and ZGEMMS (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm)

Purpose

These subroutines use Winograd's variation of the Strassen's algorithm to perform the matrix multiplication for both real and complex matrices. SGEMMS and DGEMMS can perform any one of the following matrix multiplications, using matrices ***A*** and ***B*** or their transposes, and matrix ***C***:

SGEMMS and DGEMMS Combined Matrix Computations

$\boldsymbol{C} \leftarrow \boldsymbol{AB}$	$\boldsymbol{C} \leftarrow \boldsymbol{AB}^{\mathrm{T}}$
$\boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}$	$\boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}^{\mathrm{T}}$

CGEMMS and ZGEMMS can perform any one of the following matrix multiplications, using matrices ***A*** and ***B***, their transposes or their conjugate transposes, and matrix ***C***:

CGEMMS and ZGEMMS Combined Matrix Computations

$\boldsymbol{C} \leftarrow \boldsymbol{AB}$	$\boldsymbol{C} \leftarrow \boldsymbol{AB}^{\mathrm{T}}$	$\boldsymbol{C} \leftarrow \boldsymbol{AB}^{\mathrm{H}}$
$\boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}$	$\boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}^{\mathrm{T}}$	$\boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}^{\mathrm{H}}$
$\boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{H}}\boldsymbol{B}$	$\boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{H}}\boldsymbol{B}^{\mathrm{T}}$	$\boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{H}}\boldsymbol{B}^{\mathrm{H}}$

Table 106: Data Types.

Data Types		
A, ***B***, ***C***	*aux*	**Subroutine**
Short-precision real	Short-precision real	SGEMMS
Long-precision real	Long-precision real	DGEMMS
Short-precision complex	Short-precision real	CGEMMS
Long-precision complex	Long-precision real	ZGEMMS

Syntax

Language	Syntax
Fortran	CALL SGEMMS | DGEMMS | CGEMMS | ZGEMMS (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *l*, *m*, *n*, *aux*, *naux*)
C and C++	sgemms | dgemms | cgemms | zgemms (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *l*, *m*, *n*, *aux*, *naux*);

On Entry

a

is the matrix ***A***, where:

If *transa* = 'N', ***A*** is used in the computation, and ***A*** has *l* rows and *m* columns.

If *transa* = 'T', $\boldsymbol{A}^{\mathrm{T}}$ is used in the computation, and ***A*** has *m* rows and *l* columns.

If *transa* = 'C', $\boldsymbol{A}^{\mathrm{H}}$ is used in the computation, and ***A*** has *m* rows and *l* columns.

Note: No data should be moved to form $\boldsymbol{A}^T$ or $\boldsymbol{A}^H$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 106 on page 408, where:

If *transa* = 'N', its size must be *lda* by (at least) *m*.

If *transa* = 'T' or 'C', its size must be *lda* by (at least) *l*.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and:

If *transa* = 'N', *lda* ≥ *l*.

If *transa* = 'T' or 'C', *lda* ≥ *m*.

transa
indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation.

Specified as: a single character; *transa* = 'N' or 'T' for SGEMMS and DGEMMS; *transa* = 'N', 'T', or 'C' for CGEMMS and ZGEMMS.

b
is the matrix ***B***, where:

If *transb* = 'N', ***B*** is used in the computation, and ***B*** has *m* rows and *n* columns.

If *transb* = 'T', $\boldsymbol{B}^T$ is used in the computation, and ***B*** has *n* rows and *m* columns.

If *transb* = 'C', $\boldsymbol{B}^H$ is used in the computation, and ***B*** has *n* rows and *m* columns.

Note: No data should be moved to form $\boldsymbol{B}^T$ or $\boldsymbol{B}^H$; that is, the matrix ***B*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 106 on page 408, where:

If *transb* = 'N', its size must be *ldb* by (at least) *n*.

If *transb* = 'T' or 'C', its size must be *ldb* by (at least) *m*.

ldb
is the leading dimension of the array specified for *b*.

Specified as: an integer; *ldb* > 0 and:

If *transb* = 'N', *ldb* ≥ *m*.

If *transb* = 'T' or 'C', *ldb* ≥ *n*.

transb
indicates the form of matrix ***B*** to use in the computation, where:

If *transb* = 'N', ***B*** is used in the computation.

If *transb* = 'T', $\boldsymbol{B}^T$ is used in the computation.

If *transb* = 'C', $\boldsymbol{B}^H$ is used in the computation.

Specified as: a single character; *transb* = 'N' or 'T' for SGEMMS and DGEMMS; *transb* = 'N', 'T', or 'C' for CGEMMS and ZGEMMS.

c
See On Return.

ldc

is the leading dimension of the array specified for *c*.

Specified as: an integer; $ldc > 0$ and $ldc \geq l$.

l

is the number of rows in matrix ***C***.

Specified as: an integer; $0 \leq l \leq ldc$.

m

has the following meaning, where:

If *transa* = 'N', it is the number of columns in matrix ***A***.

If *transa* = 'T' or 'C', it is the number of rows in matrix ***A***.

In addition:

If *transb* = 'N', it is the number of rows in matrix ***B***.

If *transb* = 'T' or 'C', it is the number of columns in matrix ***B***.

Specified as: an integer; $m \geq 0$.

n

is the number of columns in matrix ***C***.

Specified as: an integer; $n \geq 0$.

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage containing numbers of the data type indicated in Table 106 on page 408.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SGEMMS, DGEMMS, CGEMMS, and ZGEMMS dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise,

When this subroutine uses Strassen's algorithm:

- For SGEMMS and DGEMMS:

  Use $naux = \max[(n)(l), 0.7m(l+n)]$.

- For CGEMMS and ZGEMMS:

  Use $naux = \max[(n)(l), 0.7m(l+n)]+nb1+nb2$, where:
  If $l \geq n$, then $nb1 \geq (l)(n+20)$ and $nb2 \geq \max[(n)(l), (m)(n+20)]$.
  If $l < n$, then $nb1 \geq (m)(n+20)$ and $nb2 \geq \max[(n)(l), (l)(m+20)]$.

When this subroutine uses the direct method (_GEMUL), use $naux \geq 0$.

Note:

1. In most cases, these formulas provide an overestimate.
2. For an explanation of when this subroutine uses the direct method versus Strassen's algorithm, see “Notes” on page 411.

On Return

c
is the *l* by *n* matrix ***C***, containing the results of the computation. Returned as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 106 on page 408.

Notes

1. There are two instances when these subroutines use the direct method (_GEMUL), rather than using Strassen's algorithm:
   - When either or both of the input matrices are small
   - For CGEMMS and ZGEMMS, when input matrices ***A*** and ***B*** overlap

   In these instances when the direct method is used, the subroutine does not use auxiliary storage, and you can specify *naux* = 0.
2. For CGEMMS and ZGEMMS, one of the input matrices, ***A*** or ***B***, is rearranged during the computation and restored to its original form on return. Keep this in mind when diagnosing an abnormal termination.
3. All subroutines accept lowercase letters for the *transa* and *transb* arguments.
4. Matrix ***C*** must have no common elements with matrices ***A*** or ***B***; otherwise, results are unpredictable. See "Vector concepts" on page 71.
5. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

The matrix multiplications performed by these subroutines are functionally equivalent to those performed by SGEMUL, DGEMUL, CGEMUL, and ZGEMUL. For details on the computations performed, see "Function" on page 401.

SGEMMS, DGEMMS, CGEMMS, and ZGEMMS use Winograd's variation of the Strassen's algorithm with minor changes for tuning purposes. (See pages 45 and 46 in reference ["18" on page 1254].) The subroutines compute matrix multiplication for both real and complex matrices of large sizes. Complex matrix multiplication uses a special technique, using three real matrix multiplications and five real matrix additions. Each of these three resulting matrix multiplications then uses Strassen's algorithm.

Strassen's Algorithm
The steps of Strassen's algorithm can be repeated up to four times by these subroutines, with each step reducing the dimensions of the matrix by a factor of two. The number of steps used by this subroutine depends on the size of the input matrices. Each step reduces the number of operations by about 10% from the normal matrix multiplication. On the other hand, if the matrix is small, a normal matrix multiplication is performed without using the Strassen's algorithm, and no improvement is gained. For details about small matrices, see "Notes" on page 411.

Complex Matrix Multiplication
The complex multiplication is performed by forming the real and imaginary parts of the input matrices. These subroutines uses three real matrix multiplications and five real matrix additions, instead of the normal four real matrix multiplications and two real matrix additions. Using only three real matrix multiplications allows the subroutine to achieve up to a 25% reduction in matrix operations, which can result in a significant savings in computing time for large matrices.

Accuracy Considerations
Strassen's method is not stable for certain row or column scalings of the input matrices ***A*** and ***B***. Therefore, for matrices ***A*** and ***B*** with divergent exponent values Strassen's method may give inaccurate results. For these cases, you should use the _GEMUL or _GEMM subroutines.

Special Usage

The equivalence rules, defined for matrix multiplication of ***A*** and ***B*** in "Special Usage" on page 402, also apply to these subroutines. You should use the equivalence rules when you want to transpose or conjugate transpose the result of the multiplication computation. When coding the calling sequences for these cases, be careful to code your matrix arguments and dimension arguments in the order indicated by the rule. Also, be careful that your output array, receiving $\boldsymbol{C}^T$ or $\boldsymbol{C}^H$, has dimensions large enough to hold the resulting transposed or conjugate transposed matrix. See Example 2 and Example 4.

Error conditions

Resource Errors

Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors

None

Input-Argument Errors

1. *lda*, *ldb*, *ldc* ≤ 0
2. *l*, *m*, *n* < 0
3. *l* > *ldc*
4. *transa*, *transb* ≠ 'N' or 'T' for SGEMMS and DGEMMS
5. *transa*, *transb* ≠ 'N', 'T', or 'C' for CGEMMS and ZGEMMS
6. *transa* = 'N' and *l* > *lda*
7. *transa* = 'T' or 'C' and *m* > *lda*
8. *transb* = 'N' and *m* > *ldb*
9. *transb* = 'T' or 'C' and *n* > *ldb*
10. Error 2015 is recoverable or *naux* not equal to 0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows the computation ***C***←***AB***, where ***A***, ***B***, and ***C*** are contained in larger arrays A, B, and C, respectively. It shows how to code the calling sequence for SGEMMS, but does not use the Strassen algorithm for doing the computation. The calling sequence is shown below. The input and output, other than auxiliary storage, is the same as in Example 1 for SGEMUL.

Call Statement and Input:

```
                A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N   AUX  NAUX
                |   |    |     |   |    |     |   |   |   |   |    |    |
CALL SGEMMS( A , 8 , 'N'  , B , 6 , 'N'  , C , 7 , 6 , 5 , 4 , AUX , 0  )
```

Example 2

This example shows the computation ***C***←***AB***^{H}, where ***A*** and ***C*** are contained in larger arrays A and C, respectively, and ***B*** is the same size as the array B in which it is contained. The arrays contain complex data. This example shows how to code the calling sequence for CGEMMS, but does not use the Strassen algorithm for doing the computation. The calling sequence is shown below. The input and output, other than auxiliary storage, is the same as in Example 8 for CGEMUL.

Call Statement and Input:

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N   AUX  NAUX
             |   |    |     |   |    |     |   |   |   |   |    |    |
CALL CGEMMS( A , 4 , 'N'  , B , 3 , 'C'  , C , 4 , 3 , 2 , 3 , AUX , 0 )
```

SGEMM, DGEMM, CGEMM, and ZGEMM (Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes)

Purpose

SGEMM and DGEMM can perform any one of the following combined matrix computations, using scalars α and β, matrices ***A*** and ***B*** or their transposes, and matrix ***C***:

SGEMM and DGEMM Combined Matrix Computations

$\boldsymbol{C} \leftarrow \alpha \boldsymbol{AB}+\beta \boldsymbol{C}$	$\boldsymbol{C} \leftarrow \alpha \boldsymbol{AB}^T+\beta \boldsymbol{C}$
$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^T\boldsymbol{B}+\beta \boldsymbol{C}$	$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^T\boldsymbol{B}^T+\beta \boldsymbol{C}$

CGEMM and ZGEMM can perform any one of the following combined matrix computations, using scalars α and β, matrices ***A*** and ***B***, their transposes or their conjugate transposes, and matrix ***C***:

CGEMM and ZGEMM Combined Matrix Computations

$\boldsymbol{C} \leftarrow \alpha \boldsymbol{AB}+\beta \boldsymbol{C}$	$\boldsymbol{C} \leftarrow \alpha \boldsymbol{AB}^T+\beta \boldsymbol{C}$	$\boldsymbol{C} \leftarrow \alpha \boldsymbol{AB}^H+\beta \boldsymbol{C}$
$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^T\boldsymbol{B}+\beta \boldsymbol{C}$	$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^T\boldsymbol{B}^T+\beta \boldsymbol{C}$	$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^T\boldsymbol{B}^H+\beta \boldsymbol{C}$
$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^H\boldsymbol{B}+\beta \boldsymbol{C}$	$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^H\boldsymbol{B}^T+\beta \boldsymbol{C}$	$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^H\boldsymbol{B}^H+\beta \boldsymbol{C}$

Table 107: Data Types.

Data Types	
A, ***B***, ***C***, α, β	**Subroutine**
Short-precision real	SGEMM
Long-precision real	DGEMM
Short-precision complex	CGEMM
Long-precision complex	ZGEMM

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SGEMM | DGEMM | CGEMM | ZGEMM (*transa*, *transb*, *l*, *n*, *m*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*)
C and C++	sgemm | dgemm | cgemm | zgemm (*transa*, *transb*, *l*, *n*, *m*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*);
CBLAS	cblas_sgemm | cblas_dgemm | cblas_cgemm | cblas_zgemm (*cblas_layout*, *cblas_transa*, *cblas_transb*, *l*, *n*, *m*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*);

On Entry

cblas_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

transa

indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation.

Specified as: a single character; *transa* = 'N', 'T', or 'C'.

cblas_transa

indicates the form of matrix ***A*** to use in the computation, where:

If *cblas_transa* = CblasNoTrans, ***A*** is used in the computation.

If *cblas_transa* = CblasTrans, $\boldsymbol{A}^T$ is used in the computation.

If *cblas_transa* = CblasConjTrans, $\boldsymbol{A}^H$ is used in the computation.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

transb

indicates the form of matrix ***B*** to use in the computation, where:

If *transb* = 'N', ***B*** is used in the computation.

If *transb* = 'T', $\boldsymbol{B}^T$ is used in the computation.

If *transb* = 'C', $\boldsymbol{B}^H$ is used in the computation.

Specified as: a single character; *transb* = 'N', 'T', or 'C'.

cblas_transb

indicates the form of matrix ***B*** to use in the computation, where:

If *cblas_transb* = CblasNoTrans, ***B*** is used in the computation.

If *cblas_transb* = CblasTrans, $\boldsymbol{B}^T$ is used in the computation.

If *cblas_transb* = CblasConjTrans, $\boldsymbol{B}^H$ is used in the computation.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

l

is the number of rows in matrix ***C***.

Specified as: an integer; $0 \le l \le ldc$.

n

is the number of columns in matrix ***C***.

Specified as: an integer; $n \ge 0$.

m

has the following meaning, where:

If *transa* = 'N', it is the number of columns in matrix ***A***.

If *transa* = 'T' or 'C', it is the number of rows in matrix ***A***.

In addition:

If *transb* = 'N', it is the number of rows in matrix ***B***.

If *transb* = 'T' or 'C', it is the number of columns in matrix ***B***.

Specified as: an integer; $m \geq 0$.

alpha
is the scalar α.

Specified as: a number of the data type indicated in Table 107 on page 413.

a
is the matrix ***A***, where:

If *transa* = 'N', ***A*** is used in the computation, and ***A*** has *l* rows and *m* columns.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation, and ***A*** has *m* rows and *l* columns.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation, and ***A*** has *m* rows and *l* columns.

Note: No data should be moved to form $\boldsymbol{A}^T$ or $\boldsymbol{A}^H$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 107 on page 413, where:

If *transa* = 'N', its size must be *lda* by (at least) *m*.

If *transa* = 'T' or 'C', its size must be *lda* by (at least) *l*.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and:

If *transa* = 'N', $lda \geq l$.

If *transa* = 'T' or 'C', $lda \geq m$.

b
is the matrix ***B***, where:

If *transb* = 'N', ***B*** is used in the computation, and ***B*** has *m* rows and *n* columns.

If *transb* = 'T', $\boldsymbol{B}^T$ is used in the computation, and ***B*** has *n* rows and *m* columns.

If *transb* = 'C', $\boldsymbol{B}^H$ is used in the computation, and ***B*** has *n* rows and *m* columns.

Note: No data should be moved to form $\boldsymbol{B}^T$ or $\boldsymbol{B}^H$; that is, the matrix ***B*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 107 on page 413, where:

If *transb* = 'N', its size must be *ldb* by (at least) *n*.

If *transb* = 'T' or 'C', its size must be *ldb* by (at least) *m*.

ldb
is the leading dimension of the array specified for *b*.

Specified as: an integer; $ldb > 0$ and:

If *transb* = 'N', $ldb \geq m$.

If *transb* = 'T' or 'C', $ldb \geq n$.

beta
is the scalar β.

Specified as: a number of the data type indicated in Table 107 on page 413.

c

is the *l* by *n* matrix ***C***.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 107 on page 413.

ldc

is the leading dimension of the array specified for *c*.

Specified as: an integer; *ldc* > 0 and *ldc* ≥ *l*.

On Return

c

is the *l* by *n* matrix ***C***, containing the results of the computation. Returned as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 107 on page 413.

Notes

1. All subroutines accept lowercase letters for the *transa* and *transb* arguments.
2. For SGEMM and DGEMM, if you specify 'C' for the *transa* or *transb* argument, it is interpreted as though you specified 'T'.
3. Matrix ***C*** must have no common elements with matrices ***A*** or ***B***; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

The combined matrix addition and multiplication is expressed as follows, where a_{ik}, b_{kj}, and c_{ij} are elements of matrices ***A, B,*** and ***C***, respectively:

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} a_{ik} b_{kj} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A} \boldsymbol{B} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} a_{ki} b_{kj} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{\mathrm{T}} \boldsymbol{B} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} \bar{a}_{ki} b_{kj} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{\mathrm{H}} \boldsymbol{B} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} a_{ik} b_{jk} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A} \boldsymbol{B}^{\mathrm{T}} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} a_{ki} b_{jk} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{\mathrm{T}} \boldsymbol{B}^{\mathrm{T}} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} \bar{a}_{ki} b_{jk} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{\mathrm{H}} \boldsymbol{B}^{\mathrm{T}} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} a_{ik} \bar{b}_{jk} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A} \boldsymbol{B}^{\mathrm{H}} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} a_{ki} \bar{b}_{jk} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{\mathrm{T}} \boldsymbol{B}^{\mathrm{H}} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} \bar{a}_{ki} \bar{b}_{jk} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{\mathrm{H}} \boldsymbol{B}^{\mathrm{H}} + \beta \boldsymbol{C}$$

$$\text{for} \quad i = 1,\ l \ \text{ and } \ j = 1, n$$

See references [“42” on page 1255] and [“48” on page 1255]. In the following three cases, no computation is performed:

- *l* is 0.
- *n* is 0.

- β is 1 and α is 0.

Assuming the above conditions do not exist, if β ≠ 1 and *m* is 0, then β***C*** is returned.

Special Usage

Equivalence Rules

The equivalence rules, defined for matrix multiplication of ***A*** and ***B*** in "Special Usage" on page 402, also apply to the matrix multiplication part of the computation performed by this subroutine. You should use the equivalent rules when you want to transpose or conjugate transpose the multiplication part of the computation. When coding the calling sequences for these cases, be careful to code your matrix arguments and dimension arguments in the order indicated by the rule. Also, be careful that your input and output array C has dimensions large enough to hold the resulting matrix. See Example 4.

Error conditions

Resource Errors

Unable to allocate internal work area.

Computational Errors

None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. *lda*, *ldb*, *ldc* ≤ 0
3. *l*, *m*, *n* < 0
4. *l* > *ldc*
5. *transa*, *transb* ≠ 'N', 'T', or 'C'
6. *transa* = 'N' and *l* > *lda*
7. *transa* = 'T' or 'C' and *m* > *lda*
8. *cblas_transa* ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
9. *cblas_transa* = CblasNoTrans and *l* > *lda*
10. *cblas_transa* = CblasTrans, or CblasConjTrans and *m* > *lda*
11. *transb* = 'N' and *m* > *ldb*
12. *transb* = 'T' or 'C' and *n* > *ldb*
13. *cblas_transb* ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
14. *cblas_transb* = CblasNoTrans and *m* > *ldb*
15. *cblas_transb* = CblasTrans, or CblasConjTrans and *n* > *ldb*

Examples

Example 1

This example shows the computation ***C***←α***AB***+β***C***, where ***A***, ***B***, and ***C*** are contained in larger arrays A, B, and C, respectively.

Call Statement and Input:

```
              TRANSA TRANSB  L    N    M   ALPHA   A   LDA   B   LDB   BETA   C   LDC
                |      |     |    |    |     |     |    |    |    |     |     |    |
CALL SGEMM( 'N'   , 'N'   , 6 , 4 , 5 , 1.0 , A , 8 , B , 6 , 2.0 , C , 7 )
```

```
         |  1.0   2.0  -1.0  -1.0   4.0 |
        |  2.0   0.0   1.0   1.0  -1.0 |
        |  1.0  -1.0  -1.0   1.0   2.0 |
A    =  | -3.0   2.0   2.0   2.0   0.0 |
        |  4.0   0.0  -2.0   1.0  -1.0 |
        | -1.0  -1.0   1.0  -3.0   2.0 |
        |   .     .     .     .     .  |
        |   .     .     .     .     .  |
```

```
        |  1.0  -1.0   0.0   2.0 |
        |  2.0   2.0  -1.0  -2.0 |
B    =  |  1.0   0.0  -1.0   1.0 |
        | -3.0  -1.0   1.0  -1.0 |
        |  4.0   2.0  -1.0   1.0 |
        |   .     .     .     .  |
```

```
        | 0.5  0.5  0.5  0.5 |
        | 0.5  0.5  0.5  0.5 |
        | 0.5  0.5  0.5  0.5 |
C    =  | 0.5  0.5  0.5  0.5 |
        | 0.5  0.5  0.5  0.5 |
        | 0.5  0.5  0.5  0.5 |
        |  .    .    .    .  |
```

Output:

```
        | 24.0  13.0  -5.0   3.0 |
        | -3.0  -4.0   2.0   4.0 |
        |  4.0   1.0   2.0   5.0 |
C    =  | -2.0   6.0  -1.0  -9.0 |
        | -4.0  -6.0   5.0   5.0 |
        | 16.0   7.0  -4.0   7.0 |
        |   .     .     .     .  |
```

Example 2

This example shows the computation ***C***←α***AB***^{T}+β***C***, where ***A*** and ***C*** are contained in larger arrays A and C, respectively, and ***B*** is the same size as array B in which it is contained.

Call Statement and Input:

```
            TRANSA TRANSB  L   N   M  ALPHA  A  LDA  B  LDB  BETA  C  LDC
              |      |     |   |   |    |    |   |   |   |    |    |   |
CALL SGEMM( 'N'   , 'T'  , 3 , 3 , 2 , 1.0 , A , 4 , B , 3 , 2.0 , C , 5 )
```

```
        | 1.0  -3.0 |
A    =  | 2.0   4.0 |
        | 1.0  -1.0 |
        |  .     .  |
```

```
        | 1.0  -3.0 |
B    =  | 2.0   4.0 |
        | 1.0  -1.0 |
```

```
        | 0.5  0.5  0.5 |
        | 0.5  0.5  0.5 |
C    =  | 0.5  0.5  0.5 |
        |  .    .    .  |
        |  .    .    .  |
```

Output:

```
         | 11.0  -9.0   5.0 |
         | -9.0  21.0  -1.0 |
C    =   |  5.0  -1.0   3.0 |
         |   .     .     .  |
         |   .     .     .  |
```

Example 3

This example shows the computation ***C***←α***AB***+β***C*** using complex data, where ***A***, ***B***, and ***C*** are contained in larger arrays, A, B, and C, respectively.

Call Statement and Input:

```
           TRANSA TRANSB  L   N   M   ALPHA   A  LDA  B  LDB  BETA   C  LDC
             |      |     |   |   |     |     |   |   |   |    |     |   |
CALL CGEMM( 'N'  , 'N'  , 6 , 2 , 3 , ALPHA , A , 8 , B , 4 , BETA , C , 8 )
```

```
ALPHA    =  (1.0, 0.0)
BETA     =  (2.0, 0.0)


         | (1.0, 5.0)  (9.0, 2.0)  (1.0, 9.0) |
         | (2.0, 4.0)  (8.0, 3.0)  (1.0, 8.0) |
         | (3.0, 3.0)  (7.0, 5.0)  (1.0, 7.0) |
A    =   | (4.0, 2.0)  (4.0, 7.0)  (1.0, 5.0) |
         | (5.0, 1.0)  (5.0, 1.0)  (1.0, 6.0) |
         | (6.0, 6.0)  (3.0, 6.0)  (1.0, 4.0) |
         |     .           .           .      |
         |     .           .           .      |
```

```
         | (1.0, 8.0)  (2.0, 7.0) |
B    =   | (4.0, 4.0)  (6.0, 8.0) |
         | (6.0, 2.0)  (4.0, 5.0) |
         |     .           .      |
```

```
         | (0.5, 0.0)  (0.5, 0.0) |
         | (0.5, 0.0)  (0.5, 0.0) |
         | (0.5, 0.0)  (0.5, 0.0) |
C    =   | (0.5, 0.0)  (0.5, 0.0) |
         | (0.5, 0.0)  (0.5, 0.0) |
         | (0.5, 0.0)  (0.5, 0.0) |
         |     .           .      |
         |     .           .      |
```

Output:

```
         | (-22.0, 113.0)  (-35.0, 142.0) |
         | (-19.0, 114.0)  (-35.0, 141.0) |
         | (-20.0, 119.0)  (-43.0, 146.0) |
C    =   | (-27.0, 110.0)  (-58.0, 131.0) |
         |   (8.0, 103.0)    (0.0, 112.0) |
         | (-55.0, 116.0)  (-75.0, 135.0) |
         |       .               .        |
         |       .               .        |
```

Example 4

This example shows how to obtain the conjugate transpose of ***AB***^{H}.

$$\left(AB^{H}\right)^{H} = \overline{\overline{B}A}^{T} = BA^{H}$$

This shows the conjugate transpose of the computation performed in Example 8 for CGEMUL, which uses the following calling sequence:

```
CALL CGEMUL( A , 4 , 'N' , B , 3 , 'C' , C , 4 , 3 , 2 , 3 )
```

You instead code the calling sequence for ***C***←β***C***+α***BA***^{H}, where β = 0, α = 1, and the array ***C*** has the correct dimensions to receive the transposed matrix. Because β is zero, β***C*** = 0. For a description of all the matrix identities, see "Special Usage" on page 402.

Call Statement and Input:

```
            TRANSA TRANSB  L    N    M   ALPHA    A  LDA  B  LDB  BETA    C  LDC
              |      |     |    |    |     |      |   |   |   |     |     |   |
CALL CGEMM( 'N'   , 'C'   , 3 , 3 , 2 , ALPHA , B , 3 , A , 3 , BETA , C , 4 )
```

```
ALPHA    =  (1.0, 0.0)
BETA     =  (0.0, 0.0)


        | (1.0, 3.0)  (-3.0, 2.0) |
B    =  | (2.0, 5.0)   (4.0, 6.0) |
        | (1.0, 1.0)  (-1.0, 9.0) |
```

```
        | (1.0, 2.0)  (-3.0, 2.0) |
A    =  | (2.0, 6.0)   (4.0, 5.0) |
        | (1.0, 2.0)  (-1.0, 8.0) |
        |     .            .      |
```

C =(not relevant)

Output:

```
        | (20.0,   1.0)  (18.0, 23.0)  (26.0,  23.0) |
C    =  | (12.0, -25.0)  (80.0,  2.0)  (56.0, -37.0) |
        | (24.0, -26.0)  (49.0, 37.0)  (76.0,  -2.0) |
        |      .              .              .       |
```

Example 5

This example shows the computation ***C***←α***A***^{T}***B***^{H}+β***C*** using complex data, where ***A, B***, and ***C*** are the same size as the arrays A, B, and C, in which they are contained. Because β is zero, β***C*** = 0. (Based on the dimensions of the matrices, ***A*** is actually a column vector, and ***C*** is actually a row vector.)

Call Statement and Input:

```
            TRANSA TRANSB  L    N    M   ALPHA    A  LDA  B  LDB  BETA    C  LDC
              |      |     |    |    |     |      |   |   |   |     |     |   |
CALL CGEMM( 'T'   , 'C'   , 1 , 3 , 3 , ALPHA , A , 3 , B , 3 , BETA , C , 1 )
```

```
ALPHA    =  (1.0, 1.0)
BETA     =  (0.0, 0.0)


        | (1.0,  2.0) |
A    =  | (2.0,  5.0) |
        | (1.0,  6.0) |
```

```
        | (1.0, 6.0)  (-3.0, 4.0)   (2.0, 6.0) |
B    =  | (2.0, 3.0)   (4.0, 6.0)   (0.0, 3.0) |
        | (1.0, 3.0)  (-1.0, 6.0)  (-1.0, 9.0) |
```

C =(not relevant)

Output:

```
C    =  | (86.0, 44.0) (58.0, 70.0) (121.0, 55.0) |
```

SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM (Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian)

Purpose

These subroutines compute one of the following matrix-matrix products, using the scalars α and β and matrices ***A***, ***B***, and ***C***:

1. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{AB} + \beta \boldsymbol{C}$
2. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{BA} + \beta \boldsymbol{C}$

where matrix ***A*** is stored in either upper or lower storage mode, and:

- For SSYMM and DSYMM, matrix ***A*** is real symmetric.
- For CSYMM and ZSYMM, matrix ***A*** is complex symmetric.
- For CHEMM and ZHEMM, matrix ***A*** is complex Hermitian.

Table 108: Data Types.

Data Types	
α, ***A***, ***B***, β, ***C***	**Subprogram**
Short-precision real	SSYMM
Long-precision real	DSYMM
Short-precision complex	CSYMM and CHEMM
Long-precision complex	ZSYMM and ZHEMM

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	**Syntax**
Fortran	CALL SSYMM | DSYMM | CSYMM | ZSYMM | CHEMM | ZHEMM (*side*, *uplo*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*)
C and C++	ssymm | dsymm | csymm | zsymm | chemm | zhemm (*side*, *uplo*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*);

Language	Syntax
CBLAS	cblas_ssymm | cblas_dsymm | cblas_csymm | cblas_zsymm | cblas_chemm | cblas_zhemm (*cblas_layout*, *cblas_side*, *cblas_uplo*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*);

On Entry

cblas_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

side

indicates whether matrix ***A*** is located to the left or right of rectangular matrix ***B*** in the equation used for this computation, where:

If *side* = 'L', ***A*** is to the left of ***B***, resulting in equation 1.

If *side* = 'R', ***A*** is to the right of ***B***, resulting in equation 2.

Specified as: a single character. It must be 'L' or 'R'.

cblas_side

indicates whether matrix ***A*** is located to the left or right of rectangular matrix ***B*** in the equation used for this computation, where:

If *cblas_side* = CblasLeft, ***A*** is to the left of ***B***, resulting in equation 1.

If *cblas_side* = CblasRight, ***A*** is to the right of ***B***, resulting in equation 2.

Specified as: an object of enumerated type CBLAS_SIDE. It must be CblasLeft or CblasRight.

uplo

indicates the storage mode used for matrix ***A***, where:

If *uplo* = 'U', ***A*** is stored in upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

cblas_uplo

indicates the storage mode used for matrix ***A***, where:

If *cblas_uplo* = CblasUpper, ***A*** is stored in upper storage mode.

If *cblas_uplo* = CblasLower, ***A*** is stored in lower storage mode.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

m

is the number of rows in rectangular matrices ***B*** and ***C***, and:

If *side* = 'L', *m* is the order of matrix ***A***.

Specified as: an integer; $0 \le m \le ldb$, $m \le ldc$, and:

If *side* = 'L', $m \le lda$.

n

is the number of columns in rectangular matrices ***B*** and ***C***, and:

If *side* = 'R', *n* is the order of matrix ***A***.

Specified as: an integer; $n \geq 0$ and:

If *side* = 'R', $n \leq lda$.

alpha
is the scalar α.

Specified as: a number of the data type indicated in Table 108 on page 422.

a
is the real symmetric, complex symmetric, or complex Hermitian matrix ***A***, where:

If *side* = 'L', ***A*** is order *m*.

If *side* = 'R', ***A*** is order *n*.

and where it is stored as follows:

If *uplo* = 'U', ***A*** is stored in upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower storage mode.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 108 on page 422, where:

If *side* = 'L', its size must be *lda* by (at least) *m*.

If *side* = 'R', it size must be *lda* by (at least) *n*.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and:

If *side* = 'L', $lda \geq m$.

If *side* = 'R', $lda \geq n$.

b
is the *m* by *n* rectangular matrix ***B***.

Specified as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 108 on page 422.

ldb
is the leading dimension of the array specified for *b*.

Specified as: an integer; $ldb > 0$ and $ldb \geq m$.

beta
is the scalar β.

Specified as: a number of the data type indicated in Table 108 on page 422.

c
is the *m* by *n* rectangular matrix ***C***.

Specified as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 108 on page 422.

ldc
is the leading dimension of the array specified for *c*.

Specified as: an integer; $ldc > 0$ and $ldc \geq m$.

On Return

c
is the *m* by *n* matrix ***C***, containing the results of the computation.

Returned as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 108 on page 422.

Notes

1. These subroutines accept lowercase letters for the *side* and *uplo* arguments.
2. Matrices ***A***, ***B***, and ***C*** must have no common elements; otherwise, results are unpredictable.
3. If matrix ***A*** is upper triangular (*uplo* = 'U'), these subroutines use only the data in the upper triangular portion of the array. If matrix ***A*** is lower triangular, (*uplo* = 'L'), these subroutines use only the data in the lower triangular portion of the array. In each case, the other portion of the array is altered during the computation, but restored before exit.
4. The imaginary parts of the diagonal elements of a complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.
5. For a description of how symmetric matrices are stored in upper and lower storage mode, see "Symmetric Matrix" on page 80. For a description of how complex Hermitian matrices are stored in upper and lower storage mode, see "Complex Hermitian Matrix" on page 84.

Function

These subroutines can perform the following matrix-matrix product computations using matrix ***A***, which is real symmetric for SSYMM and DSYMM, complex symmetric for CSYMM and ZSYMM, and complex Hermitian for CHEMM and ZHEMM:

1. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{AB} + \beta \boldsymbol{C}$
2. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{BA} + \beta \boldsymbol{C}$

   where:

   α and β are scalars.

   A is a matrix of the type indicated above, stored in upper or lower storage mode. It is order *m* for equation 1 and order *n* for equation 2.

   B and ***C*** are *m* by *n* rectangular matrices.

See references ["42" on page 1255] and ["48" on page 1255]. In the following two cases, no computation is performed:

- *n* or *m* is 0.
- β is one and α is zero.

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. $m < 0$
3. $m > ldb$
4. $m > ldc$
5. $n < 0$
6. *lda*, *ldb*, *ldc* ≤ 0
7. *side* ≠ 'L' or 'R'
8. *side* = 'L' and $m > lda$
9. *side* = 'R' and $n > lda$

10. *cblas_side* ≠ CblasLeft or CblasRight
11. *cblas_side* = CblasLeft and *m* > *lda*
12. *cblas_side* = CblasRight and *n* > *lda*
13. *uplo* ≠ 'L' or 'U'
14. *cblas_uplo* ≠ CblasLower or CblasUpper

Examples

Example 1

This example shows the computation ***C***←α***AB***+β***C***, where ***A*** is a real symmetric matrix of order 5, stored in upper storage mode, and ***B*** and ***C*** are 5 by 4 rectangular matrices.

Call Statement and Input:

```
            SIDE  UPLO  M   N   ALPHA   A  LDA  B  LDC  BETA  C  LDC
             |     |    |   |     |     |   |   |   |    |    |   |
CALL SSYMM( 'L' , 'U' , 5 , 4 ,  2.0  , A , 8 , B , 6 , 1.0 , C , 5 )
```

```
         ┌                            ┐
         | 1.0  2.0  -1.0  -1.0   4.0 |
         |  .   0.0   1.0   1.0  -1.0 |
         |  .    .   -1.0   1.0   2.0 |
A    =   |  .    .     .    2.0   0.0 |
         |  .    .     .     .   -1.0 |
         |  .    .     .     .     .  |
         |  .    .     .     .     .  |
         |  .    .     .     .     .  |
                                     ┘
```

```
         ┌                        ┐
         |  1.0  -1.0   0.0   2.0 |
         |  2.0   2.0  -1.0  -2.0 |
B    =   |  1.0   0.0  -1.0   1.0 |
         | -3.0  -1.0   1.0  -1.0 |
         |  4.0   2.0  -1.0   1.0 |
         |   .     .     .     .  |
                                 ┘
```

```
         ┌                        ┐
         | 23.0  12.0  -6.0   2.0 |
         | -4.0  -5.0   1.0   3.0 |
C    =   |  5.0   6.0  -1.0  -4.0 |
         | -4.0   1.0   0.0  -5.0 |
         |  8.0  -4.0  -2.0  13.0 |
                                 ┘
```

Output:

```
         ┌                            ┐
         |  69.0   36.0  -18.0    6.0 |
         | -12.0  -15.0    3.0    9.0 |
C    =   |  15.0   18.0   -3.0  -12.0 |
         | -12.0    3.0    0.0  -15.0 |
         |   8.0  -20.0   -2.0   35.0 |
                                     ┘
```

Example 2

This example shows the computation ***C***←α***AB***+β***C***, where ***A*** is a real symmetric matrix of order 3, stored in lower storage mode, and ***B*** and ***C*** are 3 by 6 rectangular matrices.

Call Statement and Input:

```
            SIDE  UPLO  M   N   ALPHA   A  LDA  B  LDB  BETA  C  LDC
             |     |    |   |     |     |   |   |   |     |   |   |
CALL SSYMM( 'L' , 'L' , 3 , 6 ,  2.0  , A , 4 , B , 3 , 2.0 , C , 5 )
```

```
        ┌                 ┐
        | 1.0    .     .  |
A    =  | 2.0   4.0    .  |
        | 1.0  -1.0  -1.0 |
        |  .     .     .  |
                         ┘
```

```
        ┌                                   ┐
        | 1.0  -3.0   2.0   2.0  -1.0   2.0 |
B    =  | 2.0   4.0   0.0   0.0   1.0  -2.0 |
        | 1.0  -1.0  -1.0  -1.0  -1.0   1.0 |
                                          ┘
```

```
        ┌                                  ┐
        |  6.0   4.0  1.0  1.0   0.0  -1.0 |
        |  9.0  11.0  5.0  5.0   3.0  -5.0 |
C    =  | -2.0  -6.0  3.0  3.0  -1.0  32.0 |
        |   .     .    .    .     .     .  |
        |   .     .    .    .     .     .  |
                                         ┘
```

Output:

```
        ┌                                        ┐
        | 24.0   16.0   4.0   4.0   0.0   -4.0 |
        | 36.0   44.0  20.0  20.0  12.0  -20.0 |
C    =  | -8.0  -24.0  12.0  12.0  -4.0   12.0 |
        |   .      .     .     .     .      .  |
        |   .      .     .     .     .      .  |
                                             ┘
```

Example 3

This example shows the computation ***C***←α***BA***+β***C***, where ***A*** is a real symmetric matrix of order 3, stored in upper storage mode, and ***B*** and ***C*** are 2 by 3 rectangular matrices.

Call Statement and Input:

```
            SIDE  UPLO  M   N   ALPHA   A  LDA  B  LDB  BETA  C  LDC
             |     |    |   |     |     |   |   |   |     |   |   |
CALL SSYMM( 'R' , 'U' , 2 , 3 ,  2.0  , A , 4 , B , 3 , 1.0 , C , 5 )
```

```
        ┌                 ┐
        | 1.0  -3.0   1.0 |
A    =  |  .    4.0  -1.0 |
        |  .     .    2.0 |
        |  .     .     .  |
                         ┘
```

```
        ┌                 ┐
        | 1.0  -3.0   3.0 |
B    =  | 2.0   4.0  -1.0 |
        |  .     .     .  |
                         ┘
```

```
        ┌                    ┐
        |  13.0  -18.0  10.0 |
        | -11.0   11.0  -4.0 |
C    =  |    .      .     .  |
        |    .      .     .  |
        |    .      .     .  |
                           ┘
```

Output:

```
        ┌                        ┐
        |  39.0  -54.0   30.0 |
        | -33.0   33.0  -12.0 |
C    =  |    .      .      .  |
        |    .      .      .  |
        |    .      .      .  |
                             ┘
```

Example 4

This example shows the computation ***C***←α***BA***+β***C***, where ***A*** is a real symmetric matrix of order 3, stored in lower storage mode, and ***B*** and ***C*** are 3 by 3 square matrices.

Call Statement and Input:

```
            SIDE  UPLO  M   N   ALPHA   A  LDA  B  LDB  BETA  C  LDC
             |     |    |   |     |     |   |   |   |    |    |   |
CALL SSYMM( 'R' , 'L' , 3 , 3 , -1.0  , A , 3 , B , 3 , 1.0 , C , 3 )
```

```
        ┌                ┐
        | 1.0    .     .  |
A    =  | 2.0  10.0    .  |
        | 1.0  11.0  4.0 |
                         ┘
```

```
        ┌                 ┐
        | 1.0  -3.0   2.0 |
B    =  | 2.0   4.0   0.0 |
        | 1.0  -1.0  -1.0 |
                          ┘
```

```
        ┌                  ┐
        |  1.0   5.0  -9.0 |
C    =  | -3.0  10.0  -2.0 |
        | -2.0   8.0   0.0 |
                           ┘
```

Output:

```
        ┌                     ┐
        |   4.0   11.0   15.0 |
C    =  | -13.0  -34.0  -48.0 |
        |   0.0   27.0   14.0 |
                             ┘
```

Example 5

This example shows the computation ***C***←α***BA***+β***C***, where ***A*** is a complex symmetric matrix of order 3, stored in upper storage mode, and ***B*** and ***C*** are 2 by 3 rectangular matrices.

Call Statement and Input:

```
            SIDE  UPLO  M   N   ALPHA   A  LDA  B  LDB  BETA   C  LDC
             |     |    |   |     |     |   |   |   |    |     |   |
CALL CSYMM( 'R' , 'U' , 2 , 3 , ALPHA , A , 4 , B , 3 , BETA , C , 5 )
```

```
ALPHA    =  (2.0, 3.0)

BETA     =  (1.0, 6.0)
```

```
        ┌                                      ┐
        | (1.0, 5.0)  (-3.0, 2.0)   (1.0, 6.0) |
A    =  |     .         (4.0, 5.0)  (-1.0, 4.0) |
        |     .             .        (2.0, 5.0) |
         |     .          .          .    |
                                         ┘
```

```
         ┌                                ┐
         | (1.0, 1.0)  (-3.0, 2.0)   (3.0, 3.0) |
B    =   | (2.0, 6.0)   (4.0, 5.0)  (-1.0, 4.0) |
         |     .           .          .     |
                                          ┘
```

```
         ┌                                       ┐
         |  (13.0, 6.0)  (-18.0, 6.0)  (10.0, 7.0) |
         | (-11.0, 8.0)   (11.0, 1.0)  (-4.0, 2.0) |
C    =   |       .             .           .       |
         |       .             .           .       |
         |       .             .           .       |
                                                ┘
```

Output:

```
         ┌                                                      ┐
         |  (-96.0,   72.0)  (-141.0, -226.0)  (-112.0,   38.0) |
         | (-230.0, -269.0)  (-133.0,  -23.0)  (-272.0, -198.0) |
C    =   |        .                  .                 .        |
         |        .                  .                 .        |
         |        .                  .                 .        |
                                                              ┘
```

Example 6

This example shows the computation ***C***←α***BA***+β***C***, where ***A*** is a complex Hermitian matrix of order 3, stored in lower storage mode, and ***B*** and ***C*** are 3 by 3 square matrices.

Note: The imaginary parts of the diagonal elements of a complex Hermitian matrix are assumed to be zero, so you do not have to set these values.

Call Statement and Input:

```
            SIDE  UPLO  M   N   ALPHA   A  LDA  B  LDB  BETA   C  LDC
             |     |    |   |     |     |   |   |   |    |     |   |
CALL CHEMM( 'R' , 'L' , 2 , 3 , ALPHA , A , 4 , B , 3 , BETA , C , 5 )
```

```
ALPHA    =  (2.0, 3.0)

BETA     =  (1.0, 6.0)
```

```
         ┌                                  ┐
         |  (1.0,  . )       .          .      |
A    =   |  (3.0, 2.0)  (4.0,  . )      .      |
         | (-1.0, 6.0)  (1.0, 4.0)  (2.0,  . ) |
         |      .           .           .     |
                                          ┘
```

```
         ┌                                ┐
         | (1.0, 1.0)  (-3.0, 2.0)   (3.0, 3.0) |
B    =   | (2.0, 6.0)   (4.0, 5.0)  (-1.0, 4.0) |
         |     .           .          .     |
                                          ┘
```

```
         ┌                                       ┐
         |  (13.0, 6.0)  (-18.0, 6.0)  (10.0, 7.0) |
         | (-11.0, 8.0)   (11.0, 1.0)  (-4.0, 2.0) |
C    =   |       .             .           .       |
         |       .             .           .       |
         |       .             .           .       |
                                                ┘
```

Output:

```
         ┌                                                  ┐
         | (-137.0,  17.0)  (-158.0, -102.0)  (-39.0, 141.0) |
         | (-154.0, -77.0)   (-63.0,  186.0)  (159.0, 104.0) |
C    =  |         .                   .                   .         |
        |         .                   .                   .         |
        |         .                   .                   .         |
                                                                   ┘
```

STRMM, DTRMM, CTRMM, and ZTRMM (Triangular Matrix-Matrix Product)

Purpose

STRMM and DTRMM compute one of the following matrix-matrix products, using the scalar α, rectangular matrix ***B***, and triangular matrix ***A*** or its transpose:

Matrix-Matrix Products Computed by STRMM and DTRMM

1. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B}$	3. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{B}\boldsymbol{A}$
2. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{A}^{T}\boldsymbol{B}$	4. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{B}\boldsymbol{A}^{T}$

CTRMM and ZTRMM compute one of the following matrix-matrix products, using the scalar α, rectangular matrix ***B***, and triangular matrix ***A***, its transpose, or its conjugate transpose:

Matrix-Matrix Products Computed by CTRMM and ZTRMM

1. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B}$	3. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{B}\boldsymbol{A}$	5. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{A}^{H}\boldsymbol{B}$
2. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{A}^{T}\boldsymbol{B}$	4. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{B}\boldsymbol{A}^{T}$	6. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{B}\boldsymbol{A}^{H}$

Table 109: Data Types.

Data Types	
A, ***B***, α	**Subroutine**
Short-precision real	STRMM
Long-precision real	DTRMM
Short-precision complex	CTRMM
Long-precision complex	ZTRMM

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL STRMM | DTRMM | CTRMM | ZTRMM (*side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*)
C and C++	strmm | dtrmm | ctrmm | ztrmm (*side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*);
CBLAS	cblas_strmm | cblas_dtrmm | cblas_ctrmm | cblas_ztrmm (*cblas_layout*, *cblas_side*, *cblas_uplo*, *cblas_transa*, *cblas_diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*);

On Entry

cblas_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

side

indicates whether the triangular matrix ***A*** is located to the left or right of rectangular matrix ***B*** in the equation used for this computation, where:

If *side* = 'L', ***A*** is to the left of ***B*** in the equation, resulting in either equation 1, 2, or 5.

If *side* = 'R', ***A*** is to the right of ***B*** in the equation, resulting in either equation 3, 4, or 6.

Specified as: a single character. It must be 'L' or 'R'.

cblas_side

indicates whether matrix ***A*** is located to the left or right of rectangular matrix ***B*** in the equation used for this computation, where:

If *cblas_side* = CblasLeft, ***A*** is to the left of ***B*** in the equation, resulting in either equation 1, 2, or 5.

If *cblas_side* = CblasRight, ***A*** is to the right of ***B*** in the equation, resulting in either equation 3, 4, or 6.

Specified as: an object of enumerated type CBLAS_SIDE. It must be CblasLeft or CblasRight.

uplo

indicates whether matrix ***A*** is an upper or lower triangular matrix, where:

If *uplo* = 'U', ***A*** is an upper triangular matrix.

If *uplo* = 'L', ***A*** is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

cblas_uplo

indicates whether matrix ***A*** is an upper or lower triangular matrix, where:

If *cblas_uplo* = CblasUpper, ***A*** is an upper triangular matrix.

If *cblas_uplo* = CblasLower, ***A*** is a lower triangular matrix.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

transa

indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation, resulting in either equation 1 or 3.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation, resulting in either equation 2 or 4.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation, resulting in either equation 5 or 6.

Specified as: a single character. It must be 'N', 'T', or 'C'.

cblas_transa

indicates the form of matrix ***A*** to use in the computation, where:

If *cblas_transa* = CblasNoTrans, ***A*** is used in the computation, resulting in either equation 1 or 3.

If *cblas_transa* = CblasTrans, $\boldsymbol{A}^T$ is used in the computation, resulting in either equation 2 or 4.

If *cblas_transa* = CblasConjTrans, $\boldsymbol{A}^H$ is used in thecomputation, resulting in either equation 5 or 6.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

diag

indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

cblas_diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = CblasUnit, ***A*** is a unit triangular matrix.

If *diag* = CblasNonUnit ***A*** is not a unit triangular matrix.

Specified as: an object of enumerated type CBLAS_DIAG. It must be CblasNonUnit or CblasUnit.

m
is the number of rows in rectangular matrix ***B***, and:

If *side* = 'L', *m* is the order of triangular matrix ***A***.

Specified as: an integer, where:

If *side* = 'L', $0 \le m \le lda$ and $m \le ldb$.

If *side* = 'R', $0 \le m \le ldb$.

n
is the number of columns in rectangular matrix ***B***, and:

If *side* = 'R', *n* is the order of triangular matrix ***A***.

Specified as: an integer; $n \ge 0$ and:

If *side* = 'R', $n \le lda$.

alpha
is the scalar α.

Specified as: a number of the data type indicated in Table 109 on page 430.

a
is the triangular matrix ***A***, of which only the upper or lower triangular portion is used, where:

If *side* = 'L', ***A*** is order *m*.

If *side* = 'R', ***A*** is order *n*.

Note: No data should be moved to form $\boldsymbol{A}^T$ or $\boldsymbol{A}^H$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 109 on page 430, where:

If *side* = 'L', its size must be *lda* by (at least) *m*.

If *side* = 'R', it size must be *lda* by (at least) *n*.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and:

If *side* = 'L', $lda \ge m$.

If *side* = 'R', $lda \ge n$.

b
is the *m* by *n* rectangular matrix ***B***.

Specified as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 109 on page 430.

ldb
is the leading dimension of the array specified for *b*.

Specified as: an integer; *ldb* > 0 and *ldb* ≥ *m*.

On Return

b

is the *m* by *n* matrix ***B***, containing the results of the computation. Returned as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 109 on page 430.

Notes

1. These subroutines accept lowercase letters for the *side*, *uplo*, *transa*, and *diag* arguments.
2. For STRMM and DTRMM, if you specify 'C' for the *transa* argument, it is interpreted as though you specified 'T'.
3. Matrices ***A*** and ***B*** must have no common elements; otherwise, results are unpredictable.
4. ESSL assumes certain values in your array for parts of a triangular matrix. As a result, you do not have to set these values. For unit triangular matrices, the elements of the diagonal are assumed to be 1.0 for real matrices and (1.0, 0.0) for complex matrices. When using upper- or lower-triangular storage, the unreferenced elements in the lower and upper triangular part, respectively, are assumed to be zero.
5. For a description of triangular matrices and how they are stored, see "Triangular Matrix" on page 88.

Function

These subroutines can perform the following matrix-matrix product computations, using the triangular matrix ***A***, its transpose, or its conjugate transpose, where ***A*** can be either upper- or lower-triangular:

1. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{AB}$
2. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{A}^{T}\boldsymbol{B}$
3. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{A}^{H}\boldsymbol{B}$ (for CTRMM and ZTRMM only)

   where:

   αis a scalar.
   A is a triangular matrix of order *m*.
   B is an *m* by *n* rectangular matrix.
4. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{BA}$
5. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{BA}^{T}$
6. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{BA}^{H}$ (for CTRMM and ZTRMM only)

   where:

   α is a scalar.
   A is a triangular matrix of order *n*.
   B is an *m* by *n* rectangular matrix.

See references ["42" on page 1255] and ["48" on page 1255]. If *n* or *m* is 0, no computation is performed.

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor

2. *m* < 0
3. *m* > *ldb*
4. *n* < 0
5. *lda*, *ldb* ≤ 0
6. *side* ≠ 'L' or 'R'
7. *side* = 'L' and *m* > *lda*
8. *side* = 'R' and *n* > *lda*
9. *cblas_side* ≠ CblasLeft or CblasRight
10. *cblas_side* = CblasLeft and *m* > *lda*
11. *cblas_side* = CblasRight and *n* > *lda*
12. *uplo* ≠ 'L' or 'U'
13. *cblas_uplo* ≠ CblasLower or CblasUpper
14. *transa* ≠ 'T', 'N', or 'C'
15. *cblas_transa* ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
16. *diag* ≠ 'N' or 'U'
17. *cblas_diag* ≠ CblasNonUnit or CblasUnit

Examples

Example 1

This example shows the computation ***B***←α***AB***, where ***A*** is a 5 by 5 upper triangular matrix that is not unit triangular, and ***B*** is a 5 by 3 rectangular matrix.

Call Statement and Input:

```
             SIDE  UPLO TRANSA  DIAG  M   N   ALPHA   A  LDA  B  LDB
              |     |     |      |    |   |     |     |   |   |   |
CALL STRMM( 'L' , 'U' , 'N'  , 'N' , 5 , 3 ,  1.0  , A , 7 , B , 6 )
```

```
        ┌                             ┐
        | 3.0  -1.0   2.0   2.0   1.0 |
        |  .   -2.0   4.0  -1.0   3.0 |
        |  .     .   -3.0   0.0   2.0 |
A    =  |  .     .     .    4.0  -2.0 |
        |  .     .     .     .    1.0 |
        |  .     .     .     .     .  |
        |  .     .     .     .     .  |
                                      ┘
```

```
        ┌                 ┐
        |  2.0  3.0   1.0 |
        |  5.0  5.0   4.0 |
B    =  |  0.0  1.0   2.0 |
        |  3.0  1.0  -3.0 |
        | -1.0  2.0   1.0 |
        |   .    .     .  |
                          ┘
```

Output:

```
        ┌                    ┐
        |   6.0  10.0   -2.0 |
        | -16.0  -1.0    6.0 |
B    =  |  -2.0   1.0   -4.0 |
        |  14.0   0.0  -14.0 |
        |  -1.0   2.0    1.0 |
        |    .     .      .  |
                             ┘
```

Example 2

This example shows the computation **B**←α**A**^{T}**B**, where **A** is a 5 by 5 upper triangular matrix that is not unit triangular, and **B** is a 5 by 4 rectangular matrix.

Call Statement and Input:

```
             SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
              |     |     |      |    |   |      |    |   |   |   |
CALL STRMM( 'L' , 'U' , 'T'  , 'N' , 5 , 4 ,  1.0  , A , 7 , B , 6 )
```

```
         ┌                              ┐
         | -1.0  -4.0  -2.0   2.0   3.0 |
         |   .   -2.0   2.0   2.0   2.0 |
         |   .     .   -3.0  -1.0   4.0 |
A    =   |   .     .     .    1.0   0.0 |
         |   .     .     .     .   -2.0 |
         |   .     .     .     .     .  |
         |   .     .     .     .     .  |
                                       ┘
```

```
         ┌                        ┐
         |  1.0   2.0   3.0   4.0 |
         |  3.0   3.0  -1.0   2.0 |
B    =   | -2.0  -1.0   0.0   1.0 |
         |  4.0   4.0  -3.0  -3.0 |
         |  2.0   2.0   2.0   2.0 |
         |   .     .     .     .  |
                                 ┘
```

Output:

```
         ┌                          ┐
         | -1.0  -2.0   -3.0   -4.0 |
         |  2.0  -2.0  -14.0  -12.0 |
B    =   | 10.0   5.0   -8.0   -7.0 |
         | 14.0  15.0    1.0    8.0 |
         | -3.0   4.0    3.0   16.0 |
         |   .     .      .      .  |
                                   ┘
```

Example 3

This example shows the computation **B**←α**BA**, where **A** is a 5 by 5 lower triangular matrix that is not unit triangular, and **B** is a 3 by 5 rectangular matrix.

Call Statement and Input:

```
             SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
              |     |     |      |    |   |      |    |   |   |   |
CALL STRMM( 'R' , 'L' , 'N'  , 'N' , 3 , 5 ,  1.0  , A , 7 , B , 4 )
```

```
         ┌                           ┐
         | 2.0   .     .     .     . |
         | 2.0  3.0    .     .     . |
         | 2.0  1.0   1.0    .     . |
A    =   | 0.0  3.0   0.0  -2.0    . |
         | 2.0  4.0  -1.0   2.0  -1.0|
         |  .    .     .     .     . |
         |  .    .     .     .     . |
                                    ┘
```

```
         ┌                              ┐
         |  3.0   4.0  -1.0  -1.0  -1.0 |
B    =   |  2.0   1.0  -1.0   0.0   3.0 |
         | -2.0  -1.0  -3.0   0.0   2.0 |
         |   .     .     .     .     .  |
                                       ┘
```

Output:

```
        ┌                              ┐
        | 10.0   4.0   0.0  0.0   1.0 |
B    =  | 10.0  14.0  -4.0  6.0  -3.0 |
        | -8.0   2.0  -5.0  4.0  -2.0 |
        |   .     .     .    .     .  |
                                     ┘
```

Example 4

This example shows the computation ***B***←α***BA***, where ***A*** is a 6 by 6 upper triangular matrix that is unit triangular, and ***B*** is a 1 by 6 rectangular matrix.

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input:

```
            SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
             |     |     |      |    |   |      |    |   |   |   |
CALL STRMM( 'R' , 'U' , 'N'  , 'U' , 1 , 6 ,  1.0  , A , 7 , B , 2 )
```

```
        ┌                                ┐
        | .  2.0  -3.0  1.0   2.0   4.0 |
        | .   .    0.0  1.0   1.0  -2.0 |
        | .   .     .   4.0  -1.0   1.0 |
A    =  | .   .     .    .    0.0  -1.0 |
        | .   .     .    .     .    2.0 |
        | .   .     .    .     .     .  |
        | .   .     .    .     .     .  |
                                       ┘
```

```
        ┌                                ┐
B    =  | 1.0  2.0  1.0  3.0  -1.0  -2.0 |
        |  .    .    .    .     .     .  |
                                        ┘
```

Output:

```
        ┌                                ┐
B    =  | 1.0  4.0  -2.0  10.0  2.0  -6.0 |
        |  .    .     .     .    .     .  |
                                        ┘
```

Example 5

This example shows the computation ***B***←α$\boldsymbol{A}^H$***B***, where ***A*** is a 5 by 5 upper triangular matrix that is not unit triangular, and ***B*** is a 5 by 1 rectangular matrix.

Call Statement and Input:

```
            SIDE  UPLO TRANSA  DIAG  M   N    ALPHA   A  LDA  B  LDB
             |     |     |      |    |   |      |     |   |   |   |
CALL CTRMM( 'L' , 'U' , 'C'  , 'N' , 5 , 1 ,  ALPHA , A , 6 , B , 6 )

ALPHA    =  (1.0, 0.0)
```

```
        ┌                                                                 ┐
        | (-4.0, 1.0)  (4.0, -3.0)  (-1.0,  3.0)  (0.0,  0.0)  (-1.0,  0.0) |
        |      .       (-2.0,  0.0)  (-3.0, -1.0) (-2.0, -1.0)   (4.0,  3.0) |
A    =  |      .            .        (-5.0,  3.0) (-3.0, -3.0)  (-5.0, -5.0) |
        |      .            .             .        (4.0, -4.0)   (2.0,  0.0) |
        |      .            .             .             .        (2.0, -1.0) |
        |      .            .             .             .             .      |
                                                                            ┘
```

```
        ┌             ┐
        |  (3.0, 4.0) |
        | (-4.0, 2.0) |
B    =  | (-5.0, 0.0) |
        |  (1.0, 3.0) |
        |  (3.0, 1.0) |
        |      .      |
                     ┘
```

Output:

```
         ┌                ┐
         | (-8.0, -19.0)  |
         |  (8.0,  21.0)  |
B    =   | (44.0,  -8.0)  |
         | (13.0,  -7.0)  |
         | (19.0,   2.0)  |
         |      .         |
                         ┘
```

STRSM, DTRSM, CTRSM, and ZTRSM (Solution of Triangular Systems of Equations with Multiple Right-Hand Sides)

Purpose

STRSM and DTRSM perform one of the following solves for a triangular system of equations with multiple right-hand sides, using scalar α, rectangular matrix ***B***, and triangular matrix ***A*** or its transpose:

Solution	Equation
1. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-1})\boldsymbol{B}$	$\boldsymbol{AX} = \alpha\boldsymbol{B}$
2. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-T})\boldsymbol{B}$	$\boldsymbol{A}^T\boldsymbol{X} = \alpha\boldsymbol{B}$
3. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-1})$	$\boldsymbol{XA} = \alpha\boldsymbol{B}$
4. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-T})$	$\boldsymbol{XA}^T = \alpha\boldsymbol{B}$

CTRSM and ZTRSM perform one of the following solves for a triangular system of equations with multiple right-hand sides, using scalar α, rectangular matrix ***B***, and triangular matrix ***A***, its transpose, or its conjugate transpose:

Solution	Equation
1. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-1})\boldsymbol{B}$	$\boldsymbol{AX} = \alpha\boldsymbol{B}$
2. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-T})\boldsymbol{B}$	$\boldsymbol{A}^T\boldsymbol{X} = \alpha\boldsymbol{B}$
3. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-1})$	$\boldsymbol{XA} = \alpha\boldsymbol{B}$
4. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-T})$	$\boldsymbol{XA}^T = \alpha\boldsymbol{B}$
5. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-H})\boldsymbol{B}$	$\boldsymbol{A}^H\boldsymbol{X} = \alpha\boldsymbol{B}$
6. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-H})$	$\boldsymbol{XA}^H = \alpha\boldsymbol{B}$

Note: The term ***X*** used in the systems of equations listed above represents the output solution matrix. It is important to note that in these subroutines the solution matrix is actually returned in the input-output argument *b*.

Table 110: Data Types.

Data Types	
A, ***B***, α	**Subroutine**
Short-precision real	STRSM
Long-precision real	DTRSM
Short-precision complex	CTRSM

Table 110: Data Types. (continued)	
Data Types	
Long-precision complex	ZTRSM

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL STRSM | DTRSM | CTRSM | ZTRSM (*side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*)
C and C++	strsm | dtrsm | ctrsm | ztrsm (*side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*);
CBLAS	cblas_strsm | cblas_dtrsm | cblas_ctrsm | cblas_ztrsm (*cblas_layout*, *cblas_side*, *cblas_uplo*, *cblas_transa*, *cblas_diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*);

On Entry

cblas_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

side

indicates whether the triangular matrix ***A*** is located to the left or right of rectangular matrix ***B*** in the system of equations, where:

If *side* = 'L', ***A*** is to the left of ***B***, resulting in solution 1, 2, or 5.

If *side* = 'R', ***A*** is to the right of ***B***, resulting in solution 3, 4, or 6.

Specified as: a single character. It must be 'L' or 'R'.

cblas_side

indicates whether matrix ***A*** is located to the left or right of rectangular matrix ***B*** in the equation used for this computation, where:

If *cblas_side* = CblasLeft, ***A*** is to the left of ***B***, resulting in solution 1, 2, or 5.

If *cblas_side* = CblasRight, ***A*** is to the right of ***B***, resulting in solution 3, 4, or 6.

Specified as: an object of enumerated type CBLAS_SIDE. It must be CblasLeft or CblasRight.

uplo

indicates whether matrix ***A*** is an upper or lower triangular matrix, where:

If *uplo* = 'U', ***A*** is an upper triangular matrix.

If *uplo* = 'L', ***A*** is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

cblas_uplo

indicates whether matrix ***A*** is an upper or lower triangular matrix, where:

If *cblas_uplo* = CblasUpper, ***A*** is an upper triangular matrix.

If *cblas_uplo* = CblasLower, ***A*** is a lower triangular matrix.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

transa
indicates the form of matrix ***A*** used in the system of equations, where:

If *transa* = 'N', ***A*** is used, resulting in solution 1 or 3.

If *transa* = 'T', $\boldsymbol{A}^T$ is used, resulting in solution 2 or 4.

If *transa* = 'C', $\boldsymbol{A}^H$ is used, resulting in solution 5 or 6.

Specified as: a single character. It must be 'N', 'T', or 'C'.

cblas_transa
indicates the form of matrix ***A*** to use in the computation, where:

If *cblas_transa* = CblasNoTrans, ***A*** is used, resulting in solution 1 or 3.

If *cblas_transa* = CblasTrans, $\boldsymbol{A}^T$ is used, resulting in solution 2 or 4.

If *cblas_transa* = CblasConjTrans, $\boldsymbol{A}^H$ is used, resulting in solution 5 or 6.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

cblas_diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = CblasUnit, ***A*** is a unit triangular matrix.

If *diag* = CblasNonUnit ***A*** is not a unit triangular matrix.

Specified as: an object of enumerated type CBLAS_DIAG. It must be CblasNonUnit or CblasUnit.

m
is the number of rows in rectangular matrix ***B***, and:

If *side* = 'L', *m* is the order of triangular matrix ***A***.

Specified as: an integer, where:

If *side* = 'L', $0 \leq m \leq lda$ and $m \leq ldb$.

If *side* = 'R', $0 \leq m \leq ldb$.

n
is the number of columns in rectangular matrix ***B***, and:

If *side* = 'R', *n* is the order of triangular matrix ***A***.

Specified as: an integer; $n \geq 0$, and:

If *side* = 'R', $n \leq lda$.

alpha
is the scalar α. Specified as: a number of the data type indicated in Table 110 on page 437.

a
is the triangular matrix ***A***, of which only the upper or lower triangular portion is used, where:

If *side* = 'L', ***A*** is order *m*.

If *side* = 'R', ***A*** is order *n*.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 110 on page 437, where:

If *side* = 'L', its size must be *lda* by (at least) *m*.

If *side* = 'R', it size must be *lda* by (at least) *n*.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0, and:

If *side* = 'L', *lda* ≥ *m*.

If *side* = 'R', *lda* ≥ *n*.

b

is the *m* by *n* rectangular matrix ***B***, which contains the right-hand sides of the triangular system to be solved.

Specified as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 110 on page 437.

ldb

is the leading dimension of the array specified for *b*.

Specified as: an integer; *ldb* > 0 and *ldb* ≥ *m*.

On Return

b

is the *m* by *n* matrix ***B***, containing the results of the computation.

Returned as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 110 on page 437.

Notes

1. These subroutines accept lowercase letters for the *transa*, *side*, *diag*, and *uplo* arguments.
2. For STRSM and DTRSM, if you specify 'C' for the *transa* argument, it is interpreted as though you specified 'T'.
3. Matrices ***A*** and ***B*** must have no common elements or results are unpredictable.
4. If matrix ***A*** is upper triangular (*uplo* = 'U'), these subroutines refer to only the upper triangular portion of the matrix. If matrix ***A*** is lower triangular, (*uplo* = 'L'), these subroutines refer to only the lower triangular portion of the matrix. The unreferenced elements are assumed to be zero.
5. The elements of the diagonal of a unit triangular matrix are always one, so you do not need to set these values. The ESSL subroutines always assume that the values in these positions are 1.0 for STRSM and DTRSM and (1.0, 0.0) for CTRSM and ZTRSM.
6. For a description of triangular matrices and how they are stored, see “Triangular Matrix” on page 88.

Function

These subroutines solve a triangular system of equations with multiple right-hand sides. The solution ***B*** may be any of the following, where ***A*** is a triangular matrix and ***B*** is a rectangular matrix:

1. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-1})\boldsymbol{B}$
2. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-T})\boldsymbol{B}$
3. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-1})$
4. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-T})$
5. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-H})\boldsymbol{B}$ (only for CTRSM and ZTRSM)

6. $\boldsymbol{B} \leftarrow \alpha \boldsymbol{B}(\boldsymbol{A}^{-H})$ (only for CTRSM and ZTRSM)

where:

α is a scalar.
B is an *m* by *n* rectangular matrix.
A is an upper or lower triangular matrix, where:
If *side* = 'L', it has order *m*, and equation 1, 2, or 5 is performed.
If *side* = 'R', it has order *n*, and equation 3, 4, or 6 is performed.

If *n* or *m* is 0, no computation is performed. See references [“42” on page 1255] and [“46” on page 1255].

Error conditions

Resource Errors

Unable to allocate internal work area.

Computational Errors

None

Note: If the triangular matrix ***A*** is singular, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. *m* < 0
3. *m* > *ldb*
4. *n* < 0
5. *lda*, *ldb* ≤ 0
6. *side* ≠ 'L' or 'R'
7. *side* = 'L' and *m* > *lda*
8. *side* = 'R' and *n* > *lda*
9. *cblas_side* ≠ CblasLeft or CblasRight
10. *cblas_side* = CblasLeft and *m* > *lda*
11. *cblas_side* = CblasRight and *n* > *lda*
12. *uplo* ≠ 'L' or 'U'
13. *cblas_uplo* ≠ CblasLower or CblasUpper
14. *transa* ≠ 'T', 'N', or 'C'
15. *cblas_transa* ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
16. *diag* ≠ 'N' or 'U'
17. *cblas_diag* ≠ CblasNonUnit or CblasUnit

Examples

Example 1

This example shows the solution $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-1})\boldsymbol{B}$, where ***A*** is a real 5 by 5 upper triangular matrix that is not unit triangular, and ***B*** is a real 5 by 3 rectangular matrix.

Call Statement and Input:

```
            SIDE  UPLO TRANSA  DIAG  M   N   ALPHA   A  LDA  B  LDB
             |     |     |      |    |   |     |     |   |   |   |
CALL STRSM( 'L' , 'U' , 'N'  , 'N' , 5 , 3 ,  1.0  , A , 7 , B , 6 )
```

```
         ┌                             ┐
         | 3.0  -1.0   2.0   2.0   1.0 |
         |  .   -2.0   4.0  -1.0   3.0 |
         |  .     .   -3.0   0.0   2.0 |
A    =   |  .     .     .    4.0  -2.0 |
         |  .     .     .     .    1.0 |
         |  .     .     .     .     .  |
         |  .     .     .     .     .  |
                                      ┘
```

```
         ┌                    ┐
         |   6.0  10.0   -2.0 |
         | -16.0  -1.0    6.0 |
B    =   |  -2.0   1.0   -4.0 |
         |  14.0   0.0  -14.0 |
         |  -1.0   2.0    1.0 |
         |    .     .      .  |
                             ┘
```

Output:

```
         ┌                ┐
         |  2.0  3.0   1.0 |
         |  5.0  5.0   4.0 |
B    =   |  0.0  1.0   2.0 |
         |  3.0  1.0  -3.0 |
         | -1.0  2.0   1.0 |
         |   .    .     .  |
                         ┘
```

Example 2

This example shows the solution ***B***←α(***A***$^{-T}$)***B***, where ***A*** is a real 5 by 5 upper triangular matrix that is not unit triangular, and ***B*** is a real 5 by 4 rectangular matrix.

Call Statement and Input:

```
            SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
             |     |     |      |    |   |     |     |   |   |   |
CALL STRSM( 'L' , 'U' , 'T'  , 'N' , 5 , 4 ,  1.0  , A , 7 , B , 6 )
```

```
         ┌                              ┐
         | -1.0  -4.0  -2.0   2.0   3.0 |
         |   .   -2.0   2.0   2.0   2.0 |
         |   .     .   -3.0  -1.0   4.0 |
A    =   |   .     .     .    1.0   0.0 |
         |   .     .     .     .   -2.0 |
         |   .     .     .     .     .  |
         |   .     .     .     .     .  |
                                       ┘
```

```
         ┌                        ┐
         | -1.0  -2.0   -3.0   -4.0 |
         |  2.0  -2.0  -14.0  -12.0 |
B    =   | 10.0   5.0   -8.0   -7.0 |
         | 14.0  15.0    1.0    8.0 |
         | -3.0   4.0    3.0   16.0 |
         |   .     .      .      .  |
                                  ┘
```

Output:

```
         ┌                      ┐
         |  1.0   2.0   3.0   4.0 |
         |  3.0   3.0  -1.0   2.0 |
B    =   | -2.0  -1.0   0.0   1.0 |
         |  4.0   4.0  -3.0  -3.0 |
        |  2.0   2.0   2.0   2.0 |
        |   .     .     .     .  |
                                ┘
```

Example 3

This example shows the solution ***B***←α***B***(***A***$^{-1}$), where ***A*** is a real 5 by 5 lower triangular matrix that is not unit triangular, and ***B*** is a real 3 by 5 rectangular matrix.

Call Statement and Input:

```
           SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
            |     |     |      |    |   |     |     |   |   |   |
CALL STRSM( 'R' , 'L' , 'N'  , 'N' , 3 , 5 ,  1.0  , A , 7 , B , 4 )
```

```
        ┌                           ┐
        | 2.0   .     .     .     .  |
        | 2.0  3.0    .     .     .  |
        | 2.0  1.0   1.0    .     .  |
A    =  | 0.0  3.0   0.0  -2.0    .  |
        | 2.0  4.0  -1.0   2.0  -1.0 |
        |  .    .     .     .     .  |
        |  .    .     .     .     .  |
                                    ┘
```

```
        ┌                              ┐
        | 10.0   4.0   0.0  0.0   1.0 |
B    =  | 10.0  14.0  -4.0  6.0  -3.0 |
        | -8.0   2.0  -5.0  4.0  -2.0 |
        |   .     .     .    .     .  |
                                     ┘
```

Output:

```
        ┌                              ┐
        |  3.0   4.0  -1.0  -1.0  -1.0 |
B    =  |  2.0   1.0  -1.0   0.0   3.0 |
        | -2.0  -1.0  -3.0   0.0   2.0 |
        |   .     .     .     .     .  |
                                      ┘
```

Example 4

This example shows the solution ***B***←α***B***(***A***$^{-1}$), where ***A*** is a real 6 by 6 upper triangular matrix that is unit triangular, and ***B*** is a real 1 by 6 rectangular matrix.

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal element.

Call Statement and Input:

```
           SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
            |     |     |      |    |   |     |     |   |   |   |
CALL STRSM( 'R' , 'U' , 'N'  , 'U' , 1 , 6 ,  1.0  , A , 7 , B , 2 )
```

```
        ┌                               ┐
        | .  2.0  -3.0  1.0   2.0   4.0 |
        | .   .    0.0  1.0   1.0  -2.0 |
        | .   .     .   4.0  -1.0   1.0 |
A    =  | .   .     .    .    0.0  -1.0 |
        | .   .     .    .     .    2.0 |
        | .   .     .    .     .     .  |
        | .   .     .    .     .     .  |
                                       ┘
```

```
        ┌                                 ┐
B    =  | 1.0  4.0  -2.0  10.0  2.0  -6.0 |
        |  .    .     .     .    .     .  |
                                         ┘
```

Output:

```
                ┌                                 ┐
B    =  | 1.0  2.0  1.0  3.0  -1.0  -2.0 |
        |  .    .    .    .     .     .  |
                                          ┘
```

Example 5

This example shows the solution ***B***←α***B***(***A***$^{-1}$), where ***A*** is a complex 5 by 5 lower triangular matrix that is not unit triangular, and ***B*** is a complex 3 by 5 rectangular matrix.

Call Statement and Input:

```
            SIDE  UPLO TRANSA  DIAG  M   N    ALPHA   A  LDA  B  LDB
             |     |     |      |    |   |      |     |   |   |   |
CALL CTRSM( 'R' , 'L' , 'N'  , 'N' , 3 , 5 ,  ALPHA , A , 7 , B , 4 )

ALPHA    =  (1.0, 0.0)
```

```
        ┌                                                             ┐
        | (2.0, -3.0)      .            .            .            .       |
        | (2.0, -4.0) (3.0, -1.0)       .            .            .       |
        | (2.0,  2.0) (1.0,  2.0)  (1.0,  1.0)       .            .       |
A    =  | (0.0,  0.0) (3.0, -1.0)  (0.0, -1.0) (-2.0,  1.0)       .       |
        | (2.0,  2.0) (4.0,  0.0) (-1.0,  2.0)  (2.0, -4.0) (-1.0, -4.0) |
        |     .            .            .            .            .       |
        |     .            .            .            .            .       |
                                                                      ┘
```

```
        ┌                                                                       ┐
        | (22.0, -41.0)  (7.0, -26.0)  (9.0, 0.0)  (-15.0, -3.0)  (-15.0, 8.0) |
B    =  | (29.0, -18.0) (24.0, -10.0)  (9.0, 6.0) (-12.0, -24.0) (-19.0, -8.0) |
        |  (-15.0, 2.0) (-3.0, -21.0) (-2.0, 4.0)  (-4.0, -12.0) (-10.0, -6.0) |
        |       .              .            .             .              .       |
                                                                             ┘
```

Output:

```
        ┌                                                                 ┐
        |  (3.0, 0.0)   (4.0, 0.0) (-1.0, -2.0) (-1.0, -1.0) (-1.0, -4.0) |
B    =  | (2.0, -1.0)   (1.0, 2.0) (-1.0, -3.0)   (0.0, 2.0)  (3.0, -4.0) |
        | (-2.0, 1.0) (-1.0, -3.0)  (-3.0, 1.0)   (0.0, 0.0)  (2.0, -2.0) |
        |     .            .            .              .            .     |
                                                                      ┘
```

Example 6

This example shows the solution ***B***←α(***A***$^{-H}$)***B***, where ***A*** is a complex 5 by 5 upper triangular matrix that is not unit triangular, and ***B*** is a complex 5 by 1 rectangular matrix.

Call Statement and Input:

```
            SIDE  UPLO TRANSA  DIAG  M   N    ALPHA   A  LDA  B  LDB
             |     |     |      |    |   |      |     |   |   |   |
CALL CTRSM( 'L' , 'U' , 'C'  , 'N' , 5 , 1 ,  ALPHA , A , 6 , B , 6 )

ALPHA    =  (1.0, 0.0)
```

```
        ┌                                                             ┐
        | (-4.0, 1.0) (4.0, -3.0)  (-1.0, 3.0)   (0.0, 0.0)  (-1.0, 0.0) |
        |      .      (-2.0, 0.0) (-3.0, -1.0) (-2.0, -1.0)   (4.0, 3.0) |
A    =  |      .           .       (-5.0, 3.0) (-3.0, -3.0) (-5.0, -5.0) |
        |      .           .            .       (4.0, -4.0)   (2.0, 0.0) |
        |      .           .            .            .       (2.0, -1.0) |
        |      .           .            .            .            .      |
                                                                      ┘
```

```
        ┌               ┐
        | (-8.0, -19.0) |
        |  (8.0,  21.0) |
B    =  | (44.0,  -8.0) |
        | (13.0,  -7.0) |
        | (19.0,   2.0) |
        |       .       |
                        ┘
```

Output:

```
        ┌               ┐
        |  (3.0, 4.0) |
        | (-4.0, 2.0) |
B    =  | (-5.0, 0.0) |
        |  (1.0, 3.0) |
        |  (3.0, 1.0) |
        |      .      |
                      ┘
```

SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK (Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)

Purpose

These subroutines compute one of the following rank-k updates, where matrix ***C*** is stored in either upper or lower storage mode. SSYRK, DSYRK, CSYRK, and ZSYRK use the scalars α and β, real or complex matrix ***A*** or its transpose, and real or complex symmetric matrix ***C*** to compute:

1. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{A}^{T}+\beta \boldsymbol{C}$
2. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{T}\boldsymbol{A}+\beta \boldsymbol{C}$

CHERK and ZHERK use the scalars α and β, complex matrix ***A*** or its complex conjugate transpose, and complex Hermitian matrix ***C*** to compute:

1. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{A}^{H}+\beta \boldsymbol{C}$
2. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{H}\boldsymbol{A}+\beta \boldsymbol{C}$

Table 111: Data Types.

Data Types		
A, ***C***	α, β	**Subprogram**
Short-precision real	Short-precision real	SSYRK
Long-precision real	Long-precision real	DSYRK
Short-precision complex	Short-precision complex	CSYRK
Long-precision complex	Long-precision complex	ZSYRK
Short-precision complex	Short-precision real	CHERK
Long-precision complex	Long-precision real	ZHERK

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SSYRK | DSYRK | CSYRK | ZSYRK | CHERK | ZHERK (*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *lda*, *beta*, *c*, *ldc*)
C and C++	ssyrk | dsyrk | csyrk | zsyrk | cherk | zherk (*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *lda*, *beta*, *c*, *ldc*);

Language	Syntax
CBLAS	cblas_ssyrk | cblas_dsyrk | cblas_csyrk | cblas_zsyrk | cblas_cherk | cblas_zherk (*cblas_layout*, *cblas_uplo*, *cblas_trans*, *n*, *k*, *alpha*, *a*, *lda*, *beta*, *c*, *ldc*);

On Entry

cblas_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

uplo

indicates the storage mode used for matrix ***C***, where:

If *uplo* = 'U', ***C*** is stored in upper storage mode.

If *uplo* = 'L', ***C*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

cblas_uplo

indicates the storage mode used for matrix ***A***, where:

If *cblas_uplo* = CblasUpper, ***A*** is stored in upper storage mode.

If *cblas_uplo* = CblasLower, ***A*** is stored in lower storage mode.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

trans

indicates the form of matrix ***A*** to use in the computation, where:

If *trans* = 'N', ***A*** is used, resulting in equation 1 or 3.

If *trans* = 'T', $\boldsymbol{A}^T$ is used, resulting in equation 2.

If *trans* = 'C', $\boldsymbol{A}^H$ is used, resulting in equation 4.

Specified as: a single character, where:

For SSYRK and DSYRK, it must be 'N', 'T', or 'C'.

For CSYRK and ZSYRK, it must be 'N' or 'T'.

For CHERK and ZHERK, it must be 'N' or 'C'.

cblas_trans

indicates the form of matrix ***A*** to use in the computation, where:

If *cblas_trans* = CblasNoTrans, ***A*** is used, resulting in equation 1 or 3.

If *cblas_trans* = CblasTrans, $\boldsymbol{A}^T$ is used, resulting in equation 2.

If *cblas_trans* = CblasConjTrans, $\boldsymbol{A}^H$ is used, resulting in equation 4.

Specified as: an object of enumerated type CBLAS_TRANSPOSE, where:

For SSYRK and DSYRK, it must be CblasNoTrans, CblasTrans, or CblasConjTrans.

For CSYRK and ZSYRK, it must be CblasNoTrans or CblasTrans.

For CHERK and ZHERK, it must be CblasNoTrans or CblasConjTrans.

n

is the order of matrix ***C***.

Specified as: an integer; $0 \leq n \leq ldc$ and:

If *trans* = 'N', then $n \leq lda$.

k

has the following meaning, where:

If *trans* = 'N', it is the number of columns in matrix ***A***.

If *trans* = 'T' or 'C', it is the number of rows in matrix ***A***.

Specified as: an integer; $k \geq 0$ and:

If *trans* = 'T' or 'C', then $k \leq lda$.

alpha

is the scalar α.

Specified as: a number of the data type indicated in Table 111 on page 445.

a

is the rectangular matrix ***A***, where:

If *trans* = 'N', ***A*** is *n* by *k*.

If *trans* = 'T' or 'C', ***A*** is *k* by *n*.

Note: No data should be moved to form $\boldsymbol{A}^T$ or $\boldsymbol{A}^H$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 111 on page 445, where:

If *trans* = 'N', its size must be *lda* by (at least) *k*.

If *trans* = 'T' or 'C', its size must be *lda* by (at least) *n*.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and:

If *trans* = 'N', $lda \geq n$.

If *trans* = 'T' or 'C', $lda \geq k$.

beta

is the scalar β.

Specified as: a number of the data type indicated in Table 111 on page 445.

c

is matrix ***C*** of order *n*, which is real symmetric, complex symmetric, or complex Hermitian, where:

If *uplo* = 'U', ***C*** is stored in upper storage mode.

If *uplo* = 'L', ***C*** is stored in lower storage mode.

Specified as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 111 on page 445.

ldc

is the leading dimension of the array specified for *c*.

Specified as: an integer; $ldc > 0$ and $ldc \geq n$.

On Return

c

is matrix ***C*** of order *n*, which is real symmetric, complex symmetric, or complex Hermitian, containing the results of the computation, where:

If *uplo* = 'U', ***C*** is stored in upper storage mode.

If *uplo* = 'L', ***C*** is stored in lower storage mode.

Returned as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 111 on page 445.

Notes

1. These subroutines accept lowercase letters for the *uplo* and *trans* arguments.
2. For SSYRK and DSYRK, if you specify 'C' for the *trans* argument, it is interpreted as though you specified 'T'.
3. Matrices ***A*** and ***C*** must have no common elements; otherwise, results are unpredictable.
4. The imaginary parts of the diagonal elements of a complex Hermitian matrix ***C*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero, except when β is one and α or *k* is zero, in which case no computation is performed.
5. For a description of how symmetric matrices are stored in upper and lower storage mode, see "Symmetric Matrix" on page 80. For a description of how complex Hermitian matrices are stored in upper and lower storage mode, see "Complex Hermitian Matrix" on page 84.

Function

These subroutines can perform the following rank-k updates. For SSYRK and DSYRK, matrix ***C*** is real symmetric. For CSYRK and ZSYRK, matrix ***C*** is complex symmetric. They perform:

1. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{A}^T + \beta \boldsymbol{C}$
2. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^T\boldsymbol{A} + \beta \boldsymbol{C}$

For CHERK and ZHERK, matrix ***C*** is complex Hermitian. They perform:

1. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{A}^H + \beta \boldsymbol{C}$
2. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^H\boldsymbol{A} + \beta \boldsymbol{C}$

where:

α and β are scalars.

A is a rectangular matrix, which is *n* by *k* for equations 1 and 3, and is *k* by *n* for equations 2 and 4.

C is a matrix of order *n* of the type indicated above, stored in upper or lower storage mode.

See references ["42" on page 1255] and ["48" on page 1255]. In the following two cases, no computation is performed:

- *n* is 0.
- β is one, and α is zero or *k* is zero.

Assuming the above conditions do not exist, if β is not one, and αis zero or *k* is zero, then β***C*** is returned.

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. *lda*, *ldc* ≤ 0
3. *ldc* < *n*
4. *k*, *n* < 0
5. *uplo* ≠ 'U' or 'L'

6. *cblas_uplo* ≠ CblasLower or CblasUpper
7. *trans* ≠ 'N', 'T', or 'C' for SSYRK and DSYRK
8. *trans* ≠ 'N' or 'T' for CSYRK and ZSYRK
9. *trans* ≠ 'N' or 'C' for CHERK and ZHERK
10. *trans* = 'N' and *lda* < *n*
11. *trans* = 'T' or 'C' and *lda* < *k*
12. *cblas_trans* ≠ CblasNoTrans, CblasTrans, or CblasConjTrans for SSYRK and DSYRK
13. *cblas_trans* ≠ CblasNoTrans or CblasTrans for CSYRK and ZSYRK
14. *cblas_trans* ≠ CblasNoTrans or CblasConjTrans for CHERK and ZHERK
15. *cblas_trans* = CblasNoTrans and *lda* < *n*
16. *cblas_trans* = CblasTrans, or CblasConjTrans and *lda* < *k*

Examples

Example 1

This example shows the computation $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{A}^{T} + \beta \boldsymbol{C}$, where ***A*** is an 8 by 2 real rectangular matrix, and ***C*** is a real symmetric matrix of order 8, stored in upper storage mode.

Call Statement and Input:

```
             UPLO TRANS   N   K   ALPHA   A  LDA  BETA  C   LDC
              |     |     |   |     |     |   |    |    |    |
CALL SSYRK( 'U' , 'N' ,  8 , 2 ,  1.0  , A , 9 , 1.0 , C , 10 )
```

```
        ┌           ┐
        | 0.0   8.0 |
        | 1.0   9.0 |
        | 2.0  10.0 |
        | 3.0  11.0 |
A    =  | 4.0  12.0 |
        | 5.0  13.0 |
        | 6.0  14.0 |
        | 7.0  15.0 |
        |  .     .  |
                   ┘
```

```
        ┌                                                ┐
        | 0.0  1.0  3.0  6.0  10.0  15.0  21.0  28.0 |
        |  .   2.0  4.0  7.0  11.0  16.0  22.0  29.0 |
        |  .    .   5.0  8.0  12.0  17.0  23.0  30.0 |
        |  .    .    .   9.0  13.0  18.0  24.0  31.0 |
C    =  |  .    .    .    .   14.0  19.0  25.0  32.0 |
        |  .    .    .    .     .   20.0  26.0  33.0 |
        |  .    .    .    .     .     .   27.0  34.0 |
        |  .    .    .    .     .     .     .   35.0 |
        |  .    .    .    .     .     .     .     .  |
        |  .    .    .    .     .     .     .     .  |
                                                   ┘
```

Output:

```
        ┌                                                       ┐
        | 64.0  73.0   83.0   94.0  106.0  119.0  133.0  148.0 |
        |   .   84.0   96.0  109.0  123.0  138.0  154.0  171.0 |
        |   .     .   109.0  124.0  140.0  157.0  175.0  194.0 |
        |   .     .      .   139.0  157.0  176.0  196.0  217.0 |
C    =  |   .     .      .      .   174.0  195.0  217.0  240.0 |
        |   .     .      .      .      .   214.0  238.0  263.0 |
        |   .     .      .      .      .      .   259.0  286.0 |
        |   .     .      .      .      .      .      .   309.0 |
        |   .     .      .      .      .      .      .      .  |
        |   .     .      .      .      .      .      .      .  |
                                                          ┘
```

Example 2

This example shows the computation $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^T \boldsymbol{A} + \beta \boldsymbol{C}$, where ***A*** is a 3 by 8 real rectangular matrix, and ***C*** is a real symmetric matrix of order 8, stored in lower storage mode.

Call Statement and Input:

```
            UPLO TRANS   N   K   ALPHA   A  LDA  BETA  C  LDC
             |     |     |   |     |     |   |    |    |   |
CALL SSYRK( 'L' , 'T' ,  8 , 3 ,  1.0  , A , 4 , 1.0 , C , 8 )
```

```
         ┌                                              ┐
         | 0.0  3.0  6.0   9.0  12.0  15.0  18.0  21.0 |
A    =   | 1.0  4.0  7.0  10.0  13.0  16.0  19.0  22.0 |
         | 2.0  5.0  8.0  11.0  14.0  17.0  20.0  23.0 |
         |  .    .    .     .     .     .     .     .  |
                                                       ┘
```

```
         ┌                                               ┐
         | 0.0    .     .     .     .     .     .     .  |
         | 1.0   8.0    .     .     .     .     .     .  |
         | 2.0   9.0  15.0    .     .     .     .     .  |
C    =   | 3.0  10.0  16.0  21.0    .     .     .     .  |
         | 4.0  11.0  17.0  22.0  26.0    .     .     .  |
         | 5.0  12.0  18.0  23.0  27.0  30.0    .     .  |
         | 6.0  13.0  19.0  24.0  28.0  31.0  33.0    .  |
         | 7.0  14.0  20.0  25.0  29.0  32.0  34.0  35.0 |
                                                        ┘
```

Output:

```
         ┌                                                          ┐
         |  5.0     .      .      .      .       .       .       .  |
         | 15.0   58.0     .      .      .       .       .       .  |
         | 25.0   95.0  164.0     .      .       .       .       .  |
C    =   | 35.0  132.0  228.0  323.0     .       .       .       .  |
         | 45.0  169.0  292.0  414.0  535.0      .       .       .  |
         | 55.0  206.0  356.0  505.0  653.0   800.0      .       .  |
         | 65.0  243.0  420.0  596.0  771.0   945.0  1118.0      .  |
         | 75.0  280.0  484.0  687.0  889.0  1090.0  1290.0  1489.0 |
                                                                   ┘
```

Example 3

This example shows the computation $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A} \boldsymbol{A}^T + \beta \boldsymbol{C}$, where ***A*** is a 3 by 5 complex rectangular matrix, and ***C*** is a complex symmetric matrix of order 3, stored in upper storage mode.

Call Statement and Input:

```
            UPLO TRANS   N   K   ALPHA   A  LDA  BETA   C  LDC
             |     |     |   |     |     |   |    |     |   |
CALL CSYRK( 'U' , 'N' ,  3 , 5 , ALPHA , A , 3 , BETA , C , 4 )
```

```
ALPHA    =  (1.0, 1.0)
BETA     =  (1.0, 1.0)
```

```
         ┌                                                      ┐
         | (2.0, 0.0) (3.0, 2.0) (4.0, 1.0) (1.0, 7.0) (0.0, 0.0) |
A    =   | (3.0, 3.0) (8.0, 0.0) (2.0, 5.0) (2.0, 4.0) (1.0, 2.0) |
         | (1.0, 3.0) (2.0, 1.0) (6.0, 0.0) (3.0, 2.0) (2.0, 2.0) |
                                                                ┘
```

```
         ┌                                  ┐
         | (2.0, 1.0) (1.0, 9.0) (4.0, 5.0) |
C    =   |     .      (3.0, 1.0) (6.0, 7.0) |
         |     .          .      (8.0, 1.0) |
         |     .          .          .      |
                                           ┘
```

Output:

```
         ┌                                               ┐
         | (-57.0, 13.0) (-63.0, 79.0) (-24.0,  70.0) |
C    =   |       .       (-28.0, 90.0) (-55.0, 103.0) |
         |       .             .        (13.0,  75.0) |
         |       .             .              .       |
                                                      ┘
```

Example 4

This example shows the computation $C \leftarrow \alpha A^H A + \beta C$, where ***A*** is a 5 by 3 complex rectangular matrix, and ***C*** is a complex Hermitian matrix of order 3, stored in lower storage mode.

Note: The imaginary parts of the diagonal elements of a complex Hermitian matrix are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input:

```
            UPLO TRANS   N   K   ALPHA   A  LDA  BETA   C  LDC
             |     |     |   |     |     |   |    |     |   |
CALL CHERK( 'L' , 'C' ,  3 , 5 ,  1.0  , A , 5 , 1.0  , C , 4 )
```

```
         ┌                                  ┐
         | (2.0, 0.0) (3.0, 2.0) (4.0, 1.0) |
         | (3.0, 3.0) (8.0, 0.0) (2.0, 5.0) |
A    =   | (1.0, 3.0) (2.0, 1.0) (6.0, 0.0) |
         | (3.0, 3.0) (8.0, 0.0) (2.0, 5.0) |
         | (1.0, 9.0) (3.0, 0.0) (6.0, 7.0) |
                                            ┘
```

```
         ┌                                    ┐
         | (6.0,  . )       .           .     |
C    =   | (3.0, 4.0)  (10.0,  . )      .     |
         | (9.0, 1.0)  (12.0, 2.0)  (3.0, . ) |
         |     .            .           .     |
                                              ┘
```

Output:

```
         ┌                                                ┐
         | (138.0,  0.0)          .               .       |
C    =   |  (65.0, 80.0)  (165.0,   0.0)          .       |
         | (134.0, 46.0)   (88.0, -88.0)  (199.0, 0.0)    |
         |       .                .               .       |
                                                          ┘
```

SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K (Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)

Purpose

These subroutines compute one of the following rank-2k updates, where matrix ***C*** is stored in upper or lower storage mode. SSYR2K, DSYR2K, CSYR2K, and ZSYR2K use the scalars α and β, real or complex matrices ***A*** and ***B*** or their transposes, and real or complex symmetric matrix ***C*** to compute:

1. $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$
2. $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$

CHER2K and ZHER2K use the scalars α and β, complex matrices ***A*** and ***B*** or their complex conjugate transposes, and complex Hermitian matrix ***C*** to compute:

3. $C \leftarrow \alpha A B^{H} + \overline{\alpha} B A^{H} + \beta C$

4. $C \leftarrow \alpha A^{H} B + \overline{\alpha} B^{H} A + \beta C$

Table 112: Data Types.

Data Types		
A, ***B***, ***C***, α	β	**Subprogram**
Short-precision real	Short-precision real	SSYR2K
Long-precision real	Long-precision real	DSYR2K
Short-precision complex	Short-precision complex	CSYR2K
Long-precision complex	Long-precision complex	ZSYR2K
Short-precision complex	Short-precision real	CHER2K
Long-precision complex	Long-precision real	ZHER2K

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SSYR2K | DSYR2K | CSYR2K | ZSYR2K | CHER2K | ZHER2K (*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*)
C and C++	ssyr2k | dsyr2k | csyr2k | zsyr2k | cher2k | zher2k (*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*);
CBLAS	cblas_ssyr2k | cblas_dsyr2k | cblas_csyr2k | cblas_zsyr2k | cblas_cher2k | cblas_zher2k (*cblas_layout*, *cblas_uplo*, *cblas_trans*, *n*, *k*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*);

On Entry

cblas_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *cblas_layout* = CblasRowMajor, the matrices are stored in row major order.
- If *cblas_layout* = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

uplo

indicates the storage mode used for matrix ***C***, where:

If *uplo* = 'U', ***C*** is stored in upper storage mode.

If *uplo* = 'L', ***C*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

cblas_uplo

indicates the storage mode used for matrix ***A***, where:

If *cblas_uplo* = CblasUpper, ***A*** is stored in upper storage mode.

If *cblas_uplo* = CblasLower, ***A*** is stored in lower storage mode.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

trans

indicates the form of matrices ***A*** and ***B*** to use in the computation, where:

If *trans* = 'N', ***A*** and ***B*** are used, resulting in equation 1 or 3.

If *trans* = 'T', $\boldsymbol{A}^T$ and $\boldsymbol{B}^T$ are used, resulting in equation 2.

If *trans* = 'C', $\boldsymbol{A}^H$ and $\boldsymbol{B}^H$ are used, resulting in equation 4.

Specified as: a single character, where:

For SSYR2K and DSYR2K, it must be 'N', 'T', or 'C'.

For CSYR2K and ZSYR2K, it must be 'N' or 'T'.

For CHER2K and ZHER2K, it must be 'N' or 'C'.

cblas_trans

indicates the form of matrix ***A*** to use in the computation, where:

If *cblas_trans* = CblasNoTrans, ***A*** is used, resulting in equation 1 or 3.

If *cblas_trans* = CblasTrans, $\boldsymbol{A}^T$ is used, resulting in equation 2.

If *cblas_trans* = CblasConjTrans, $\boldsymbol{A}^H$ is used, resulting in equation 4.

Specified as: an object of enumerated type CBLAS_TRANSPOSE, where:

For SSYR2K and DSYR2K, it must be CblasNoTrans, CblasTrans, or CblasConjTrans.

For CSYR2K and ZSYR2K, it must be CblasNoTrans or CblasTrans.

For CHER2K and ZHER2K, it must be CblasNoTrans or CblasConjTrans.

n

is the order of matrix ***C***.

Specified as: an integer; $0 \le n \le ldc$ and:

If *trans* = 'N', then $n \le lda$ and $n \le ldb$.

k

has the following meaning, where:

If *trans* = 'N', it is the number of columns in matrices ***A*** and ***B***.

If *trans* = 'T' or 'C', it is the number of rows in matrices ***A*** and ***B***.

Specified as: an integer; $k \ge 0$ and:

If *trans* = 'T' or 'C', then $k \le lda$ and $k \le ldb$.

alpha

is the scalar α.

Specified as: a number of the data type indicated in Table 112 on page 452.

a

is the rectangular matrix ***A***, where:

If *trans* = 'N', ***A*** is *n* by *k*.

If *trans* = 'T' or 'C', ***A*** is *k* by *n*.

Note: No data should be moved to form $\boldsymbol{A}^T$ or $\boldsymbol{A}^H$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 112 on page 452, where:

If *trans* = 'N', its size must be *lda* by (at least) *k*.

If *trans* = 'T' or 'C', its size must be *lda* by (at least) *n*.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and:

If *trans* = 'N', *lda* ≥ *n*.

If *trans* = 'T' or 'C', *lda* ≥ *k*.

b

is the rectangular matrix ***B***, where:

If *trans* = 'N', ***B*** is *n* by *k*.

If *trans* = 'T' or 'C', ***B*** is *k* by *n*.

Note: No data should be moved to form $\boldsymbol{B}^T$ or $\boldsymbol{B}^H$; that is, the matrix ***B*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 112 on page 452, where:

If *trans* = 'N', its size must be *ldb* by (at least) *k*.

If *trans* = 'T' or 'C', its size must be *ldb* by (at least) *n*.

ldb

is the leading dimension of the array specified for *b*.

Specified as: an integer; *ldb* > 0 and:

If *trans* = 'N', *ldb* ≥ *n*.

If *trans* = 'T' or 'C', *ldb* ≥ *k*.

beta

is the scalar β.

Specified as: a number of the data type indicated in Table 112 on page 452.

c

is matrix ***C*** of order *n*, which is real symmetric, complex symmetric, or complex Hermitian, where:

If *uplo* = 'U', ***C*** is stored in upper storage mode.

If *uplo* = 'L', ***C*** is stored in lower storage mode.

Specified as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 112 on page 452.

ldc

is the leading dimension of the array specified for *c*.

Specified as: an integer; *ldc* > 0 and *ldc* ≥ *n*.

On Return

c

is matrix ***C*** of order *n*, which is real symmetric, complex symmetric, or complex Hermitian, containing the results of the computation, where:

If *uplo* = 'U', ***C*** is stored in upper storage mode.

If *uplo* = 'L', ***C*** is stored in lower storage mode.

Returned as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 112 on page 452.

Notes

1. These subroutines accept lowercase letters for the *uplo* and *trans* arguments.
2. For SSYR2K and DSYR2K, if you specify 'C' for the *trans* argument, it is interpreted as though you specified 'T'.
3. Matrices ***A*** and ***B*** must have no common elements with matrix ***C***; otherwise, results are unpredictable.
4. The imaginary parts of the diagonal elements of a complex Hermitian matrix ***C*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero, except when β is one and α or *k* is zero, in which case no computation is performed.
5. For a description of how symmetric matrices are stored in upper and lower storage mode, see "Symmetric Matrix" on page 80. For a description of how complex Hermitian matrices are stored in upper and lower storage mode, see "Complex Hermitian Matrix" on page 84.

Function

These subroutines can perform the following rank-2k updates. For SSYR2K and DSYR2K, matrix ***C*** is real symmetric. For CSYR2K and ZSYR2K, matrix ***C*** is complex symmetric. They perform:

1. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B}^{T} + \alpha \boldsymbol{B}\boldsymbol{A}^{T} + \beta \boldsymbol{C}$
2. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{T}\boldsymbol{B} + \alpha \boldsymbol{B}^{T}\boldsymbol{A} + \beta \boldsymbol{C}$

For CHER2K and ZHER2K, matrix ***C*** is complex Hermitian. They perform:

3. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B}^{H} + \overline{\alpha} \boldsymbol{B}\boldsymbol{A}^{H} + \beta \boldsymbol{C}$
4. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{H}\boldsymbol{B} + \overline{\alpha} \boldsymbol{B}^{H}\boldsymbol{A} + \beta \boldsymbol{C}$

where:

α and β are scalars.

A and ***B*** are rectangular matrices, which are *n* by *k* for equations 1 and 3, and are *k* by *n* for equations 2 and 4.

C is a matrix of order *n* of the type indicated above, stored in upper or lower storage mode.

See references ["42" on page 1255], ["48" on page 1255], and ["86" on page 1257]. In the following two cases, no computation is performed:

- *n* is 0.
- β is one, and α is zero or *k* is zero.

Assuming the above conditions do not exist, if β is not one, and α is zero or *k* is zero, then β***C*** is returned.

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors

1. *cblas_layout* ≠ CblasRowMajor or CblasColMajor
2. *lda*, *ldb*, *ldc* ≤ 0

3. *ldc* < *n*
4. *k*, *n* < 0
5. *uplo* ≠ 'U' or 'L'
6. *cblas_uplo* ≠ CblasLower or CblasUpper
7. *trans* ≠ 'N', 'T', or 'C' for SSYR2K and DSYR2K
8. *trans* ≠ 'N' or 'T' for CSYR2K and ZSYR2K
9. *trans* ≠ 'N' or 'C' for CHER2K and ZHER2K
10. *trans* = 'N' and *lda* < *n*
11. *trans* = 'T' or 'C' and *lda* < *k*
12. *trans* = 'N' and *ldb* < *n*
13. *trans* = 'T' or 'C' and *ldb* < *k*
14. *cblas_transa* ≠ CblasNoTrans, CblasTrans, or CblasConjTrans for SSYR2K and DSYR2K
15. *cblas_transa* ≠ CblasNoTrans or CblasTrans for CSYR2K and ZSYR2K
16. *cblas_transa* ≠ CblasNoTrans or CblasConjTrans for CHER2K and ZHER2K
17. *cblas_transa* = CblasNoTrans and *lda* < *n*
18. *cblas_transa* = CblasTrans, or CblasConjTrans and *lda* < *k*
19. *cblas_trans* = CblasNoTrans and *ldb* < *n*
20. *cblas_trans* = CblasNoTrans or CblasConjTrans and *ldb* < *k*

Examples

Example 1

This example shows the computation $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B}^T + \alpha \boldsymbol{B}\boldsymbol{A}^T + \beta \boldsymbol{C}$, where ***A*** and ***B*** are 8 by 2 real rectangular matrices, and ***C*** is a real symmetric matrix of order 8, stored in upper storage mode.

Call Statement and Input:

```
              UPLO TRANS   N   K   ALPHA   A  LDA  B  LDB  BETA  C   LDC
               |     |     |   |     |     |   |   |   |    |    |    |
CALL SSYR2K( 'U' , 'N' ,  8 , 2 ,  1.0  , A , 9 , B , 8 , 1.0 , C , 10 )
```

```
         ┌           ┐
         | 0.0   8.0 |
         | 1.0   9.0 |
         | 2.0  10.0 |
         | 3.0  11.0 |
A    =   | 4.0  12.0 |
         | 5.0  13.0 |
         | 6.0  14.0 |
         | 7.0  15.0 |
         |  .     .  |
                    ┘
```

```
         ┌           ┐
         | 15.0  7.0 |
         | 14.0  6.0 |
         | 13.0  5.0 |
B    =   | 12.0  4.0 |
         | 11.0  3.0 |
         | 10.0  2.0 |
         |  9.0  1.0 |
         |  8.0  0.0 |
                    ┘
```

```
         ┌                                                   ┐
         | 0.0  1.0  3.0  6.0  10.0  15.0  21.0  28.0 |
         |  .   2.0  4.0  7.0  11.0  16.0  22.0  29.0 |
         |  .    .   5.0  8.0  12.0  17.0  23.0  30.0 |
         |  .    .    .   9.0  13.0  18.0  24.0  31.0 |
C    =   |  .    .    .    .   14.0  19.0  25.0  32.0 |
        |  .    .    .    .     .   20.0  26.0  33.0 |
        |  .    .    .    .     .     .   27.0  34.0 |
        |  .    .    .    .     .     .     .   35.0 |
        |  .    .    .    .     .     .     .     .  |
        |  .    .    .    .     .     .     .     .  |
                                                   ┘
```

Output:

```
        ┌                                                   ┐
        | 112.0  127.0  143.0  160.0  178.0  197.0  217.0  238.0 |
        |    .   138.0  150.0  163.0  177.0  192.0  208.0  225.0 |
        |    .      .   157.0  166.0  176.0  187.0  199.0  212.0 |
        |    .      .      .   169.0  175.0  182.0  190.0  199.0 |
C    =  |    .      .      .      .   174.0  177.0  181.0  186.0 |
        |    .      .      .      .      .   172.0  172.0  173.0 |
        |    .      .      .      .      .      .   163.0  160.0 |
        |    .      .      .      .      .      .      .   147.0 |
        |    .      .      .      .      .      .      .      .  |
        |    .      .      .      .      .      .      .      .  |
                                                             ┘
```

Example 2

This example shows the computation ***C***←α***A***^{T}***B***+α***B***^{T}***A***+β***C***, where ***A*** and ***B*** are 3 by 8 real rectangular matrices, and ***C*** is a real symmetric matrix of order 8, stored in lower storage mode.

Call Statement and Input:

```
             UPLO TRANS   N   K   ALPHA   A  LDA  B  LDB  BETA  C  LDC
              |     |     |   |     |     |   |   |   |    |    |   |
CALL SSYR2K( 'L' , 'T' ,  8 , 3 ,  1.0  , A , 4 , B , 5 , 1.0 , C , 8 )
```

```
        ┌                                        ┐
        | 0.0  3.0  6.0   9.0  12.0  15.0  18.0  21.0 |
A    =  | 1.0  4.0  7.0  10.0  13.0  16.0  19.0  22.0 |
        | 2.0  5.0  8.0  11.0  14.0  17.0  20.0  23.0 |
        |  .    .    .     .     .     .     .     .  |
                                                    ┘
```

```
        ┌                                    ┐
        | 1.0  2.0  3.0  4.0  5.0  6.0  7.0   8.0 |
        | 2.0  3.0  4.0  5.0  6.0  7.0  8.0   9.0 |
B    =  | 3.0  4.0  5.0  6.0  7.0  8.0  9.0  10.0 |
        |  .    .    .    .    .    .    .     .  |
        |  .    .    .    .    .    .    .     .  |
                                                ┘
```

```
        ┌                                        ┐
        | 0.0    .     .     .     .     .     .     .  |
        | 1.0   8.0    .     .     .     .     .     .  |
        | 2.0   9.0  15.0    .     .     .     .     .  |
C    =  | 3.0  10.0  16.0  21.0    .     .     .     .  |
        | 4.0  11.0  17.0  22.0  26.0    .     .     .  |
        | 5.0  12.0  18.0  23.0  27.0  30.0    .     .  |
        | 6.0  13.0  19.0  24.0  28.0  31.0  33.0    .  |
        | 7.0  14.0  20.0  25.0  29.0  32.0  34.0  35.0 |
                                                   ┘
```

Output:

```
        ┌                                                ┐
        |  16.0    .     .     .     .     .      .      .  |
        |  38.0  84.0    .     .     .     .      .      .  |
        |  60.0 124.0 187.0    .     .     .      .      .  |
C    =  |  82.0 164.0 245.0 325.0    .     .      .      .  |
        | 104.0 204.0 303.0 401.0 498.0    .      .      .  |
        | 126.0 244.0 361.0 477.0 592.0 706.0     .      .  |
        | 148.0 284.0 419.0 553.0 686.0 818.0  949.0     .  |
        | 170.0 324.0 477.0 629.0 780.0 930.0 1079.0 1227.0 |
                                                        ┘
```

Example 3

This example shows the computation $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B}^T + \alpha \boldsymbol{B}\boldsymbol{A}^T + \beta \boldsymbol{C}$, where ***A*** and ***B*** are 3 by 5 complex rectangular matrices, and ***C*** is a complex symmetric matrix of order 3, stored in lower storage mode.

Call Statement and Input:

```
              UPLO TRANS   N   K   ALPHA   A  LDA  B  LDB  BETA   C  LDC
               |     |     |   |     |     |   |   |   |    |     |   |
CALL CSYR2K( 'L' , 'N' ,  3 , 5 , ALPHA , A , 3 , B , 3 , BETA , C , 4 )
```

```
ALPHA    =  (1.0, 1.0)

BETA     =  (1.0, 1.0)
```

```
        ┌                                                  ┐
        | (2.0, 5.0) (3.0, 2.0) (4.0, 1.0) (1.0, 7.0) (0.0, 0.0) |
A    =  | (3.0, 3.0) (8.0, 5.0) (2.0, 5.0) (2.0, 4.0) (1.0, 2.0) |
        | (1.0, 3.0) (2.0, 1.0) (6.0, 5.0) (3.0, 2.0) (2.0, 2.0) |
                                                          ┘
```

```
        ┌                                                  ┐
        | (1.0, 5.0) (6.0, 2.0) (3.0, 1.0) (2.0, 0.0) (1.0, 0.0) |
B    =  | (2.0, 4.0) (7.0, 5.0) (2.0, 5.0) (2.0, 4.0) (0.0, 0.0) |
        | (3.0, 5.0) (8.0, 1.0) (1.0, 5.0) (1.0, 0.0) (1.0, 1.0) |
                                                          ┘
```

```
        ┌                                     ┐
        | (2.0, 3.0)      .          .        |
C    =  | (1.0, 9.0) (3.0, 3.0)      .        |
        | (4.0, 5.0) (6.0, 7.0) (8.0, 3.0)    |
        |      .          .          .        |
                                            ┘
```

Output:

```
        ┌                                                      ┐
        | (-101.0, 121.0)          .                .          |
C    =  | (-182.0, 192.0) (-274.0, 248.0)           .          |
        |  (-98.0, 146.0) (-163.0, 205.0) (-151.0, 115.0)      |
        |        .                .                .           |
                                                          ┘
```

Example 4

This example shows the computation:

$$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^H \boldsymbol{B} + \overline{\alpha} \boldsymbol{B}^H \boldsymbol{A} + \beta \boldsymbol{C}$$

where ***A*** and ***B*** are 5 by 3 complex rectangular matrices, and ***C*** is a complex Hermitian matrix of order 3, stored in upper storage mode.

Note: The imaginary parts of the diagonal elements of a complex Hermitian matrix are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input:

```
              UPLO TRANS   N   K   ALPHA   A  LDA  B  LDB  BETA   C  LDC
               |     |     |   |     |     |   |   |   |    |     |   |
CALL CHER2K( 'U' , 'C' ,  3 , 5 , ALPHA , A , 5 , B , 5 , 1.0  , C , 4 )

ALPHA    =  (1.0, 1.0)
```

```
        ┌                                     ┐
        | (2.0, 0.0) (3.0, 2.0) (4.0, 1.0)    |
        | (3.0, 3.0) (8.0, 0.0) (2.0, 5.0)    |
A    =  | (1.0, 3.0) (2.0, 1.0) (6.0, 0.0)    |
        | (3.0, 3.0) (8.0, 0.0) (2.0, 5.0)    |
        | (1.0, 9.0) (3.0, 0.0) (6.0, 7.0) |
                                          ┘
```

```
        ┌                                  ┐
        | (4.0, 5.0) (6.0, 7.0) (8.0, 0.0) |
        | (1.0, 9.0) (3.0, 0.0) (6.0, 7.0) |
B    =  | (3.0, 3.0) (8.0, 0.0) (2.0, 5.0) |
        | (1.0, 3.0) (2.0, 1.0) (6.0, 0.0) |
        | (2.0, 0.0) (3.0, 2.0) (4.0, 1.0) |
                                          ┘
```

```
        ┌                                    ┐
        | (6.0,  . )  (3.0, 4.0)  (9.0, 1.0) |
C    =  |     .      (10.0,  . ) (12.0, 2.0) |
        |     .           .       (3.0,  . ) |
        |     .           .           .      |
                                            ┘
```

Output:

```
        ┌                                                ┐
        | (102.0, 0.0)  (56.0, -143.0) (244.0, -96.0) |
C    =  |       .      (174.0,    0.0) (238.0,  78.0) |
        |       .            .         (363.0,   0.0) |
        |       .            .               .        |
                                                     ┘
```

SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI and ZGECMI (General Matrix Transpose or Conjugate Transpose [In-Place])

Purpose

Subroutines SGETMI, DGETMI, CGETMI, and ZGETMI perform a transpose of an *n* by *n* matrix ***A*** in place—that is, in matrix ***A***:

$\boldsymbol{A} \leftarrow \boldsymbol{A}^{T}$

Subroutines CGECMI and ZGECMI perform a conjugate transpose of an *n* by *n* matrix ***A*** in place—that is, in matrix ***A***:

$\boldsymbol{A} \leftarrow \boldsymbol{A}^{H}$

Table 113: Data Types.

Data Types	
A	**Subroutine**
Short-precision real	SGETMI
Long-precision real	DGETMI
Short-precision complex	CGETMI CGECMI
Long-precision complex	ZGETMI ZGECMI

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SGETMI | DGETMI | CGETMI | ZGETMI | CGECMI | ZGECMI (*a*, *lda*, *n*)
C and C++	sgetmi | dgetmi | cgetmi | zgetmi | cgecmi | zgecmi (*a*, *lda*, *n*);

On Entry

a

is the matrix ***A*** having *n* rows and *n* columns.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 113 on page 459.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

n

is the number of rows and columns in matrix ***A***.

Specified as: an integer; $n \geq 0$.

On Return

a

is the *n* by *n* matrix, containing the results of the operation.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 113 on page 459.

Function

Subroutines SGETMI, DGETMI, CGETMI, and ZGETMI perform a transpose of matrix ***A*** in place. For matrix ***A*** with elements a_{ij}, where $i, j = 1, n$, the in-place transpose is expressed as:

$a_{ji} = a_{ij}$ for $i, j = 1, n$

Subroutines CGECMI and ZGECMI perform a conjugate transpose of matrix ***A*** in place. For matrix ***A*** with elements a_{ij}, where $i, j = 1, n$, the in-place conjugate transpose is expressed as:

$a_{ji} = \bar{a}_{ij}$ for $i, j = 1, n$

If *n* is 0, no computation is performed.

Error conditions

Computational Errors

None

Input-Argument Errors

1. $n < 0$ or $n > lda$
2. $lda \leq 0$

Examples

Example 1

This example shows an in-place matrix transpose of matrix ***A*** having 5 rows and 5 columns.

Call Statement and Input:

```
                A     LDA   N
                |      |    |
CALL SGETMI( A(2,3) , 10 , 5 )
```

```
        ┌                                    ┐
        | .   .   .     .     .     .     .  |
        | .   .  1.0   6.0  11.0  16.0  21.0 |
        | .   .  2.0   7.0  12.0  17.0  22.0 |
        | .   .  3.0   8.0  13.0  18.0  23.0 |
A    =  | .   .  4.0   9.0  14.0  19.0  24.0 |
        | .   .  5.0  10.0  15.0  20.0  25.0 |
        | .   .   .     .     .     .     .  |
        | .   .   .     .     .     .     .  |
        | .   .   .     .     .     .     .  |
        | .   .   .     .     .     .     .  |
                                            ┘
```

Output:

```
        ┌                                     ┐
        | .   .    .     .     .     .     .  |
        | .   .   1.0   2.0   3.0   4.0   5.0 |
        | .   .   6.0   7.0   8.0   9.0  10.0 |
        | .   .  11.0  12.0  13.0  14.0  15.0 |
A    =  | .   .  16.0  17.0  18.0  19.0  20.0 |
        | .   .  21.0  22.0  23.0  24.0  25.0 |
        | .   .    .     .     .     .     .  |
        | .   .    .     .     .     .     .  |
        | .   .    .     .     .     .     .  |
        | .   .    .     .     .     .     .  |
                                             ┘
```

Example 2

This example shows an in-place matrix conjugate transpose of matrix ***A*** having 5 rows and 5 columns.

Call Statement and Input:

```
                A     LDA   N
                |      |    |
CALL ZGECMI( A(2,3) , 10 , 5 )
```

```
        | .   .      .         .           .           .           .       |
        | .   .  (1.0,1.0) (6.0, 6.0)  (11.0,11.0) (16.0,16.0) (21.0,21.0) |
        | .   .  (2.0,2.0) (7.0, 7.0)  (12.0,12.0) (17.0,17.0) (22.0,22.0) |
        | .   .  (3.0,3.0) (8.0, 8.0)  (13.0,13.0) (18.0,18.0) (23.0,23.0) |
A    =  | .   .  (4.0,4.0) (9.0, 9.0)  (14.0,14.0) (19.0,19.0) (24.0,24.0) |
        | .   .  (5.0,5.0)(10.0,10.0)  (15.0,15.0) (20.0,20.0) (25.0,25.0) |
        | .   .      .         .           .           .           .       |
        | .   .      .         .           .           .           .       |
        | .   .      .         .           .           .           .       |
        | .   .      .         .           .           .           .       |
```

Output:

```
        | .   .       .             .             .            .            .       |
        | .   .   (1.0, -1.0)   (2.0, -2.0)  (3.0, -3.0)  (4.0, -4.0)  (5.0, -5.0) |
        | .   .   (6.0, -6.0)   (7.0, -7.0)  (8.0, -8.0)  (9.0, -9.0) (10.0,-10.0) |
        | .   .  (11.0,-11.0)  (12.0,-12.0) (13.0,-13.0) (14.0,-14.0) (15.0,-15.0) |
A    =  | .   .  (16.0,-16.0)  (17.0,-17.0) (18.0,-18.0) (19.0,-19.0) (20.0,-20.0) |
        | .   .  (21.0,-21.0)  (22.0,-22.0) (23.0,-23.0) (24.0,-24.0) (25.0,-25.0) |
        | .   .       .             .             .            .            .       |
        | .   .       .             .             .            .            .       |
        | .   .       .             .             .            .            .       |
        | .   .       .             .             .            .            .       |
```

SGETMO, DGETMO, CGETMO, ZGETMO, CGECMO, and ZGECMO (General Matrix Transpose or Conjugate Transpose [Out-of-Place])

Purpose

Subroutines SGETMO, DGETMO, CGETMO, and ZGETMO perform a transpose of an *m* by *n* matrix ***A*** out of place, returning the result in matrix ***B***:

$\mathbf{B} \leftarrow \mathbf{A}^T$

Subroutines CGECMO, and ZGECMO perform a conjugate transpose of an *m* by *n* matrix ***A*** out of place, returning the result in matrix ***B***:

$\mathbf{B} \leftarrow \mathbf{A}^H$

Table 114: Data Types.

Data Types	
A, ***B***	**Subroutine**
Short-precision real	SGETMO
Long-precision real	DGETMO
Short-precision complex	CGETMO CGECMO
Long-precision complex	ZGETMO ZGECMO

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SGETMO | DGETMO | CGETMO | ZGETMO | CGECMO | ZGECMO (*a*, *lda*, *m*, *n*, *b*, *ldb*)
C and C++	sgetmo | dgetmo | cgetmo | zgetmo | cgecmo | zgecmo (*a*, *lda*, *m*, *n*, *b*, *ldb*);

On Entry

a

is the matrix ***A*** having *m* rows and *n* columns.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 114 on page 462.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and $lda \geq m$.

m

is the number of rows in matrix ***A*** and the number of columns in matrix ***B***.

Specified as: an integer; $m \geq 0$.

n

is the number of columns in matrix ***A*** and the number of rows in matrix ***B***.

Specified as: an integer; $n \geq 0$.

b

See On Return.

ldb

is the leading dimension of the array specified for *b*.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

On Return

b

is the matrix ***B*** having *n* rows and *m* columns, containing the results of the operation.

Returned as: an *ldb* by (at least) *m* array, containing numbers of the data type indicated in Table 114 on page 462.

Notes

1. The matrix ***B*** must have no common elements with matrix ***A***; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

Subroutines SGETMO, DGETMO, CGETMO, and ZGETMO perform a transpose of matrix ***A*** out of place. For matrix ***A*** with elements a_{ij}, where $i = 1, m$ and $j = 1, n$, the out-of-place transpose is expressed as:

$b_{ji} = a_{ij}$ for $i = 1, m$ and $j = 1, n$

Subroutines CGECMO and ZGECMO perform a conjugate transpose of matrix ***A*** out of place. For matrix ***A*** with elements a_{ij}, where $i = 1, m$ and $j = 1, n$, the out-of-place transpose is expressed as:

$b_{ji} = \bar{a}_{ij}$ for $i = 1, m$ and $j = 1, n$

If *m* or *n* is 0, no computation is performed.

Error conditions

Computational Errors

None

Input-Argument Errors

1. $m < 0$ or $m > lda$
2. $n < 0$ or $n > ldb$
3. $lda \leq 0$
4. $ldb \leq 0$

Examples

Example 1

This example shows an out-of-place matrix transpose of matrix ***A***, having 5 rows and 4 columns, with the result going into matrix ***B***.

Call Statement and Input:

```
                A     LDA   M   N     B     LDB
                |      |    |   |     |      |
CALL SGETMO( A(2,3) , 10 , 5 , 4 , B(2,2) , 6 )
```

```
         | .  .   .     .     .     .   . |
         | .  .  1.0   6.0  11.0  16.0  . |
         | .  .  2.0   7.0  12.0  17.0  . |
         | .  .  3.0   8.0  13.0  18.0  . |
A    =   | .  .  4.0   9.0  14.0  19.0  . |
         | .  .  5.0  10.0  15.0  20.0  . |
         | .  .   .     .     .     .   . |
         | .  .   .     .     .     .   . |
         | .  .   .     .     .     .   . |
         | .  .   .     .     .     .   . |
```

Output:

```
         | .    .     .     .     .     .   . |
         | .   1.0   2.0   3.0   4.0   5.0  . |
B    =   | .   6.0   7.0   8.0   9.0  10.0  . |
         | .  11.0  12.0  13.0  14.0  15.0  . |
         | .  16.0  17.0  18.0  19.0  20.0  . |
         | .    .     .     .     .     .   . |
```

Example 2

This example uses the same input matrix ***A*** as in Example 1 to show that transposes can be achieved in the same array as long as the input and output data do not overlap. On output, the input data is not overwritten in the array.

Call Statement and Input:

```
                A     LDA   M   N     B     LDB
                |      |    |   |     |      |
CALL SGETMO( A(2,3) , 10 , 5 , 4 , A(7,1) , 10 )
```

Output:

```
         |   .     .     .     .     .     .   . |
         |   .     .    1.0   6.0  11.0  16.0  . |
         |   .     .    2.0   7.0  12.0  17.0  . |
         |   .     .    3.0   8.0  13.0  18.0  . |
A    =   |   .     .    4.0   9.0  14.0  19.0  . |
         |   .     .    5.0  10.0  15.0  20.0  . |
         |  1.0   2.0   3.0   4.0   5.0    .   . |
         |  6.0   7.0   8.0   9.0  10.0    .   . |
         | 11.0  12.0  13.0  14.0  15.0    .   . |
         | 16.0  17.0  18.0  19.0  20.0    .   . |
```

Example 3

This example shows an out-of-place matrix conjugate transpose of matrix ***A***, having 5 rows and 4 columns, with the result going into matrix ***B***.

Call Statement and Input:

```
                A     LDA   M   N     B     LDB
                |      |    |   |     |      |
CALL ZGECMO( A(2,3) , 10 , 5 , 4 , B(2,2) , 6 )
```

```
         | .   .       .           .            .           .       |
         | .   .   (1.0,1.0)  (6.0, 6.0)  (11.0,11.0) (16.0,16.0) |
         | .   .   (2.0,2.0)  (7.0, 7.0)  (12.0,12.0) (17.0,17.0) |
         | .   .   (3.0,3.0)  (8.0, 8.0)  (13.0,13.0) (18.0,18.0) |
A    =  | .   .   (4.0,4.0)  (9.0, 9.0)  (14.0,14.0) (19.0,19.0) |
        | .   .   (5.0,5.0) (10.0,10.0)  (15.0,15.0) (20.0,20.0) |
        | .   .       .          .           .           .       |
        | .   .       .          .           .           .       |
        | .   .       .          .           .           .       |
        | .   .       .          .           .           .       |
```

Output:

```
        | .       .            .           .           .           .       |
        | .    (1.0, -1.0)   (2.0, -2.0)  (3.0, -3.0)  (4.0, -4.0)  (5.0, -5.0) |
B    =  | .    (6.0, -6.0)   (7.0, -7.0)  (8.0, -8.0)  (9.0, -9.0) (10.0,-10.0) |
        | .   (11.0,-11.0)  (12.0,-12.0) (13.0,-13.0) (14.0,-14.0) (15.0,-15.0) |
        | .   (16.0,-16.0)  (17.0,-17.0) (18.0,-18.0) (19.0,-19.0) (20.0,-20.0) |
        | .       .            .           .           .           .       |
```

Chapter 10. Linear algebraic equations subroutines

The linear algebraic equation subroutines, provided in four areas, are described here.

Overview of the Linear Algebraic Equation Subroutines

This describes the subroutines in each of the four linear algebraic equation areas:

- "Dense Linear Algebraic Equation Subroutines" on page 467
- "Banded Linear Algebraic Equation Subroutines" on page 472
- "Sparse Linear Algebraic Equation Subroutines" on page 474
- "Linear Least Squares Subroutines" on page 475

Note: Some of the linear algebraic equations were designed in accordance with the LAPACK de facto standard. If these subprograms do not comply with the standard as approved, IBM will consider updating them to do so. If IBM updates these subprograms, the updates could require modifications of the calling application program. For details on LAPACK, see ["9" on page 1253].

Dense Linear Algebraic Equation Subroutines

The dense linear algebraic equation subroutines provide solutions to linear systems of equations for both real and complex general matrices and their transposes, positive definite real symmetric and complex Hermitian matrices, indefinite real or complex symmetric or complex Hermitian matrices, and triangular matrices. Some of these subroutines correspond to the LAPACK routines described in reference ["9" on page 1253].

Table 115: List of LAPACK Dense Linear Algebraic Equation Subroutines.

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SGESV$^{\Delta GPU}$ CGESV$^{\Delta GPU}$ LAPACKE_sgesv$^{\Delta}$ LAPACKE_cgesv$^{\Delta}$	DGESV$^{\Delta GPU}$ ZGESV$^{\Delta GPU}$ LAPACKE_dgesv$^{\Delta}$ LAPACKE_zgesv$^{\Delta}$	"SGESV, DGESV, CGESV, ZGESV (General Matrix Factorization and Multiple Right-Hand Side Solve)" on page 480
SGETRF$^{\Delta GPU}$ CGETRF$^{\Delta GPU}$ LAPACKE_sgetrf$^{\Delta}$ LAPACKE_cgetrf$^{\Delta}$	DGETRF$^{\Delta GPU}$ ZGETRF$^{\Delta GPU}$ LAPACKE_dgetrf$^{\Delta}$ LAPACKE_zgetrf$^{\Delta}$	"SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)" on page 484
SGETRS$^{\Delta GPU}$ CGETRS$^{\Delta GPU}$ LAPACKE_sgetrs$^{\Delta}$ LAPACKE_cgetrs$^{\Delta}$	DGETRS$^{\Delta GPU}$ ZGETRS$^{\Delta GPU}$ LAPACKE_dgetrf$^{\Delta}$ LAPACKE_zgetrf$^{\Delta}$	"SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)" on page 488
SGECON$^{\Delta}$ CGECON$^{\Delta}$ LAPACKE_sgecon Δ LAPACKE_cgecon Δ	DGECON$^{\Delta}$ ZGECON$^{\Delta}$ LAPACKE_dgecon Δ LAPACKE_zgecon Δ	"SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)" on page 504

Table 115: List of LAPACK Dense Linear Algebraic Equation Subroutines. (continued)

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SGETRI$^{\Delta}$ CGETRI$^{\Delta}$ LAPACKE_sgetri$^{\Delta}$ LAPACKE_cgetri$^{\Delta}$	DGETRI$^{\Delta}$ ZGETRI$^{\Delta}$ LAPACKE_dgetri$^{\Delta}$ LAPACKE_zgetri$^{\Delta}$	"SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)" on page 511
SLANGE$^{\Delta}$ CLANGE$^{\Delta}$ LAPACKE_slange$^{\Delta}$ LAPACKE_clange$^{\Delta}$	DLANGE$^{\Delta}$ ZLANGE$^{\Delta}$ LAPACKE_dlange Δ LAPACKE_zlange Δ	"SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)" on page 518
SPPSV$^{\Delta GPU}$ CPPSV$^{\Delta GPU}$ LAPACKE_sppsv$^{\Delta}$ LAPACKE_cppsv$^{\Delta}$	DPPSV$^{\Delta GPU}$ ZPPSV$^{\Delta GPU}$ LAPACKE_dppsv$^{\Delta}$ LAPACKE_zppsv$^{\Delta}$	"SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)" on page 521
SPOSV$^{\Delta GPU}$ CPOSV$^{\Delta GPU}$ LAPACKE_sposv$^{\Delta}$ LAPACKE_cposv$^{\Delta}$	DPOSV$^{\Delta GPU}$ ZPOSV$^{\Delta GPU}$ LAPACKE_dposv$^{\Delta}$ LAPACKE_zposv$^{\Delta}$	"SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)" on page 527
SPOTRF$^{\Delta GPU}$ CPOTRF$^{\Delta GPU}$ SPPTRF$^{\Delta GPU}$ CPPTRF$^{\Delta GPU}$ LAPACKE_spotrf$^{\Delta}$ LAPACKE_cpotrf$^{\Delta}$ LAPACKE_spptrf$^{\Delta}$ LAPACKE_cpptrf$^{\Delta}$	DPOTRF$^{\Delta GPU}$ ZPOTRF$^{\Delta GPU}$ DPPTRF$^{\Delta GPU}$ ZPPTRF$^{\Delta GPU}$ LAPACKE_dpotrf$^{\Delta}$ LAPACKE_zpotrf$^{\Delta}$ LAPACKE_dpptrf$^{\Delta}$ LAPACKE_zpptrf$^{\Delta}$	"SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)" on page 533
SPOTRS$^{\Delta GPU}$ CPOTRS$^{\Delta GPU}$ SPPTRS$^{\Delta GPU}$ CPPTRS$^{\Delta GPU}$ LAPACKE_spotrs$^{\Delta}$ LAPACKE_cpotrs$^{\Delta}$ LAPACKE_spptrs$^{\Delta}$ LAPACKE_cpptrs$^{\Delta}$	DPOTRS$^{\Delta GPU}$ ZPOTRS$^{\Delta GPU}$ DPPTRS$^{\Delta GPU}$ ZPPTRS$^{\Delta GPU}$ LAPACKE_dpotrs$^{\Delta}$ LAPACKE_zpotrs$^{\Delta}$ LAPACKE_dpptrs$^{\Delta}$ LAPACKE_zpptrs$^{\Delta}$	"SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS, CPPTRS, and ZPPTRS (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)" on page 545

Table 115: List of LAPACK Dense Linear Algebraic Equation Subroutines. (continued)		
Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SPOCONΔ CPOCONΔ SPPCONΔ CPPCONΔ LAPACKE_spocon Δ LAPACKE_cpocon Δ LAPACKE_sppcon Δ LAPACKE_cppcon Δ	DPOCONΔ ZPOCONΔ DPPCONΔ ZPPCONΔ LAPACKE_dpocon Δ LAPACKE_zpocon Δ LAPACKE_dppcon Δ LAPACKE_zppcon Δ	"SPOCON, DPOCON, CPOCON, ZPOCON, SPPCON, DPPCON, CPPCON, and ZPPCON (Estimate the Reciprocal of the Condition Number of a Positive Definite Real Symmetric or Complex Hermitian Matrix)" on page 555
SPOTRIΔ CPOTRIΔ SPPTRIΔ CPPTRIΔ LAPACKE_spotriΔ LAPACKE_cpotriΔ LAPACKE_spptriΔ LAPACKE_cpptriΔ	DPOTRIΔ ZPOTRIΔ DPPTRIΔ ZPPTRIΔ LAPACKE_dpotriΔ LAPACKE_zpotriΔ LAPACKE_dpptriΔ LAPACKE_zpptriΔ	"SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPPTRI, DPPTRI, CPPTRI, ZPPTRI, SPPICD, and DPPICD (Positive Definite Real Symmetric or Complex Hermitian Matrix Inverse, Condition Number Reciprocal, and Determinant)" on page 569
SLANSYΔ >|CLANSYΔ|< CLANHEΔ SLANSPΔ CLANHPΔ LAPACKE_slansyΔ >| LAPACKE_clansyΔ |< LAPACKE_clanhe Δ LAPACKE_slanspΔ LAPACKE_clanhp Δ	DLANSYΔ >|ZLANSYΔ|< ZLANHEΔ DLANSPΔ ZLANHPΔ LAPACKE_dlansyΔ >| LAPACKE_zlansyΔ |< LAPACKE_zlanhe Δ LAPACKE_dlansp Δ LAPACKE_zlanhp Δ	"SLANSY, DLANSY, CLANSY, ZLANSY, CLANHE, ZLANHE, SLANSP, DLANSP, CLANHP, and ZLANHP (Real Symmetric or Complex Hermitian Matrix Norm)" on page 579

Table 115: List of LAPACK Dense Linear Algebraic Equation Subroutines. (continued)

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SSYSV$^{\Delta}$	DSYSV$^{\Delta}$	"SSYSV, DSYSV, CSYSV, ZSYSV, CHESV, ZHESV, SSPSV, DSPSV, CSPSV, ZSPSV, CHPSV, and ZHPSV (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)" on page 584
CSYSV$^{\Delta}$	ZSYSV$^{\Delta}$	
CHESV$^{\Delta}$	ZHESV$^{\Delta}$	
SSPSV$^{\Delta}$	DSPSV$^{\Delta}$	
CSPSV$^{\Delta}$	ZSPSV$^{\Delta}$	
CHPSV$^{\Delta}$	ZHPSV$^{\Delta}$	
LAPACKE_ssysv$^{\Delta}$	LAPACKE_dsysv$^{\Delta}$	
LAPACKE_csysv$^{\Delta}$	LAPACKE_zsysv$^{\Delta}$	
LAPACKE_chesv$^{\Delta}$	LAPACKE_zhesv$^{\Delta}$	
LAPACKE_sspsv$^{\Delta}$	LAPACKE_dspsv$^{\Delta}$	
LAPACKE_cspsv$^{\Delta}$	LAPACKE_zspsv$^{\Delta}$	
LAPACKE_chpsv$^{\Delta}$	LAPACKE_zhpsv$^{\Delta}$	
SSYTRF$^{\Delta}$	DSYTRF$^{\Delta}$	"SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, and ZHPTRF (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization)" on page 593
CSYTRF$^{\Delta}$	ZSYTRF$^{\Delta}$	
CHETRF$^{\Delta}$	ZHETRF$^{\Delta}$	
SSPTRF$^{\Delta}$	DSPTRF$^{\Delta}$	
CSPTRF$^{\Delta}$	ZSPTRF$^{\Delta}$	
CHPTRF$^{\Delta}$	ZHPTRF$^{\Delta}$	
LAPACKE_ssytrf$^{\Delta}$	LAPACKE_dsytrf$^{\Delta}$	
LAPACKE_csytrf$^{\Delta}$	LAPACKE_zsytrf$^{\Delta}$	
LAPACKE_chetrf$^{\Delta}$	LAPACKE_zhetrf$^{\Delta}$	
LAPACKE_ssptrf$^{\Delta}$	LAPACKE_dsptrf$^{\Delta}$	
LAPACKE_csptrf$^{\Delta}$	LAPACKE_zsptrf$^{\Delta}$	
LAPACKE_chptrf$^{\Delta}$	LAPACKE_zhptrf$^{\Delta}$	
SSYTRS$^{\Delta}$	DSYTRS$^{\Delta}$	"SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, ZHETRS, SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, and ZHPTRS (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)" on page 601
CSYTRS$^{\Delta}$	ZSYTRS$^{\Delta}$	
CHETRS$^{\Delta}$	ZHETRS$^{\Delta}$	
SSPTRS$^{\Delta}$	DSPTRS$^{\Delta}$	
CSPTRS$^{\Delta}$	ZSPTRS$^{\Delta}$	
CHPTRS$^{\Delta}$	ZHPTRS$^{\Delta}$	
LAPACKE_ssytrs$^{\Delta}$	LAPACKE_dsytrs$^{\Delta}$	
LAPACKE_csytrs$^{\Delta}$	LAPACKE_zsytrs$^{\Delta}$	
LAPACKE_chetrs$^{\Delta}$	LAPACKE_zhetrs$^{\Delta}$	
LAPACKE_ssptrs$^{\Delta}$	LAPACKE_dsptrs$^{\Delta}$	
LAPACKE_csptrs$^{\Delta}$	LAPACKE_zsptrs$^{\Delta}$	
LAPACKE_chptrs$^{\Delta}$	LAPACKE_zhptrs$^{\Delta}$	
STRTRI$^{\Delta}$	DTRTRI$^{\Delta}$	"STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)" on page 621
CTRTRI$^{\Delta}$	ZTRTRI$^{\Delta}$	
STPTRI$^{\Delta}$	DTPTRI$^{\Delta}$	
CTPTRI$^{\Delta}$	ZTPTRI$^{\Delta}$	
LAPACKE_strtri$^{\Delta}$	LAPACKE_dtrtri$^{\Delta}$	
LAPACKE_stptri$^{\Delta}$	LAPACKE_dtptri$^{\Delta}$	
LAPACKE_ctrtri$^{\Delta}$	LAPACKE_ztrtri$^{\Delta}$	
LAPACKE_ctptri$^{\Delta}$	LAPACKE_ztptri$^{\Delta}$	

Table 115: List of LAPACK Dense Linear Algebraic Equation Subroutines. (continued)		
Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SLANTR^Δ CLANTR^Δ SLANTP^Δ CLANTP^Δ LAPACKE_slantr^Δ LAPACKE_clantr^Δ LAPACKE_slantp^Δ LAPACKE_clantp^Δ	DLANTR^Δ ZLANTR^Δ DLANTP^Δ ZLANTP^Δ LAPACKE_dlantr^Δ LAPACKE_zlantr^Δ LAPACKE_dlantp^Δ LAPACKE_zlantp^Δ	"SLANTR, DLANTR, CLANTR, ZLANTR, SLANTP, DLANTP, CLANTP, and ZLANTP (Trapezoidal or Triangular Matrix Norm)" on page 628
^Δ LAPACK > |^GPU This subroutine can offload computations to the GPUs when the ESSL SMP CUDA Library is used.| <		

Table 116: List of Dense Linear Algebraic Equation Subroutines.		
Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SGEF CGEF	DGEF ZGEF DGEFP§	"SGEF, DGEF, CGEF, and ZGEF (General Matrix Factorization)" on page 493
SGESM CGESM	DGESM ZGESM	"SGESM, DGESM, CGESM, and ZGESM (General Matrix, Its Transpose, or Its Conjugate Transpose Multiple Right-Hand Side Solve)" on page 499
SGES CGES	DGES ZGES	"SGES, DGES, CGES, and ZGES (General Matrix, Its Transpose, or Its Conjugate Transpose Solve)" on page 496
SGEFCD	DGEFCD	"SGEFCD and DGEFCD (General Matrix Factorization, Condition Number Reciprocal, and Determinant)" on page 508
SGEICD	DGEICD	"SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)" on page 511
SPOF CPOF SPPF	DPOF ZPOF DPPF DPPFP§	"SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)" on page 533
SPOSM CPOSM	DPOSM ZPOSM	"SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS, CPPTRS, and ZPPTRS (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)" on page 545
SPPS	DPPS	"SPPS and DPPS (Positive Definite Real Symmetric Matrix Solve)" on page 552
SPPFCD SPOFCD	DPPFCD DPOFCD	"SPPFCD, DPPFCD, SPOFCD, and DPOFCD (Positive Definite Real Symmetric Matrix Factorization, Condition Number Reciprocal, and Determinant)" on page 563

Table 116: List of Dense Linear Algebraic Equation Subroutines. (continued)

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SPPICD SPOICD	DPPICD DPOICD	"SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPPTRI, DPPTRI, CPPTRI, ZPPTRI, SPPICD, and DPPICD (Positive Definite Real Symmetric or Complex Hermitian Matrix Inverse, Condition Number Reciprocal, and Determinant)" on page 569
	DBSSV	"DBSSV (Symmetric Indefinite Matrix Factorization and Multiple Right-Hand Side Solve)" on page 607
	DBSTRF	"DBSTRF (Symmetric Indefinite Matrix Factorization)" on page 613
	DBSTRS	"DBSTRS (Symmetric Indefinite Matrix Multiple Right-Hand Side Solve)" on page 617
STRI§ STPI§	DTRI§ DTPI§	"STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)" on page 621
§ This subroutine is provided for migration from earlier releases of ESSL and is not intended for use in new programs. Documentation for this subroutine is no longer provided.		

Banded Linear Algebraic Equation Subroutines

The banded linear algebraic equation subroutines provide solutions to linear systems of equations for:

- Real or complex general band matrices
- Positive definite real symmetric or complex Hermitian band matrices
- Real or complex general tridiagonal matrices
- Positive definite real symmetric or complex Hermitian tridiagonal matrices

Table 117: List of LAPACK Banded Linear Algebraic Equation Subroutines.

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SGBSV∆ CGBSV∆ LAPACKE_sgbsv∆ LAPACKE_cgbsv∆	DGBSV∆ ZGBSV∆ LAPACKE_dgbsv∆ LAPACKE_zgbsv∆	"SGBSV, DGBSV, CGBSV, and ZGBSV (General Band Matrix Factorization and Multiple Right-Hand Side Solve)" on page 635
SGBTRF∆ CGBTRF∆ LAPACKE_sgbtrf∆ LAPACKE_cgbtrf∆	DGBTRF∆ ZGBTRF∆ LAPACKE_dgbtrf∆ LAPACKE_zgbtrf∆	"SGBTRF, DGBTRF, CGBTRF and ZGBTRF (General Band Matrix Factorization)" on page 639
SGBTRS∆ CGBTRS∆ LAPACKE_sgbtrs∆ LAPACKE_cgbtrs∆	DGBTRS∆ ZGBTRS∆ LAPACKE_dgbtrs∆ LAPACKE_zgbtrs∆	"SGBTRS, DGBTRS, CGBTRS, and ZGBTRS (General Band Matrix Multiple Right-Hand Side Solve)" on page 643

Table 117: List of LAPACK Banded Linear Algebraic Equation Subroutines. (continued)

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SPBSV$^{\Delta}$ CPBSV$^{\Delta}$ LAPACKE_spbsv$^{\Delta}$ LAPACKE_cpbsv$^{\Delta}$	DPBSV$^{\Delta}$ ZPBSV$^{\Delta}$ LAPACKE_dpbsv$^{\Delta}$ LAPACKE_zpbsv$^{\Delta}$	"SPBSV, DPBSV, CPBSV, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization and Multiple Right-Hand Side Solve)" on page 651
SPBTRF$^{\Delta}$ CPBTRF$^{\Delta}$ LAPACKE_spbtrf$^{\Delta}$ LAPACKE_cpbtrf$^{\Delta}$	DPBTRF$^{\Delta}$ ZPBTRF$^{\Delta}$ LAPACKE_dpbtrf$^{\Delta}$ LAPACKE_zpbtrf$^{\Delta}$	"SPBTRF, DPBTRF, CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)" on page 656
SPBTRS$^{\Delta}$ CPBTRS$^{\Delta}$ LAPACKE_spbtrs$^{\Delta}$ LAPACKE_cpbtrs$^{\Delta}$	DPBTRS$^{\Delta}$ ZPBTRS$^{\Delta}$ LAPACKE_dpbtrf$^{\Delta}$ LAPACKE_zpbtrf$^{\Delta}$	"SPBTRS, DPBTRS, CPBTRS, and ZPBTRS (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Multiple Right-Hand Side Solve)" on page 661
SGTSV$^{\Delta}$ CGTSV$^{\Delta}$ LAPACKE_sgtsv$^{\Delta}$ LAPACKE_cgtsv$^{\Delta}$	DGTSV$^{\Delta}$ ZGTSV$^{\Delta}$ LAPACKE_sgtsv$^{\Delta}$ LAPACKE_cgtsv$^{\Delta}$	"SGTSV, DGTSV, CGTSV, and ZGTSV (General Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)" on page 666
SGTTRF$^{\Delta}$ CGTTRF$^{\Delta}$ LAPACKE_sgttrf$^{\Delta}$ LAPACKE_cgttrf$^{\Delta}$	DGTTRF$^{\Delta}$ ZGTTRF$^{\Delta}$ LAPACKE_dgttrf$^{\Delta}$ LAPACKE_zgttrf$^{\Delta}$	"SGTTRF, DGTTRF, CGTTRF, and ZGTTRF (General Tridiagonal Matrix Factorization)" on page 670
SGTTRS$^{\Delta}$ CGTTRS$^{\Delta}$ LAPACKE_sgttrs$^{\Delta}$ LAPACKE_cgttrs$^{\Delta}$	DGTTRS$^{\Delta}$ ZGTTRS$^{\Delta}$ LAPACKE_dgttrs$^{\Delta}$ LAPACKE_zgttrs$^{\Delta}$	"SGTTRS, DGTTRS, CGTTRS, and ZGTTRS (General Tridiagonal Matrix Multiple Right-Hand Side Solve)" on page 674
SPTSV$^{\Delta}$ CPTSV$^{\Delta}$ LAPACKE_sptsv$^{\Delta}$ LAPACKE_cptsv$^{\Delta}$	DPTSV$^{\Delta}$ ZPTSV$^{\Delta}$ LAPACKE_dptsv$^{\Delta}$ LAPACKE_zptsv$^{\Delta}$	"SPTSV, DPTSV, CPTSV, and ZPTSV (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)" on page 679
SPTTRF$^{\Delta}$ CPTTRF$^{\Delta}$ LAPACKE_spttrf$^{\Delta}$ LAPACKE_cpttrf$^{\Delta}$	DPTTRF$^{\Delta}$ ZPTTRF$^{\Delta}$ LAPACKE_dpttrf$^{\Delta}$ LAPACKE_zpttrf$^{\Delta}$	"SPTTRF, DPTTRF, CPTTRF, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)" on page 684

Table 117: List of LAPACK Banded Linear Algebraic Equation Subroutines. (continued)

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SPTTRS$^{\Delta}$ CPTTRS$^{\Delta}$ LAPACKE_spttrs$^{\Delta}$ LAPACKE_cpttrs$^{\Delta}$	DPTTRS$^{\Delta}$ ZPTTRS$^{\Delta}$ LAPACKE_dpttrs$^{\Delta}$ LAPACKE_zpttrs$^{\Delta}$	"SPTTRS, DPTTRS, CPTTRS, and ZPTTRS (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Multiple Right-Hand Solve)" on page 687
$^{\Delta}$ LAPACK		

Table 118: List of non-LAPACK Banded Linear Algebraic Equation Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SGBF§	DGBF§	"SGBF and DGBF (General Band Matrix Factorization)" on page 693
SGBS§	DGBS§	"SGBS and DGBS (General Band Matrix Solve)" on page 649
SPBF§ SPBCHF§	DPBF§ DPBCHF§	"SPBF, DPBF, SPBCHF, and DPBCHF (Positive Definite Symmetric Band Matrix Factorization)" on page 698
SPBS§ SPBCHS§	DPBS§ DPBCHS§	"SPBS, DPBS, SPBCHS, and DPBCHS (Positive Definite Symmetric Band Matrix Solve)" on page 701
SGTF§	DGTF§	"SGTF and DGTF (General Tridiagonal Matrix Factorization)" on page 704
SGTS§	DGTS§	"SGTS and DGTS (General Tridiagonal Matrix Solve)" on page 706
SGTNP CGTNP	DGTNP ZGTNP	"SGTNP, DGTNP, CGTNP, and ZGTNP (General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting)" on page 708
SGTNPF CGTNPF	DGTNPF ZGTNPF	"SGTNPF, DGTNPF, CGTNPF, and ZGTNPF (General Tridiagonal Matrix Factorization with No Pivoting)" on page 711
SGTNPS CGTNPS	DGTNPS ZGTNPS	"SGTNPS, DGTNPS, CGTNPS, and ZGTNPS (General Tridiagonal Matrix Solve with No Pivoting)" on page 714
SPTF§	DPTF§	"SPTF and DPTF (Positive Definite Symmetric Tridiagonal Matrix Factorization)" on page 716
SPTS§	DPTS§	"SPTS and DPTS (Positive Definite Symmetric Tridiagonal Matrix Solve)" on page 718
§ This subroutine is provided for migration from earlier releases of ESSL and is not intended for use in new programs.		

Sparse Linear Algebraic Equation Subroutines

The sparse linear algebraic equation subroutines provide direct and iterative solutions to linear systems of equations both for general sparse matrices and their transposes and for sparse symmetric matrices.

Table 119: List of Sparse Linear Algebraic Equation Subroutines.

Long-Precision Subroutine	**Descriptive Name and Location**
DGSF	"DGSF (General Sparse Matrix Factorization Using Storage by Indices, Rows, or Columns)" on page 720
DGSS	"DGSS (General Sparse Matrix or Its Transpose Solve Using Storage by Indices, Rows, or Columns)" on page 725
DGKFS DGKFSP§	"DGKFS (General Sparse Matrix or Its Transpose Factorization, Determinant, and Solve Using Skyline Storage Mode)" on page 729
DSKFS DSKFSP§	"DSKFS (Symmetric Sparse Matrix Factorization, Determinant, and Solve Using Skyline Storage Mode)" on page 744
DSRIS	"DSRIS (Iterative Linear System Solver for a General or Symmetric Sparse Matrix Stored by Rows)" on page 760
DSMCG‡	"DSMCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Matrix Storage Mode)" on page 769
DSDCG	"DSDCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)" on page 776
DSMGCG‡	"DSMGCG (General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode)" on page 782
DSDGCG	"DSDGCG (General Sparse Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)" on page 788

§ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs. Documentation for this subroutine is no longer provided.

‡ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs. Use DSRIS instead.

Linear Least Squares Subroutines

The linear least squares subroutines provide least squares solutions to linear systems of equations for general matrices using a QR factorization or a singular value decomposition. Some of these subroutines correspond to the LAPACK routines described in reference ["9" on page 1253].

Table 120: List of LAPACK Linear Least Squares Subroutines.

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SGESVD^Δ CGESVD^Δ SGESDD^Δ CGESDD^Δ LAPACKE_sgesvd Δ LAPACKE_cgesvd Δ LAPACKE_sgesdd Δ LAPACKE_cgesdd Δ	DGESVD^Δ ZGESVD^Δ DGESDD^Δ ZGESDD^Δ LAPACKE_dgesvd Δ LAPACKE_zgesvd Δ LAPACKE_dgesdd Δ LAPACKE_zgesdd Δ	"SGESVD, DGESVD, CGESVD, ZGESVD, SGESDD, DGESDD, CGESDD, and ZGESDD (Singular Value Decomposition for a General Matrix)" on page 794
SGEQRF^ΔGPU CGEQRF^ΔGPU LAPACKE_sgeqrf^Δ LAPACKE_cgeqrf^Δ	DGEQRF^ΔGPU ZGEQRF^ΔGPU LAPACKE_dgeqrf^Δ LAPACKE_zgeqrf^Δ	"SGEQRF, DGEQRF, CGEQRF, and ZGEQRF (General Matrix QR Factorization)" on page 807
SGELS^Δ GPU CGELS^Δ GPU LAPACKE_sgels^Δ LAPACKE_cgels^Δ	DGELS^Δ GPU ZGELS^Δ GPU LAPACKE_dgels^Δ LAPACKE_zgels^Δ	"SGELS, DGELS, CGELS, and ZGELS (Linear Least Squares Solution for a General Matrix)" on page 812
SGELSD^Δ CGELSD^Δ LAPACKE_sgelsd^Δ LAPACKE_cgelsd^Δ	DGELSD^Δ ZGELSD^Δ LAPACKE_dgelsd Δ LAPACKE_zgelsd^Δ	"SGELSD, DGELSD, CGELSD, and ZGELSD (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)" on page 822

^Δ LAPACK

> |^GPU This subroutine can offload computations to the GPUs when the ESSL SMP CUDA Library is used.| <

Table 121: List of Non–LAPACK Linear Least Squares Subroutines.

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SGESVF§	DGESVF§	"SGESVF and DGESVF (Singular Value Decomposition for a General Matrix)" on page 830
SGESVS§	DGESVS§	"SGESVS and DGESVS (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)" on page 837
SGELLS§	DGELLS§	"SGELLS and DGELLS (Linear Least Squares Solution for a General Matrix with Column Pivoting)" on page 841

§ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs.

Dense and Banded Linear Algebraic Equation Considerations

This provides some key points about using the dense and banded linear algebraic equation subroutines.

Use Considerations

To solve a system of equations, you have two choices:

- Use the combined factorization-and-solve subroutine for the type of matrix you have.
- Use both the factorization subroutine and the solve subroutine for the type of matrix you have. When doing so, note the following:
  - Each factorization subroutine should be followed in your program by the corresponding solve subroutine. The output from the factorization subroutine should be used as input to the solve subroutine.
  - To solve a system of equations with one or more right-hand sides, follow the call to the factorization subroutine with one or more calls to a solve subroutine or one call to a multiple solve subroutine.

Performance and Accuracy Considerations

1. Except in a few instances, the _GTNP subroutines provide better performance than the _GTNPF and _GTNPS subroutines. For details, see the subroutine descriptions.
2. The general subroutines (dense and banded) use partial pivoting for accuracy and fast performance.
3. The short-precision subroutines provide increased accuracy by accumulating intermediate results in long precision when the VSX unit is not used. Occasionally, for performance reasons, these intermediate results are stored.
4. There are ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 61.

Sparse Matrix Direct Solver Considerations

This provides some key points about using the sparse matrix direct solver subroutines.

Use Considerations

1. To solve a sparse system of equations by a direct method, you must use both the factorization and solve subroutines. The factorization subroutine should be followed in your program by the corresponding solve subroutine; that is, the output from the factorization subroutine should be used as input to the solve subroutine.
2. To solve a system of equations with one or more right-hand sides, follow the call to the factorization subroutine with one or more calls to the solve subroutine.
3. The amount of storage required for the arrays depends on the sparsity pattern of the matrix. The requirement that $lna > 2nz$ on entry to DGSF does not guarantee a successful run of the program. Some programs may be terminated because of the large number of fill-ins generated upon factorization. Fill-ins generated in a program depend on the structure of each matrix. If a large number of fill-ins is anticipated when factoring a matrix, the value of *lna* should be large enough to accommodate your problem.

Performance and Accuracy Considerations

1. To make the subroutine more efficient, an input matrix comprised of all nonzero elements is preferable. See the syntax description of each subroutine for details.
2. DGSF optionally checks the validity of the indices and pointers of the input matrix. Use of this option is suggested; however, it may affect performance. For details, see the syntax description for DGSF.

3. In DGSS, if there are multiple sparse right-hand sides to be solved, you should take advantage of the sparsity by selecting a proper value for *jopt* (such as *jopt* = 10 or 11). If there is only one right-hand side to be solved, it is suggested that you do not exploit the sparsity.
4. In DGSF, the value you enter for the lower bound of all elements in the matrix (RPARM(1)) affects the accuracy of the result. Specifying a larger number allows you to gain some performance; however, you may lose some accuracy in the solution.
5. In DGSF, the threshold pivot tolerance (RPARM(2)) is used to select pivots. A value that is close to 0.0 approaches no pivoting. A value close to 1.0 approaches partial pivoting. A value of 0.1 is considered to be a good compromise between numerical stability and sparsity.
6. If the ESSL subroutine performs storage compressions, you receive an attention message. When this occurs, the performance of this subroutine is affected. You can improve the performance by increasing the value specified for *lna*.
7. There are ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 61.

Sparse Matrix Skyline Solver Considerations

This provides some key points about using the sparse matrix skyline solver subroutines.

Use Considerations

1. To solve a system of equations with one or more right-hand sides, where the matrix is stored in skyline storage mode, you can use either of the following methods. The factored output matrix is the same for both of these methods.

   - Call the skyline subroutine with the combined factor-and-solve option.
   - Call the skyline subroutine with the factor-only option, followed in your program by a call to the same subroutine with the solve-only option. The factored output matrix resulting from the factorization should be used as input to the same subroutine to do the solve. You can solve for the right-hand sides in a single call or in individual calls.

   You also have the option of doing a partial factorization, where the subroutine assumes that the initial part of the input matrix is already factored. It then factors the remaining rows and columns. If you want, you can factor a very large matrix progressively by using this option.
2. Forward elimination can be done with or without scaling the right-hand side by the diagonal matrix elements. To perform the computation without scaling, call DGKFS with the normal solve-only option, and define the upper triangular skyline matrix (AU) as a diagonal. To perform the computation with scaling, call DGKFS with the transpose solve-only, option and define the lower triangular skyline matrix (AL) as a diagonal.
3. Back substitution can be done with or without scaling the right-hand side by the diagonal matrix elements. To perform the computation without scaling, call DGKFS with the transpose solve-only option, and define the upper triangular skyline matrix (AU) as a diagonal. To perform the computation with scaling, call DGKFS with the normal solve-only option, and define the lower triangular skyline matrix (AL) as a diagonal.

Performance and Accuracy Considerations

1. For optimal performance, use diagonal-out skyline storage mode for both your input and output matrices. If you specify profile-in skyline storage mode for your input matrix, and either you do not plan to use the factored output or you plan to do a solve only, it is more efficient to specify diagonal-out skyline storage mode for your output matrix. These rules apply to all the computations.
2. In some cases, elapsed time may be reduced significantly by using the combined factor-and-solve option to solve for all right-hand sides at once, in conjunction with the factorization, rather than doing the factorization and solve separately.

3. If you do a solve only, and you solve for more than one right-hand side, it is most efficient to call the skyline subroutine once with all right-hand sides, rather than once for each right-hand side.
4. The skyline subroutines allow some control over processing of the pivot (diagonal) elements of the matrix during the factorization phase. Pivot processing is controlled by `IPARM(10)` through `IPARM(15)` and `RPARM(10)` through `RPARM(15)`. If a pivot occurs within a range that is designated to be fixed (`IPARM(0)` = 1, `IPARM(10)` = 1, and the appropriate element `IPARM(11)` through `IPARM(15)` = 1), it is replaced with the corresponding element of `RPARM(11)` through `RPARM(15)`. Should this pivot fix-up occur, you receive an attention message. This message indicates that the matrix being factored may be unstable (singular or not definite). The results produced in this situation may be inaccurate, and you should review them carefully.

Sparse Matrix Iterative Solver Considerations

This provides some key points about using the sparse matrix iterative solver subroutines.

Use Considerations

If you need to solve linear systems with different right-hand sides but with the same matrix using the preconditioned algorithms, you can reuse the incomplete factorization computed during the first call to the subroutine.

Performance and Accuracy Considerations

1. The DSMCG and DSMGCG subroutines are provided for migration purposes from earlier releases of ESSL. You get better performance and a wider choice of algorithms if you use the DSRIS subroutine.
2. To select the sparse matrix subroutine that provides the best performance, you must consider the sparsity pattern of the matrix. From this, you can determine the most efficient storage mode for your sparse matrix. ESSL provides a number of versions of the sparse matrix iterative solve subroutines. They operate on sparse matrices stored in row-wise, diagonal, and compressed-matrix storage modes. These storage modes are described in "Sparse Matrix" on page 110.

   Storage-by-rows is generally applicable. You should use this storage mode unless your matrices are already set up in one of the other storage modes. If, however, your matrix has a regular sparsity pattern—that is, where the nonzero elements are concentrated along a few diagonals—you may want to use compressed-diagonal storage mode. This can save some storage space. Compressed-matrix storage mode is provided for migration purposes from earlier releases of ESSL and is not intended for use. (You get better performance and a wider choice of algorithms if you use the DSRIS subroutine, which uses storage-by-rows.)
3. The performance achieved in the sparse matrix iterative solver subroutines depends on the value specified for the relative accuracy ε.
4. You can select the iterative algorithm you want to use to solve your linear system. The methods include conjugate gradient (CG), conjugate gradient squared (CGS), generalized minimum residual (GMRES), more smoothly converging variant of the CGS method (Bi-CGSTAB), or transpose-free quasi-minimal residual method (TFQMR).
5. For a general sparse or positive definite symmetric matrix, the iterative algorithm may fail to converge for one of the following reasons:
   - The value of ε is too small, asking for too much precision.
   - The maximum number of iterations is too small, allowing too few iterations for the algorithm to converge.
   - The matrix is not positive real; that is, the symmetric part, $(\boldsymbol{A}+\boldsymbol{A}^T)/2$, is not positive definite.
   - The matrix is ill-conditioned, which may cause overflows during the computation.
6. These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.

Linear Least Squares Considerations

This provides some key points about using the linear least squares subroutines.

Use Considerations

If you want to use a singular value decomposition method to compute the minimal norm linear least squares solution of $\boldsymbol{AX} \cong \boldsymbol{B}$, calls to SGESVF or DGESVF should be followed by calls to SGESVS or DGESVS, respectively.

Performance and Accuracy Considerations

1. Least squares solutions obtained by using a singular value decomposition require more storage and run time than those obtained using a QR decomposition with column pivoting. The singular value decomposition method, however, is a more reliable way to handle rank deficiency.
2. The short-precision subroutines provide increased accuracy by accumulating intermediate results in long precision when the VSX unit is not used. Occasionally, for performance reasons, these intermediate results are stored.
3. The accuracy of the resulting singular values and singular vectors varies between the short- and long-precision versions of each subroutine. The degree of difference depends on the size and conditioning of the matrix computation.
4. There are ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 61.

Dense Linear Algebraic Equation Subroutines

This contains the dense linear algebraic equation subroutine descriptions.

SGESV, DGESV, CGESV, ZGESV (General Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system of linear equations $\boldsymbol{AX} = \boldsymbol{B}$ for $\boldsymbol{X}$, where $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{X}$ are general matrices.

The matrix $\boldsymbol{A}$ is factored using Gaussian elimination with partial pivoting.

Table 122: Data Types.

A*, *B	**Subroutine**
Short-precision real	SGESV^
Long-precision real	DGESV^
Short-precision complex	CGESV^
Long-precision complex	ZGESV^

^ LAPACK

Syntax

Table 123: Syntax for supported computing languages.

Language	Syntax
Fortran	CALL SGESV | DGESV | CGESV | ZGESV (*n*, *nrhs*, *a*, *lda*, *ipvt*, *bx*, *ldb*, *info*)
C and C++	sgesv | dgesv | cgesv | zgesv (*n*, *nrhs*, *a*, *lda*, *ipvt*, *bx*, *ldb*, *info*);
LAPACKE	info = LAPACKE_sgesv | LAPACKE_dgesv | LAPACKE_cgesv | LAPACKE_zgesv (*matrix_layout*, *n*, *nrhs*, *a*, *lda*, *ipvt*, *bx*, *ldb*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

n

is the order *n* of matrix ***A*** and the number of rows of matrix ***B***.

Specified as: an integer; $n \geq 0$, $n \leq lda$, and $n \leq ldb$.

nrhs

is the number of right-hand sides; that is, the number of columns of matrix ***B***.

Specified as: an integer; $nrhs \geq 0$.

a

is the general matrix ***A*** to be factored.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 122 on page 480.

lda

is the leading dimension of the array specified for ***A***.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

ipvt

See On Return.

bx

is the general matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of matrix ***B***.

Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 122 on page 480.

ldb

is the leading dimension of the array specified for ***B***.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

info

See On Return.

On Return

a

is the transformed matrix ***A*** of order *n*, containing the results of the factorization.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 122 on page 480. See "Function" on page 482.

ipvt
is the integer vector of length *n*, containing the pivot indices.

Returned as: a one-dimensional array of (at least) length *n*, containing integers, where $1 \le ipvt(i) \le n$.

bx
is the matrix ***X***, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of ***X***.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 122 on page 480.

info
has the following meaning:

If *info* = 0, the subroutine completed successfully.

If *info* > 0, the factorization was unsuccessful and the solution was not computed. *info* is set equal to the first *i* where $\boldsymbol{U}_{ii}$ is singular and its inverse could not be computed.

Returned as:

- For SGESV, DGESV, CGESV, and ZGESV returned as: an integer; info ≥ 0.
- For LAPACKE_sgesv, LAPACKE_dgesv, LAPACKE_cgesv, and LAPACKE_zgesv returned as an integer function value; info ≥ 0.

Note:

1. In your C program, argument *info* must be passed by reference.
2. The matrices and vector used in this computation must have no common elements; otherwise, results are unpredictable.
3. The way these subroutines handle singularity differs from LAPACK. Like LAPACK, these subroutines use the *info* argument to provide information about the singularity of ***A***, but they also provide an error message.
4. On both input and output, matrices ***A*** and ***B*** conform to LAPACK format.

Function

These subroutines solve the system of linear equations ***AX*** = ***B*** for ***X***, where ***A***, ***B***, and ***X*** are general matrices.

The matrix ***A*** is factored using Gaussian elimination with partial pivoting to compute the ***LU*** factorization of ***A***, where:

A=PLU

and

L is a unit lower triangular matrix.
U is an upper triangular matrix.
P is the permutation matrix.

If *n* is 0, no computation is performed and the subroutine returns after doing some parameter checking. If *n* > 0 and *nrhs* is 0, no solutions are computed and the subroutine returns after factoring the matrix.

See references ["9" on page 1253], ["46" on page 1255], and ["75" on page 1257].

Error conditions

Resource Errors

Unable to allocate internal work area.

Computational Errors

Matrix ***A*** is singular.

- The first column, *i*, of ***L*** with a corresponding $\boldsymbol{U}_{ii}$ = 0 diagonal element is identified in the computational error message.
- The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2146 is set to be unlimited in the ESSL error option table.

Input-Argument Errors

1. $n < 0$
2. $nrhs < 0$
3. $n > lda$
4. $lda \le 0$
5. $n > ldb$
6. $ldb \le 0$

Examples

Example 1

This example shows how to solve the system ***AX*** = ***B***, where:

Matrix ***A*** is the same used as input in Example 1 for DGETRF.
Matrix ***B*** is the same used as input in Example 1 for DGETRS.

Call Statement and Input:

```
             N  NRHS  A  LDA  IPVT  BX  LDB  INFO
             |   |    |   |    |    |    |    |
CALL DGESV( 9 , 5 ,  A , 9 , IPVT, BX , 9 , INFO)
```

A = (same as input A in Example 1)
BX = (same as input BX in Example 1)

Output:

```
IPIV     =  (9, 9, 9, 9, 9, 9, 9, 9, 9)
```

```
        | 2.6   2.4  2.2  2.0  1.8  1.6  1.4  1.2  1.0 |
        | 0.4   0.3  0.6  0.8  1.1  1.4  1.7  1.9  2.2 |
        | 0.5  -0.4  0.4  0.8  1.2  1.6  2.0  2.4  2.8 |
        | 0.5  -0.3  0.0  0.4  0.8  1.2  1.6  2.0  2.4 |
A     = | 0.6  -0.3  0.0  0.0  0.4  0.8  1.2  1.6  2.0 |
        | 0.7  -0.2  0.0  0.0  0.0  0.4  0.8  1.2  1.6 |
        | 0.8  -0.2  0.0  0.0  0.0  0.0  0.4  0.8  1.2 |
        | 0.8  -0.1  0.0  0.0  0.0  0.0  0.0  0.4  0.8 |
        | 0.9  -0.1  0.0  0.0  0.0  0.0  0.0  0.0  0.4 |
```

```
        | 1.0   2.0   3.0   4.0   5.0 |
        | 2.0   4.0   6.0   8.0  10.0 |
        | 3.0   6.0   9.0  12.0  15.0 |
        | 4.0   8.0  12.0  16.0  20.0 |
BX    = | 5.0  10.0  15.0  20.0  25.0 |
        | 6.0  12.0  18.0  24.0  30.0 |
        | 7.0  14.0  21.0  28.0  35.0 |
        | 8.0  16.0  24.0  32.0  40.0 |
        | 9.0  18.0  27.0  36.0  45.0 |
```

```
INFO  =  0
```

Example 2

This example shows how to solve the system ***AX*** = ***B***, where:

Matrix ***A*** is the same used as input in Example 2 for ZGETRF.
Matrix ***B*** is the same used as input in Example 2 for ZGETRS.

Call Statement and Input:

```
            N  NRHS  A  LDA  IPVT  BX  LDB  INFO
            |   |    |   |     |    |    |    |
CALL ZGESV( 9 , 5 ,  A , 9 , IPVT, BX,  9 , INFO)
```

A = (same as input A in Example 2)
IPVT = (same as input IPVT in Example 2)
BX = (same as input BX in Example 2)

Output:

```
        | (1.0,1.0)  (1.0,2.0)  (1.0,3.0) (1.0,4.0) (1.0,5.0) |
        | (2.0,1.0)  (2.0,2.0)  (2.0,3.0) (2.0,4.0) (2.0,5.0) |
        | (3.0,1.0)  (3.0,2.0)  (3.0,3.0) (3.0,4.0) (3.0,5.0) |
        | (4.0,1.0)  (4.0,2.0)  (4.0,3.0) (4.0,4.0) (4.0,5.0) |
BX    = | (5.0,1.0)  (5.0,2.0)  (5.0,3.0) (5.0,4.0) (5.0,5.0) |
        | (6.0,1.0)  (6.0,2.0)  (6.0,3.0) (6.0,4.0) (6.0,5.0) |
        | (7.0,1.0)  (7.0,2.0)  (7.0,3.0) (7.0,4.0) (7.0,5.0) |
        | (8.0,1.0)  (8.0,2.0)  (8.0,3.0) (8.0,4.0) (8.0,5.0) |
        | (9.0,1.0)  (9.0,2.0)  (9.0,3.0) (9.0,4.0) (9.0,5.0) |
```

```
INFO     =  0
```

SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)

Purpose

These subroutines factor general matrix ***A*** using Gaussian elimination with partial pivoting.

To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGETRS, DGETRS CGETRS, or ZGETRS, respectively.

To compute the inverse of matrix ***A***, follow the call to these subroutines with a call to SGETRI, DGETRI, CGETRI, or ZGETRI, respectively.

To estimate the reciprocal of the condition number of matrix ***A***, follow the call to these subroutines with a call to SGECON, DGECON, CGECON, or ZGECON, respectively.

Table 124: Data Types.

A	**Subroutine**
Short-precision real	SGETRF▵
Long-precision real	DGETRF▵
Short-precision complex	CGETRF▵
Long-precision complex	ZGETRF▵

Table 124: Data Types. (continued)

A	**Subroutine**
^ LAPACK	

Note: The output from each of these subroutines should be used only as input for specific other subroutines, as shown in the table below:

Output from this subroutine:	**Should be used only as input to the following subroutines:**		
Solve	**Inverse**	**Reciprocal of the condition number**	
SGETRF	SGETRS	SGETRI	SGECON
DGETRF	DGETRS	DGETRI	DGECON
CGETRF	CGETRS	CGETRI	CGECON
ZGETRF	ZGETRS	ZGETRI	ZGECON

Syntax

Table 125: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SGETRF | DGETRF | CGETRF | ZGETRF (*m*, *n*, *a*, *lda*, *ipvt*, *info*)
C and C++	sgetrf | dgetrf | cgetrf | zgetrf (*m*, *n*, *a*, *lda*, *ipvt*, *info*);
LAPACKE	info = LAPACKE_sgetrf | LAPACKE_dgetrf | LAPACKE_cgetrf | LAPACKE_zgetrf (*matrix_layout*, *m*, *n*, *a*, *lda*, *ipvt*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

m

the number of rows in general matrix ***A*** used in the computation.

Specified as: an integer; $0 \leq m \leq lda$.

n

the number of columns in general matrix ***A*** used in the computation.

Specified as: an integer; $n \geq 0$.

a

is the *m* by *n* general matrix ***A*** to be factored.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 124 on page 484.

lda

is the leading dimension of matrix ***A***.

Specified as: an integer; *lda* > 0 and *lda* ≥ *m*.

ipvt
See On Return.

info
See On Return.

On Return

a
is the *m* by *n* transformed matrix ***A***, containing the results of the factorization. See "Function" on page 486. Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 124 on page 484.

ipvt
is the integer vector ***ipvt*** of length min(*m*,*n*), containing the pivot indices. Returned as: a one-dimensional array of (at least) length min(*m*,*n*), containing integers,where 1 ≤ *ipvt(i)* ≤ *m*.

info
has the following meaning:

If *info* = 0, the factorization of general matrix ***A*** completed successfully.

If *info* > 0, *info* is set equal to the first *i* where $\boldsymbol{U}_{ii}$ is singular and its inverse could not be computed.

Returned as:

- For SGETRF, DGETRF, CGETRF and ZGETRF returned as: an integer; info ≥ 0.
- For LAPACKE_sgetrf, LAPACKE_dgetrf, LAPACKE_cgetrf and LAPACKE_zgetrf, returned as an integer function value; info ≥ 0.

Returned as: an integer; *info* ≥ 0.

Note:

1. In your C program, argument *info* must be passed by reference.
2. The matrix ***A*** and vector ***ipvt*** must have no common elements; otherwise results are unpredictable.
3. The way these subroutines handle singularity differs from LAPACK. Like LAPACK, these subroutines use the *info* argument to provide information about the singularity of ***A***, but they also provide an error message.
4. On both input and output, matrix ***A*** conforms to LAPACK format.

Function

The matrix ***A*** is factored using Gaussian elimination with partial pivoting to compute the ***LU*** factorization of ***A***, where:

A=PLU

and

L is a unit lower triangular matrix.
U is an upper triangular matrix.
P is the permutation matrix.

On output, the transformed matrix ***A*** contains ***U*** in the upper triangle (if *m* ≥ *n*) or upper trapezoid (if *m* < *n*) and ***L*** in the strict lower triangle (if *m* ≤ *n*) or lower trapezoid (if *m* > *n*). ***ipvt*** contains the pivots representing permutation ***P***, such that ***A*** = ***PLU***.

If *m* or *n* is 0, no computation is performed and the subroutine returns after doing some parameter checking. See references ["9" on page 1253],["46" on page 1255], and ["75" on page 1257].

Error conditions

Resource Errors

Unable to allocate internal work area.

Computational Errors

Matrix ***A*** is singular.

- The first column, *i*, of ***L*** with a corresponding $\boldsymbol{U}_{ii} = 0$ diagonal element is identified in the computational error message.
- The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2146 is set to be unlimited in the ESSL error option table.

Input-Argument Errors

1. $m < 0$
2. $n < 0$
3. $m > lda$
4. $lda \le 0$

Examples

Example 1

This example shows a factorization of a real general matrix ***A*** of order 9.

Call Statement and Input:

```
              M   N   A  LDA  IPVT  INFO
              |   |   |   |     |     |
CALL DGETRF( 9 , 9 , A,  9 , IPVT, INFO )
```

```
        | 1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.6 |
        | 1.2  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4 |
        | 1.4  1.2  1.0  1.2  1.4  1.6  1.8  2.0  2.2 |
        | 1.6  1.4  1.2  1.0  1.2  1.4  1.6  1.8  2.0 |
A    =  | 1.8  1.6  1.4  1.2  1.0  1.2  1.4  1.6  1.8 |
        | 2.0  1.8  1.6  1.4  1.2  1.0  1.2  1.4  1.6 |
        | 2.2  2.0  1.8  1.6  1.4  1.2  1.0  1.2  1.4 |
        | 2.4  2.2  2.0  1.8  1.6  1.4  1.2  1.0  1.2 |
        | 2.6  2.4  2.2  2.0  1.8  1.6  1.4  1.2  1.0 |
```

Output:

```
        | 2.6   2.4  2.2  2.0  1.8  1.6  1.4  1.2  1.0 |
        | 0.4   0.3  0.6  0.8  1.1  1.4  1.7  1.9  2.2 |
        | 0.5  -0.4  0.4  0.8  1.2  1.6  2.0  2.4  2.8 |
        | 0.5  -0.3  0.0  0.4  0.8  1.2  1.6  2.0  2.4 |
A    =  | 0.6  -0.3  0.0  0.0  0.4  0.8  1.2  1.6  2.0 |
        | 0.7  -0.2  0.0  0.0  0.0  0.4  0.8  1.2  1.6 |
        | 0.8  -0.2  0.0  0.0  0.0  0.0  0.4  0.8  1.2 |
        | 0.8  -0.1  0.0  0.0  0.0  0.0  0.0  0.4  0.8 |
        | 0.9  -0.1  0.0  0.0  0.0  0.0  0.0  0.0  0.4 |
```

```
IPVT     =  (9, 9, 9, 9, 9, 9, 9, 9, 9)
INFO     =  0
```

Example 2

This example shows a factorization of a complex general matrix ***A*** of order 9.

Call Statement and Input:

```
             M   N   A  LDA  IPVT  INFO
             |   |   |   |     |     |
CALL ZGETRF( 9 , 9 , A,  9 , IPVT, INFO )
```

```
      | (2.0, 1.0) (2.4,-1.0) (2.8,-1.0) (3.2,-1.0)  (3.6,-1.0) (4.0,-1.0) (4.4,-1.0) (4.8,-1.0) (5.2,-1.0) |
      | (2.4, 1.0) (2.0, 1.0) (2.4,-1.0) (2.8,-1.0)  (3.2,-1.0) (3.6,-1.0) (4.0,-1.0) (4.4,-1.0) (4.8,-1.0) |
      | (2.8, 1.0) (2.4, 1.0) (2.0, 1.0) (2.4,-1.0)  (2.8,-1.0) (3.2,-1.0) (3.6,-1.0) (4.0,-1.0) (4.4,-1.0) |
      | (3.2, 1.0) (2.8, 1.0) (2.4, 1.0) (2.0, 1.0)  (2.4,-1.0) (2.8,-1.0) (3.2,-1.0) (3.6,-1.0) (4.0,-1.0) |
A   = | (3.6, 1.0) (3.2, 1.0) (2.8, 1.0) (2.4, 1.0)  (2.0, 1.0) (2.4,-1.0) (2.8,-1.0) (3.2,-1.0) (3.6,-1.0) |
      | (4.0, 1.0) (3.6, 1.0) (3.2, 1.0) (2.8, 1.0)  (2.4, 1.0) (2.0, 1.0) (2.4,-1.0) (2.8,-1.0) (3.2,-1.0) |
      | (4.4, 1.0) (4.0, 1.0) (3.6, 1.0) (3.2, 1.0)  (2.8, 1.0) (2.4, 1.0) (2.0, 1.0) (2.4,-1.0) (2.8,-1.0) |
      | (4.8, 1.0) (4.4, 1.0) (4.0, 1.0) (3.6, 1.0)  (3.2, 1.0) (2.8, 1.0) (2.4, 1.0) (2.0, 1.0) (2.4,-1.0) |
      | (5.2, 1.0) (4.8, 1.0) (4.4, 1.0) (4.0, 1.0)  (3.6, 1.0) (3.2, 1.0) (2.8, 1.0) (2.4, 1.0) (2.0, 1.0) |
```

Output:

```
      | (5.2, 1.0) (4.8, 1.0)  (4.4, 1.0)  (4.0, 1.0)   (3.6, 1.0)  (3.2, 1.0) (2.8, 1.0) (2.4, 1.0) (2.0, 1.0) |
      | (0.4, 0.1) (0.6,-2.0)  (1.1,-1.9)  (1.7,-1.9)   (2.3,-1.8)  (2.8,-1.8) (3.4,-1.7) (3.9,-1.7) (4.5,-1.6) |
      | (0.5, 0.1) (0.0,-0.1)  (0.6,-1.9)  (1.2,-1.8)   (1.8,-1.7)  (2.5,-1.6) (3.1,-1.5) (3.7,-1.4) (4.3,-1.3) |
      | (0.6, 0.1) (0.0,-0.1) (-0.1,-0.1)  (0.7,-1.9)   (1.3,-1.7)  (2.0,-1.6) (2.7,-1.5) (3.4,-1.4) (4.0,-1.2) |
A   = | (0.6, 0.1) (0.0,-0.1) (-0.1,-0.1) (-0.1, 0.0)   (0.7,-1.9)  (1.5,-1.7) (2.2,-1.6) (2.9,-1.5) (3.7,-1.3) |
      | (0.7, 0.1) (0.0,-0.1)  (0.0, 0.0) (-0.1, 0.0)  (-0.1, 0.0)  (0.8,-1.9) (1.6,-1.8) (2.4,-1.6) (3.2,-1.5) |
      | (0.8, 0.0) (0.0, 0.0)  (0.0, 0.0)  (0.0, 0.0)   (0.0, 0.0)  (0.0, 0.0) (0.8,-1.9) (1.7,-1.8) (2.5,-1.8) |
      | (0.9, 0.0) (0.0, 0.0)  (0.0, 0.0)  (0.0, 0.0)   (0.0, 0.0)  (0.0, 0.0) (0.0, 0.0) (0.8,-2.0) (1.7,-1.9) |
      | (0.9, 0.0) (0.0, 0.0)  (0.0, 0.0)  (0.0, 0.0)   (0.0, 0.0)  (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.8,-2.0) |
```

```
IPVT     =  (9, 9, 9, 9, 9, 9, 9, 9, 9)
INFO     =  0
```

Example 3

This example shows a factorization of a real general matrix ***A*** of order 9.

Call Statement and Input:

```
             M   N   A  LDA  IPVT  INFO
             |   |   |   |     |     |
CALL SGETRF( 9 , 9 , A,  9 , IPVT, INFO )
```

```
        | 1.0  1.0  1.0  1.0  0.0  0.0   0.0   0.0   0.0 |
        | 1.0  1.0  1.0  1.0  1.0  0.0   0.0   0.0   0.0 |
        | 4.0  1.0  1.0  1.0  1.0  1.0   0.0   0.0   0.0 |
        | 0.0  5.0  1.0  1.0  1.0  1.0   1.0   0.0   0.0 |
A    =  | 0.0  0.0  6.0  1.0  1.0  1.0   1.0   1.0   0.0 |
        | 0.0  0.0  0.0  7.0  1.0  1.0   1.0   1.0   1.0 |
        | 0.0  0.0  0.0  0.0  8.0  1.0   1.0   1.0   1.0 |
        | 0.0  0.0  0.0  0.0  0.0  9.0   1.0   1.0   1.0 |
        | 0.0  0.0  0.0  0.0  0.0  0.0  10.0  11.0  12.0 |
```

Output:

```
      | 4.0000  1.0000  1.0000  1.0000   1.0000   1.0000   0.0000   0.0000   0.0000 |
      | 0.0000  5.0000  1.0000  1.0000   1.0000   1.0000   1.0000   0.0000   0.0000 |
      | 0.0000  0.0000  6.0000  1.0000   1.0000   1.0000   1.0000   1.0000   0.0000 |
      | 0.0000  0.0000  0.0000  7.0000   1.0000   1.0000   1.0000   1.0000   1.0000 |
A   = | 0.0000  0.0000  0.0000  0.0000   8.0000   1.0000   1.0000   1.0000   1.0000 |
      | 0.0000  0.0000  0.0000  0.0000   0.0000   9.0000   1.0000   1.0000   1.0000 |
      | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000  10.0000  11.0000  12.0000 |
      | 0.2500  0.1500  0.1000  0.0714   0.0536  -0.0694  -0.0306   0.1806   0.3111 |
      | 0.2500  0.1500  0.1000  0.0714  -0.0714  -0.0556  -0.0194   0.9385  -0.0031 |
```

```
IPVT     =  (3, 4, 5, 6, 7, 8, 9, 8, 9)
```

SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)

Purpose

SGETRS and DGETRS solve one of the following systems of equations for multiple right-hand sides:

1. $\boldsymbol{AX} = \boldsymbol{B}$
2. $\boldsymbol{A}^{T}\boldsymbol{X} = \boldsymbol{B}$

CGETRS and ZGETRS solve one of the following systems of equations for multiple right-hand sides:

1. $\boldsymbol{AX} = \boldsymbol{B}$
2. $\boldsymbol{A}^{T}\boldsymbol{X} = \boldsymbol{B}$
3. $\boldsymbol{A}^{H}\boldsymbol{X} = \boldsymbol{B}$

In the formulas above:

- ***A*** represents the general matrix ***A*** containing the ***LU*** factorization.
- ***B*** represents the general matrix ***B*** containing the right-hand sides in its columns.
- ***X*** represents the general matrix ***B*** containing the solution vectors in its columns.

These subroutines use the results of the factorization of matrix ***A***, produced by a preceding call to SGETRF, DGETRF, CGETRF, or ZGETRF, respectively.

Table 126: Data Types.

A*, *B	**Subroutine**
Short-precision real	SGETRS$^{\triangle}$
Long-precision real	DGETRS$^{\triangle}$
Short-precision complex	CGETRS$^{\triangle}$
Long-precision complex	ZGETRS$^{\triangle}$
$^{\triangle}$ LAPACK	

Note: The input to these solve subroutines must be the output from the factorization subroutines SGETRF, DGETRF, CGETRF and ZGETRF, respectively.

Syntax

Table 127: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SGETRS | DGETRS | CGETRS | ZGETRS (*transa*, *n*, *nrhs*, *a*, *lda*, *ipvt*, *bx*, *ldb*, *info*)
C and C++	sgetrs | dgetrs | cgetrs | zgetrs (*transa*, *n*, *nrhs*, *a*, *lda*, *ipvt*, *bx*, *ldb*, *info*);
LAPACKE	info = LAPACKE_sgetrs | LAPACKE_dgetrs | LAPACKE_cgetrs | LAPACKE_zgetrs (*matrix_layout*, *transa*, *n*, *nrhs*, *a*, *lda*, *ipvt*, *bx*, *ldb*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

transa

indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation, resulting in solution 1.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation, resulting in solution 2.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation, resulting in solution 3.

Specified as: a single character; *transa* = 'N', 'T', or 'C'.

n

is the order of factored matrix ***A*** and the number of rows in matrix ***B***.

Specified as: an integer; $n \geq 0$.

nrhs

the number of right-hand sides—that is, the number of columns in matrix ***B*** used in the computation.

Specified as: an integer; $nrhs \geq 0$.

a

is the factorization of matrix ***A***, produced by a preceding call to SGETRF, DGETRF, CGETRF, or ZGETRF, respectively.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 126 on page 489.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

ipvt

is the integer vector ***ipvt*** of length *n*, containing the pivot indices produced by a preceding call to SGETRF, DGETRF, CGETRF, or ZGETRF, respectively.

Specified as: a one-dimensional array of (at least) length *n*, containing integers, where $1 \leq ipvt(i) \leq n$.

bx

is the general matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of matrix ***B***.

Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 126 on page 489

.

ldb

is the leading dimension of the array specified for ***B***.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

info

See On Return.

On Return

bx

is the matrix ***X***, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of ***X***.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 126 on page 489.

info

info has the following meaning:

If *info* = 0, the solve of general matrix ***A*** completed successfully.

Returned as:

- For SGETRS, DGETRS, CGETRS, and ZGETRS, returned as: an integer; info ≥ 0.
- For LAPACKE_sgetrs, LAPACKE_dgetrs, LAPACKE_cgetrs, and LAPACKE_zgetrs, returned as an integer function value; info ≥ 0.

Notes

1. In your C program, argument *info* must be passed by reference.
2. These subroutines accept lower case letters for the *transa* argument.
3. For SGETRS and DGETRS, if you specify 'C' for the *transa* argument, it is interpreted as though you specified 'T'.
4. The scalar data specified for input argument *n* must be the same for both _GETRF and _GETRS. In addition, the scalar data specified for input argument *m* in _GETRF **must be the same** as input argument *n* in both _GETRF and _GETRS.

   If, however, you do **not** plan to call _GETRS after calling _GETRF, then input arguments *m* and *n* in _GETRF do not need to be equal.
5. The array data specified for input arguments *a* and *ipvt* for these subroutines must be the same as the corresponding output arguments for SGETRF, DGETRF, CGETRF, and ZGETRF, respectively.
6. The matrices and vector used in this computation must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
7. On both input and output, matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ conform to LAPACK format.

Function

One of the following systems of equations is solved for multiple right-hand sides:

1. $\boldsymbol{AX} = \boldsymbol{B}$
2. $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{X} = \boldsymbol{B}$
3. $\boldsymbol{A}^{\mathrm{H}}\boldsymbol{X} = \boldsymbol{B}$ (only for CGETRS and ZGETRS)

where $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{X}$ are general matrices. These subroutines uses the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SGETRF, DGETRF, CGETRF or ZGETRF, respectively. For details on the factorization, see "SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)" on page 484.

If $n = 0$ or $nrhs = 0$, no computation is performed and the subroutine returns after doing some parameter checking. See references ["9" on page 1253, ["46" on page 1255], and ["75" on page 1257].

Error conditions

Computational Errors

None

Note: If the factorization performed by SGETRF, DGETRF, CGETRF or ZGETRF failed because a pivot element is zero, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. *transa* ≠ 'N', 'T', or 'C'
2. $n < 0$
3. $nrhs < 0$
4. $n > lda$
5. $lda \leq 0$
6. $n > ldb$
7. $ldb \leq 0$

Examples

Example 1

This example shows how to solve the system $\boldsymbol{AX} = \boldsymbol{B}$, where matrix $\boldsymbol{A}$ is the same matrix factored in the Example 1 for DGETRF.

Call Statement and Input:

```
          TRANSA   N  NRHS   A   LDA  IPIV   BX  LDB  INFO
            |      |    |    |    |     |    |    |     |
CALL DGETRS('N' ,  9 ,  5 , A , 9 , IPIV, BX , 9 , INFO)
```

IPVT = (9, 9, 9, 9, 9, 9, 9, 9, 9)
A = (same as output A in Example 1)

```
        | 93.0  186.0  279.0  372.0  465.0 |
        | 84.4  168.8  253.2  337.6  422.0 |
        | 76.6  153.2  229.8  306.4  383.0 |
        | 70.0  140.0  210.0  280.0  350.0 |
BX   =  | 65.0  130.0  195.0  260.0  325.0 |
        | 62.0  124.0  186.0  248.0  310.0 |
        | 61.4  122.8  184.2  245.6  307.0 |
        | 63.6  127.2  190.8  254.4  318.0 |
        | 69.0  138.0  207.0  276.0  345.0 |
```

Output:

```
        | 1.0   2.0   3.0   4.0   5.0 |
        | 2.0   4.0   6.0   8.0  10.0 |
        | 3.0   6.0   9.0  12.0  15.0 |
        | 4.0   8.0  12.0  16.0  20.0 |
BX   =  | 5.0  10.0  15.0  20.0  25.0 |
        | 6.0  12.0  18.0  24.0  30.0 |
        | 7.0  14.0  21.0  28.0  35.0 |
        | 8.0  16.0  24.0  32.0  40.0 |
        | 9.0  18.0  27.0  36.0  45.0 |
```

```
INFO =  0
```

Example 2

This example shows how to solve the system ***AX*** = ***b***, where matrix ***A*** is the same matrix factored in the Example 2 for ZGETRF.

Call Statement and Input:

```
          TRANS    N  NRHS   A   LDA  IPIV   BX  LDB  INFO
            |      |    |    |    |     |    |    |     |
CALL ZGETRS('N' ,  9 ,  5 , A , 9 , IPIV, BX , 9 , INFO)
```

IPVT = (9, 9, 9, 9, 9, 9, 9, 9, 9)
A = (same as output A in Example 2)

```
        | (193.0,-10.6)  (200.0, 21.8)  (207.0, 54.2)  (214.0, 86.6)  (221.0,119.0) |
        | (173.8, -9.4)  (178.8, 20.2)  (183.8, 49.8)  (188.8, 79.4)  (193.8,109.0) |
        | (156.2, -5.4)  (159.2, 22.2)  (162.2, 49.8)  (165.2, 77.4)  (168.2,105.0) |
        | (141.0,  1.4)  (142.0, 27.8)  (143.0, 54.2)  (144.0, 80.6)  (145.0,107.0) |
BX   =  | (129.0, 11.0)  (128.0, 37.0)  (127.0, 63.0)  (126.0, 89.0)  (125.0,115.0) |
        | (121.0, 23.4)  (118.0, 49.8)  (115.0, 76.2)  (112.0,102.6)  (109.0,129.0) |
        | (117.8, 38.6)  (112.8, 66.2)  (107.8, 93.8)  (102.8,121.4)   (97.8,149.0) |
        | (120.2, 56.6)  (113.2, 86.2)  (106.2,115.8)   (99.2,145.4)   (92.2,175.0) |
        | (129.0, 77.4)  (120.0,109.8)  (111.0,142.2)  (102.0,174.6)   (93.0,207.0) |
```

Output:

```
        | (1.0,1.0)  (1.0,2.0)  (1.0,3.0) (1.0,4.0) (1.0,5.0) |
        | (2.0,1.0)  (2.0,2.0)  (2.0,3.0) (2.0,4.0) (2.0,5.0) |
        | (3.0,1.0)  (3.0,2.0)  (3.0,3.0) (3.0,4.0) (3.0,5.0) |
        | (4.0,1.0)  (4.0,2.0)  (4.0,3.0) (4.0,4.0) (4.0,5.0) |
BX    = | (5.0,1.0)  (5.0,2.0)  (5.0,3.0) (5.0,4.0) (5.0,5.0) |
        | (6.0,1.0)  (6.0,2.0)  (6.0,3.0) (6.0,4.0) (6.0,5.0) |
        | (7.0,1.0)  (7.0,2.0)  (7.0,3.0) (7.0,4.0) (7.0,5.0) |
        | (8.0,1.0)  (8.0,2.0)  (8.0,3.0) (8.0,4.0) (8.0,5.0) |
        | (9.0,1.0)  (9.0,2.0)  (9.0,3.0) (9.0,4.0) (9.0,5.0) |
```

```
INFO  = 0
```

SGEF, DGEF, CGEF, and ZGEF (General Matrix Factorization)

Purpose

This subroutine factors a square general matrix ***A*** using Gaussian elimination with partial pivoting. To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGES/SGESM, DGES/DGESM, CGES/CGESM, or ZGES/ZGESM, respectively. To compute the inverse of matrix ***A***, follow the call to these subroutines with a call to SGEICD or DGEICD, respectively.

Table 128: Data Types.

Data Types	
A	**Subroutine**
Short-precision real	SGEF
Long-precision real	DGEF
Short-precision complex	CGEF
Long-precision complex	ZGEF

Note: The output from these factorization subroutines should be used only as input to the following subroutines for performing a solve or inverse: SGES/SGESM/SGEICD, DGES/DGESM/DGEICD, CGES/CGESM, and ZGES/ZGESM, respectively.

Syntax

Language	**Syntax**
Fortran	CALL SGEF | DGEF | CGEF | ZGEF (*a*, *lda*, *n*, *ipvt*)
C and C++	sgef | dgef | cgef | zgef (*a*, *lda*, *n*, *ipvt*);

On Entry

a

is the *n* by *n* general matrix ***A*** to be factored.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 128 on page 493.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

n
: is the order of matrix ***A***.

  Specified as: an integer; $0 \le n \le lda$.

ipvt
: See On Return.

On Return

a
: is the *n* by *n* transformed matrix ***A***, containing the results of the factorization. See "Function" on page 494. Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 128 on page 493.

ipvt
: is the integer vector ***ipvt*** of length *n*, containing the pivot indices. Returned as: a one-dimensional array of (at least) length *n*, containing integers.

Notes

1. Calling SGEFCD or DGEFCD with *iopt* = 0 is equivalent to calling SGEF or DGEF.
2. On both input and output, matrix ***A*** conforms to LAPACK format.

Function

The matrix ***A*** is factored using Gaussian elimination with partial pivoting (***ipvt***) to compute the ***LU*** factorization of ***A***, where (***A*** = ***PLU***):

L is a unit lower triangular matrix.
U is an upper triangular matrix.
P is the permutation matrix.

On output, the transformed matrix ***A*** contains ***U*** in the upper triangle and ***L*** in the strict lower triangle where ***ipvt*** contains the pivots representing permutation ***P***, such that ***A*** = ***PLU***.

If *n* is 0, no computation is performed. See references ["46" on page 1255] and ["48" on page 1255].

Error conditions

Resource Errors
: Unable to allocate internal work area.

Computational Errors
: Matrix ***A*** is singular.

  - One or more columns of ***L*** and the corresponding diagonal of ***U*** contain all zeros (all columns of ***L*** are checked). The first column, *i*, of ***L*** with a corresponding ***U*** = 0 diagonal element is identified in the computational error message.
  - The return code is set to 1.
  - *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2103 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

Input-Argument Errors

1. $lda \le 0$
2. $n < 0$
3. $n > lda$

Examples

Example 1

This example shows a factorization of a real general matrix ***A*** of order 9.

Call Statement and Input:

```
          A  LDA  N   IPVT
          |   |   |    |
CALL SGEF( A , 9 , 9 , IPVT )
```

```
        | 1.0  1.0  1.0  1.0  0.0  0.0   0.0   0.0   0.0 |
        | 1.0  1.0  1.0  1.0  1.0  0.0   0.0   0.0   0.0 |
        | 4.0  1.0  1.0  1.0  1.0  1.0   0.0   0.0   0.0 |
        | 0.0  5.0  1.0  1.0  1.0  1.0   1.0   0.0   0.0 |
A    =  | 0.0  0.0  6.0  1.0  1.0  1.0   1.0   1.0   0.0 |
        | 0.0  0.0  0.0  7.0  1.0  1.0   1.0   1.0   1.0 |
        | 0.0  0.0  0.0  0.0  8.0  1.0   1.0   1.0   1.0 |
        | 0.0  0.0  0.0  0.0  0.0  9.0   1.0   1.0   1.0 |
        | 0.0  0.0  0.0  0.0  0.0  0.0  10.0  11.0  12.0 |
```

Output:

```
        | 4.0000  1.0000  1.0000  1.0000   1.0000   1.0000   0.0000   0.0000   0.0000 |
        | 0.0000  5.0000  1.0000  1.0000   1.0000   1.0000   1.0000   0.0000   0.0000 |
        | 0.0000  0.0000  6.0000  1.0000   1.0000   1.0000   1.0000   1.0000   0.0000 |
        | 0.0000  0.0000  0.0000  7.0000   1.0000   1.0000   1.0000   1.0000   1.0000 |
A    =  | 0.0000  0.0000  0.0000  0.0000   8.0000   1.0000   1.0000   1.0000   1.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   9.0000   1.0000   1.0000   1.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000  10.0000  11.0000  12.0000 |
        | 0.2500  0.1500  0.1000  0.0714   0.0536  -0.0694  -0.0306   0.1806   0.3111 |
        | 0.2500  0.1500  0.1000  0.0714  -0.0714  -0.0556  -0.0194   0.9385  -0.0031 |
```

```
IPVT     =  (3, 4, 5, 6, 7, 8, 9, 8, 9)
```

Example 2

This example shows a factorization of a complex general matrix ***A*** of order 4.

Call Statement and Input:

```
          A  LDA  N   IPVT
          |   |   |    |
CALL CGEF( A , 4 , 4 , IPVT )


        | (1.0, 2.0) (1.0, 7.0) (2.0, 4.0) (3.0, 1.0) |
A    =  | (2.0, 0.0) (1.0, 3.0) (4.0, 4.0) (2.0, 3.0) |
        | (2.0, 1.0) (5.0, 0.0) (3.0, 6.0) (0.0, 0.0) |
        | (8.0, 5.0) (1.0, 9.0) (6.0, 6.0) (8.0, 1.0) |
```

Output:

```
        |  (8.0000, 5.0000)   (1.0000, 9.0000)  (6.0000, 6.0000)  (8.0000, 1.0000) |
A    =  |  (0.2022, 0.1236)   (1.9101, 5.0562)  (1.5281, 2.0449) (1.5056, -0.1910) |
        | (0.2360, -0.0225) (-0.0654, -0.9269) (-0.3462, 6.2692) (-1.6346, 1.3269) |
        | (0.1798, -0.1124)   (0.2462, 0.1308) (0.4412, -0.3655)  (0.2900, 2.3864) |
```

```
IPVT     =  (4, 4, 3, 4)
```

SGES, DGES, CGES, and ZGES (General Matrix, Its Transpose, or Its Conjugate Transpose Solve)

Purpose

These subroutines solve the system ***Ax*** = ***b*** for ***x***, where ***A*** is a general matrix and ***x*** and ***b*** are vectors. Using the *iopt* argument, they can also solve the real system $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$ or the complex system $\boldsymbol{A}^H\boldsymbol{x} = \boldsymbol{b}$ for ***x***. These subroutines use the results of the factorization of matrix ***A***, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively.

Table 129: Data Types.

Data Types	
A, ***b***, ***x***	**Subroutine**
Short-precision real	SGES
Long-precision real	DGES
Short-precision complex	CGES
Long-precision complex	ZGES

Note: The input to these solve subroutines must be the output from the factorization subroutines SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, and ZGEF, respectively.

Syntax

Language	Syntax
Fortran	CALL SGES | DGES | CGES | ZGES (*a*, *lda*, *n*, *ipvt*, *bx*, *iopt*)
C and C++	sges | dges | cges | zges (*a*, *lda*, *n*, *ipvt*, *bx*, *iopt*);

On Entry

a
: is the factorization of matrix ***A***, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 129 on page 496.

lda
: is the leading dimension of the array specified for *a*.

  Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

n
: is the order of matrix ***A***.

  Specified as: an integer; 0 ≤ *n* ≤ *lda*.

ipvt
: is the integer vector ***ipvt*** of length *n*, containing the pivot indices produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively.

  Specified as: a one-dimensional array of (at least) length *n*, containing integers.

bx
: is the vector ***b*** of length *n*, containing the right-hand side of the system.

  Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 129 on page 496.

iopt

determines the type of computation to be performed, where:

If *iopt* = 0, ***A*** is used in the computation.

If *iopt* = 1, $\boldsymbol{A}^T$ is used in SGES and DGES. $\boldsymbol{A}^H$ is used in CGES and ZGES.

Note: No data should be moved to form $\boldsymbol{A}^T$ or $\boldsymbol{A}^H$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: an integer; *iopt* = 0 or 1.

On Return

bx

is the solution vector ***x*** of length *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 129 on page 496.

Notes

1. The scalar data specified for input arguments *lda* and *n* for these subroutines must be the same as the corresponding input arguments specified for SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, and ZGEF, respectively.
2. The array data specified for input arguments *a* and *ipvt* for these subroutines must be the same as the corresponding output arguments for SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, and ZGEF, respectively.
3. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

The system ***Ax = b*** is solved for ***x***, where ***A*** is a general matrix and ***x*** and ***b*** are vectors. Using the *iopt* argument, this subroutine can also solve the real system $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$ or the complex system $\boldsymbol{A}^H\boldsymbol{x} = \boldsymbol{b}$ for ***x***. These subroutines use the results of the factorization of matrix ***A***, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively. For a description of how ***A*** is factored, see "SGEF, DGEF, CGEF, and ZGEF (General Matrix Factorization)" on page 493.

If *n* is 0, no computation is performed. See references ["46" on page 1255] and ["48" on page 1255].

Error conditions

Computational Errors

None

Note: If the factorization performed by SGEF, DGEF, CGEF, ZGEF, SGEFCD, DGEFCD, or DGEFP failed because a pivot element is zero, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. $lda \le 0$
2. $n < 0$
3. $n > lda$
4. $iopt \neq 0$ or 1

Examples

Example 1

Part 1

This part of the example shows how to solve the system ***Ax*** = ***b***, where matrix ***A*** is the same matrix factored in the Example 1 for SGEF and DGEF.

Call Statement and Input:

```
            A  LDA  N   IPVT   BX  IOPT
            |   |   |    |     |    |
CALL SGES( A , 9 , 9 , IPVT , BX , 0  )
```

IPVT = (3, 4, 5, 6, 7, 8, 9, 8, 9)
BX = (4.0, 5.0, 9.0, 10.0, 11.0, 12.0, 12.0, 12.0, 33.0)
A = (same as output A in Example 1)

Output:

```
BX       =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

Part 2

This part of the example shows how to solve the system $\boldsymbol{A}^{T}\boldsymbol{x} = \boldsymbol{b}$, where matrix $\boldsymbol{A}$ is the input matrix factored in Example 1 for SGEF and DGEF. Most of the input is the same in Part 2 as in Part 1.

Call Statement and Input:

```
            A  LDA  N   IPVT   BX  IOPT
            |   |   |    |     |    |
CALL SGES( A , 9 , 9 , IPVT , BX , 1  )
```

IPVT = (3, 4, 5, 6, 7, 8, 9, 8, 9)
BX = (6.0, 8.0, 10.0, 12.0, 13.0, 14.0, 15.0, 15.0, 15.0)
A = (same as output A in Example 1)

Output:

```
BX       =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

Example 2

Part 1

This part of the example shows how to solve the system $\boldsymbol{Ax} = \boldsymbol{b}$, where matrix $\boldsymbol{A}$ is the same matrix factored in the Example 2 for CGEF and ZGEF.

Call Statement and Input:

```
            A  LDA  N   IPVT   BX  IOPT
            |   |   |    |     |    |
CALL CGES( A , 4 , 4 , IPVT , BX , 0  )
```

IPVT = (4, 4, 3, 4)
BX = ((-10.0, 85.0), (-6.0, 61.0), (10.0, 38.0),
(58.0, 168.0))
A = (same as output A in Example 1)

Output:

```
BX       =  ((9.0, 0.0), (5.0, 1.0), (1.0, 6.0), (3.0, 4.0))
```

Part 2

This part of the example shows how to solve the system $\boldsymbol{A}^{H}\boldsymbol{x} = \boldsymbol{b}$, where matrix $\boldsymbol{A}$ is the input matrix factored in Example 2 for CGEF and ZGEF. Most of the input is the same in Part 2 as in Part 1.

Call Statement and Input:

```
            A  LDA  N   IPVT   BX  IOPT
            |   |   |    |     |    |
CALL CGES( A , 4 , 4 , IPVT , BX , 1  )
```

```
IPVT = (4, 4, 3, 4)
BX = ((71.0, 12.0), (61.0, -70.0), (123.0, -34.0),
(68.0, 7.0))
A  =  (same as output A in Example 1)
```

Output:

```
BX        =  ((9.0, 0.0), (5.0, 1.0), (1.0, 6.0), (3.0, 4.0))
```

SGESM, DGESM, CGESM, and ZGESM (General Matrix, Its Transpose, or Its Conjugate Transpose Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the following systems of equations for multiple right-hand sides, where ***A***, ***X***, and ***B*** are general matrices. SGESM and DGESM solve one of the following:

1. $\boldsymbol{AX} = \boldsymbol{B}$
2. $\boldsymbol{A}^{T}\boldsymbol{X} = \boldsymbol{B}$

CGESM and ZGESM solve one of the following:

1. $\boldsymbol{AX} = \boldsymbol{B}$
2. $\boldsymbol{A}^{T}\boldsymbol{X} = \boldsymbol{B}$
3. $\boldsymbol{A}^{H}\boldsymbol{X} = \boldsymbol{B}$

These subroutines use the results of the factorization of matrix ***A***, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively.

Table 130: Data Types.

Data Types	
A, ***B***, ***X***	**Subroutine**
Short-precision real	SGESM
Long-precision real	DGESM
Short-precision complex	CGESM
Long-precision complex	ZGESM

Note: The input to these solve subroutines must be the output from the factorization subroutines SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, and ZGEF, respectively.

Syntax

Language	Syntax
Fortran	CALL SGESM | DGESM | CGESM | ZGESM (*trans*, *a*, *lda*, *n*, *ipvt*, *bx*, *ldb*, *nrhs*)
C and C++	sgesm | dgesm | cgesm | zgesm (*trans*, *a*, *lda*, *n*, *ipvt*, *bx*, *ldb*, *nrhs*);

On Entry

trans

indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation, resulting in equation 1.

If *transa* = 'T', $\boldsymbol{A}^{T}$ is used in the computation, resulting in equation 2.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation, resulting in equation 3.

Specified as: a single character. It must be 'N', 'T', or 'C'.

a
is the factorization of matrix ***A***, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 130 on page 499.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

n
is the order of matrix ***A***.

Specified as: an integer; 0 ≤ *n* ≤ *lda*.

ipvt
is the integer vector ***ipvt*** of length *n*, containing the pivot indices produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively.

Specified as: a one-dimensional array of (at least) length *n*, containing integers.

bx
is the general matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of matrix ***B***.

Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 130 on page 499.

ldb
is the leading dimension of the array specified for *b*.

Specified as: an integer; *ldb* > 0 and *ldb* ≥ *n*.

nrhs
is the number of right-hand sides in the system to be solved.

Specified as: an integer; *nrhs* ≥ 0.

On Return

bx
is the matrix ***X***, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of ***X***.

Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 130 on page 499.

Notes

1. For SGESM and DGESM, if you specify 'C' for the *trans* argument, it is interpreted as though you specified 'T'.
2. The scalar data specified for input arguments *lda* and *n* for these subroutines must be the same as the corresponding input arguments specified for SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, and ZGEF, respectively.
3. The array data specified for input arguments *a* and *ipvt* for these subroutines must be the same as the corresponding output arguments for SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, and ZGEF, respectively.
4. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

One of the following systems of equations is solved for multiple right-hand sides:

1. $\boldsymbol{AX} = \boldsymbol{B}$
2. $\boldsymbol{A}^T\boldsymbol{X} = \boldsymbol{B}$
3. $\boldsymbol{A}^H\boldsymbol{X} = \boldsymbol{B}$ (only for CGESM and ZGESM)

where ***A***, ***B***, and ***X*** are general matrices. These subroutines use the results of the factorization of matrix ***A***, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively. For a description of how ***A*** is factored, see "SGEF, DGEF, CGEF, and ZGEF (General Matrix Factorization)" on page 493.

If *n* or *nrhs* is 0, no computation is performed. See references ["46" on page 1255] and ["48" on page 1255].

Error conditions

Computational Errors

None

Note: If the factorization performed by SGEF, DGEF, CGEF, ZGEF, SGEFCD, DGEFCD, or DGEFP failed because a pivot element is zero, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. *trans* ≠ 'N', 'T', or 'C'
2. *lda*, *ldb* ≤ 0
3. *n* < 0
4. *n* > *lda*, *ldb*
5. *nrhs* < 0

Examples

Example 1

Part 1

This part of the example shows how to solve the system ***AX*** = ***B*** for two right-hand sides, where matrix ***A*** is the same matrix factored in the Example 1 for SGEF and DGEF.

Call Statement and Input:

```
           TRANS  A  LDA  N   IPVT   BX  LDB  NRHS
             |    |   |   |    |      |   |     |
CALL SGESM( 'N' , A , 9 , 9 , IPVT , BX , 9 ,   2  )
```

```
IPVT = (3, 4, 5, 6, 7, 8, 9, 8, 9)
```
A = (same as output A in Example 1)

```
         ┌            ┐
         |  4.0   10.0 |
         |  5.0   15.0 |
         |  9.0   24.0 |
         | 10.0   35.0 |
BX   =   | 11.0   48.0 |
         | 12.0   63.0 |
         | 12.0   70.0 |
         | 12.0   78.0 |
         | 33.0  266.0 |
         └            ┘
```

Output:

```
         ┌         ┐
         | 1.0  1.0 |
         | 1.0  2.0 |
         | 1.0  3.0 |
         | 1.0  4.0 |
BX   =   | 1.0  5.0 |
         | 1.0  6.0 |
         | 1.0  7.0 |
         | 1.0  8.0 |
         | 1.0  9.0 |
                   ┘
```

Part 2

This part of the example shows how to solve the system $\boldsymbol{A}^T\boldsymbol{X} = \boldsymbol{B}$ for two right-hand sides, where matrix ***A*** is the input matrix factored in Example 1 for SGEF and DGEF.

Call Statement and Input:

```
            TRANS  A  LDA  N   IPVT   BX  LDB  NRHS
              |    |   |   |     |     |   |     |
CALL SGESM( 'T' , A , 9 , 9 , IPVT , BX , 9 ,   2  )
```

`IPVT = (3, 4, 5, 6, 7, 8, 9, 8, 9)`
A = (same as output A in Example 1)

```
         ┌             ┐
         |  6.0   15.0 |
         |  8.0   26.0 |
         | 10.0   40.0 |
         | 12.0   57.0 |
BX   =   | 13.0   76.0 |
         | 14.0   97.0 |
         | 15.0  120.0 |
         | 15.0  125.0 |
         | 15.0  129.0 |
                      ┘
```

Output:

```
         ┌         ┐
         | 1.0  1.0 |
         | 1.0  2.0 |
         | 1.0  3.0 |
         | 1.0  4.0 |
BX   =   | 1.0  5.0 |
         | 1.0  6.0 |
         | 1.0  7.0 |
         | 1.0  8.0 |
         | 1.0  9.0 |
                   ┘
```

Example 2

Part 1

This part of the example shows how to solve the system ***AX*** = ***B*** for two right-hand sides, where matrix ***A*** is the same matrix factored in the Example 2 for CGEF and ZGEF.

Call Statement and Input:

```
            TRANS  A  LDA  N   IPVT   BX  LDB  NRHS
              |    |   |   |    |      |   |     |
CALL CGESM( 'N' , A , 4 , 4 , IPVT , BX , 4 ,   2  )
```

`IPVT = (4, 4, 3, 4)`
A = (same as output A in Example 2)

```
         ┌                             ┐
         | (-10.0, 85.0)  (-11.0, 53.0) |
BX   =  |  (-6.0, 61.0)   (-6.0, 54.0) |
        |  (10.0, 38.0)    (2.0, 40.0) |
        | (58.0, 168.0)  (15.0, 105.0) |
                                      ┘
```

Output:

```
        ┌                       ┐
        | (9.0, 0.0)  (1.0, 1.0) |
BX   =  | (5.0, 1.0)  (2.0, 2.0) |
        | (1.0, 6.0)  (3.0, 3.0) |
        | (3.0, 4.0)  (4.0, 4.0) |
                                ┘
```

Part 2

This part of the example shows how to solve the system $\boldsymbol{A}^T\boldsymbol{X} = \boldsymbol{B}$ for two right-hand sides, where matrix ***A*** is the input matrix factored in Example 2 for CGEF and ZGEF.

Call Statement and Input:

```
            TRANS  A  LDA  N   IPVT   BX  LDB  NRHS
              |    |   |   |    |      |   |     |
CALL CGESM( 'T' , A , 4 , 4 , IPVT , BX , 4 ,   2  )
```

IPVT = (4, 4, 3, 4)
A = (same as output A in Example 2)

```
        ┌                               ┐
        |   (71.0, 12.0)   (18.0, 68.0) |
BX   =  |  (61.0, -70.0)  (-27.0, 71.0) |
        | (123.0, -34.0)  (-11.0, 97.0) |
        |    (68.0, 7.0)   (28.0, 50.0) |
                                       ┘
```

Output:

```
        ┌                       ┐
        | (9.0, 0.0)  (1.0, 1.0) |
BX   =  | (5.0, 1.0)  (2.0, 2.0) |
        | (1.0, 6.0)  (3.0, 3.0) |
        | (3.0, 4.0)  (4.0, 4.0) |
                                ┘
```

Part 3:

This part of the example shows how to solve the system $\boldsymbol{A}^H\boldsymbol{X} = \boldsymbol{B}$ for two right-hand sides, where matrix ***A*** is the input matrix factored in Example 2 for CGEF and ZGEF.

Call Statement and Input:

```
            TRANS  A  LDA  N   IPVT   BX  LDB  NRHS
              |    |   |   |    |      |   |     |
CALL CGESM( 'C' , A , 4 , 4 , IPVT , BX , 4 ,   2  )
```

IPVT = (4, 4, 3, 4)
A = (same as output A in Example 2)

```
        ┌                              ┐
        |  (58.0, -3.0)   (45.0, 20.0) |
BX   =  | (68.0, -31.0)  (83.0, -20.0) |
        | (89.0, -22.0)    (98.0, 1.0) |
        |  (53.0, 15.0)   (45.0, 25.0) |
                                      ┘
```

Output:

```
        ┌                       ┐
        | (1.0, 4.0)  (4.0, 5.0) |
BX   =  | (2.0, 3.0)  (3.0, 4.0) |
         | (3.0, 2.0)  (2.0, 3.0) |
         | (4.0, 1.0)  (1.0, 2.0) |
                                 ┘
```

SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)

Purpose

SGECON, DGECON, CGECON, and ZGECON estimate the reciprocal of the condition number of general matrix ***A***. These subroutines use the results of the factorization of matrix ***A*** produced by a preceding call to SGETRF, DGETRF, CGETRF, or ZGETRF, respectively. For details on the factorization, see "SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)" on page 484.

Table 131: Data Types.

Data Types		
A, *work*	*anorm*, *rcond*, *rwork*	**Subroutine**
Short-precision real	Short-precision real	SGECON^
Long-precision real	Long-precision real	DGECON^
Short-precision complex	Short-precision real	CGECON^
Long-precision complex	Long-precision real	ZGECON^

Syntax

Language	Syntax
Fortran	CALL SGECON | DGECON (*norm*, *n*, *a*, *lda*, *anorm*, *rcond*, *work*, *iwork*, *info*) CALL CGECON | ZGECON (*norm*, *n*, *a*, *lda*, *anorm*, *rcond*, *work*, *rwork*, *info*)

C and C++	sgecon | dgecon (*norm*, *n*, *a*, *lda*, *anorm*, *rcond*, *work*, *iwork*, *info*);
	cgecon | zgecon (*norm*, *n*, *a*, *lda*, *anorm*, *rcond*, *work*, *rwork*, *info*);

LAPACKE	info = LAPACKE_sgecon | LAPACKE_dgecon (*matrix_layout*, *norm*, *n*, *a*, *lda*, *anorm*, *rcond*);
	info = LAPACKE_cgecon | LAPACKE_zgecon (*matrix_layout*, *norm*, *n*, *a*, *lda*, *anorm*, *rcond*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

norm

specifies whether the estimate of the condition number is computed using the one norm or the infinity norm; where:

If *norm* = 'O' or '1', the one norm is used in the computation.

If *norm* = 'I', the infinity norm is used in the computation.

Specified as: a single character; *norm* = 'O', '1', or 'I'.

n
the order of the factored general matrix ***A*** used in the computation.

Specified as: an integer; $n \geq 0$.

a
is the general matrix ***A***, containing the factorization of matrix ***A*** produced by a preceding call to SGETRF, DGETRF, CGETRF, or ZGETRF, respectively.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 131 on page 504.

lda
is the leading dimension of matrix ***A***.

Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

anorm
has the following meaning:

If *norm* = 'O' or '1', then *anorm* is the one norm of the original matrix.

If *norm* = 'I', then *anorm* is the infinity norm of the original matrix.

Note: You may obtain the value of *anorm* by a preceding call to SLANGE, DLANGE, CLANGE, or ZLANGE, respectively. Refer to "SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)" on page 518.

Specified as: a number ≥ 0.0, of the data type indicated in Table 131 on page 504.

rcond
See On Return.

work
is the work area used by this subroutine, where:

For SGECON and DGECON
The size of *work* is (at least) of length 4*n*.

For CGECON and ZGECON
The size of *work* is (at least) of length 2*n*.

Specified as: an area of storage containing numbers of data type indicated in Table 131 on page 504.

iwork
is a work area used by this subroutine whose size is (at least) of length *n*.

Specified as: an area of storage containing integers.

rwork
is a work area used by this subroutine whose size is (at least) of length 2*n*.

Specified as: an area of storage containing numbers of the data type indicated in Table 131 on page 504.

info
See On Return.

On Return

rcond
has the following meaning:

If *info* = 0, an estimate of the reciprocal of the condition number of general matrix ***A*** is returned; i.e., *rcond* = 1.0/(NORM(A) ×NORM(A $^{-1}$)).

If *n* = 0, the subroutines return with *rcond* = 1.0.

If *n* ≠ 0 and *anorm* = 0.0, the subroutines return with *rcond* = 0.0.

Returned as: a number ≥ 0.0, of the data type indicated in Table 131 on page 504.

info

has the following meaning:

If *info* = 0, the computation completed normally.

Returned as:

- For SGECON, DGECON, CGECON, and ZGECON returned as: an integer; info ≥ 0.
- For LAPACKE_sgecon, LAPACKE_dgecon, LAPACKE_cgecon, and LAPACKE_zgecon returned as an integer function value; info ≥ 0.

Notes

1. In your C program, arguments *rcond* and *info* must be passed by reference.
2. This subroutine accepts lowercase letters for the *norm* argument.
3. The scalar data specified for input argument *n* must be the same for SLANGE/DLANGE/CLANGE/ZLANGE, SGETRF/DGETRF/CGETRF/ZGETRF, and SGECON/DGECON/CGECON/ZGECON. In addition, the scalar data specified for input argument *m* in SLANGE/DLANGE/CLANGE/ZLANGE and SGETRF/DGETRF/CGETRF/ZGETRF must be the same as input argument *n* in SLANGE/DLANGE/CLANGE/ZLANGE, SGETRF/DGETRF/CGETRF/ZGETRF, and SGECON/DGECON/CGECON/ZGECON.
4. The matrix ***A*** input to SLANGE/DLANGE/CLANGE/ZLANGE must be the same as the corresponding input argument for SGETRF/DGETRF/CGETRF/ZGETRF.
5. The matrix ***A*** input to SGECON/DGECON/CGECON/ZGECON must be the same as the corresponding output argument for SGETRF/DGETRF/CGETRF/ZGETRF.
6. On both input and output, matrix ***A*** conforms to LAPACK format.

Function

The reciprocal of the condition number of general matrix ***A*** is estimated, using the results of the factorization of matrix ***A*** produced by a preceding call of SGETRF, DGETRF, CGETRF, or ZGETRF.

rcond = 1.0/(NORM(A) × NORM(A^{-1})).

If *n* = 0, the subroutines return with *rcond* = 1.0.

If *n* ≠ 0 and *anorm* = 0.0, the subroutines return with *rcond* = 0.0.

See reference [“84” on page 1257].

Error conditions

Resource Errors

None.

Computational Errors

None.

Input-Argument Errors

1. *norm* ≠ 'O', '1', or 'I'
2. *n* < 0
3. *n* > *lda*
4. *lda* ≤ 0
5. *anorm* < 0
6. *anorm* ≠ 0 and *anorm* > *big* or *anorm* < *tiny*

   Where:

   For SGECON and CGECON

   big and *tiny* have the following values:

$big = 2^{127} \times (1 - ULP)$
$tiny = 2^{-126} \times (2^{21})$

For DGECON
big and *tiny* have the following values:

$big = 2^{1023} \times (1 - ULP)$
$tiny = 2^{-1022} \times (2^{49})$

For ZGECON
big and *tiny* have the following values:

$big = 2^{1023} \times (1 - ULP)$
$tiny = 2^{-1022} \times (2^{50})$

Where ULP = unit in last place.

Note: To avoid this error, scale matrix ***A*** so that *tiny* ≤ *anorm* ≤ *big*.

Examples

Example 1

This example estimates the reciprocal of the condition number of real general matrix ***A***. The input matrix ***A*** to DLANGE and DGETRF is the same as input matrix ***A*** in Example 3.

Call Statements and Input:

```
                 NORM  M   N   A  LDA  WORK
                   |   |   |   |   |     |
ANORM = DLANGE(  '1', 9 , 9 , A , 9 , WORK )
```

```
              M   N   A  LDA  IPVT  INFO
              |   |   |   |     |     |
CALL DGETRF(  9 , 9 , A , 9 , IPVT, INFO )
```

```
            NORM  N   A  LDA  ANORM  RCOND  WORK  IWORK  INFO
              |   |   |   |     |      |      |     |      |
CALL DGECON( '1', 9 , A , 9 , ANORM, RCOND, WORK, IWORK, INFO )
```

A = (same as output A in Example 3)

ANORM = (same as output ANORM in Example 1)

Output:

RCOND = 5.44×10^{-5}

INFO = 0

Example 2

This example estimates the reciprocal of the condition number of complex general matrix ***A***. The input matrix ***A*** to ZLANGE and ZGETRF is the same as input matrix ***A*** in Example 2.

Call Statements and Input:

```
                 NORM  M   N   A  LDA  WORK
```

```
                  |   |   |   |   |      |
ANORM = ZLANGE(  '1', 4 , 4 , A , 4 , RWORK )
```

```
              M   N   A  LDA  IPVT  INFO
              |   |   |   |     |     |
CALL ZGETRF(  4 , 4 , A , 4 , IPVT, INFO )
```

```
            NORM  N   A  LDA  ANORM  RCOND  WORK  RWORK  INFO
              |   |   |   |     |      |      |     |      |
CALL ZGECON( '1', 4 , A , 4 , ANORM, RCOND, WORK, RWORK, INFO )
```

A = (same as output A in Example 2)

ANORM = (same as output ANORM in Example 2)

Output:

RCOND = 3.66×10^{-2}

INFO = 0

SGEFCD and DGEFCD (General Matrix Factorization, Condition Number Reciprocal, and Determinant)

Purpose

These subroutines factor general matrix ***A*** using Gaussian elimination. An estimate of the reciprocal of the condition number and the determinant of matrix ***A*** can also be computed. To solve a system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGES/SGESM or DGES/DGESM, respectively. To compute the inverse of matrix ***A***, follow the call to these subroutines with a call to SGEICD and DGEICD, respectively.

Table 132: Data Types.

Data Types	
A, *aux*, *rcond*, ***det***	**Subroutine**
Short-precision real	SGEFCD
Long-precision real	DGEFCD

Note: The output from these factorization subroutines should be used only as input to the following subroutines for performing a solve or inverse: SGES/SGESM/SGEICD and DGES/DGESM/DGEICD, respectively.

Syntax

Language	**Syntax**
Fortran	CALL SGEFCD | DGEFCD (*a*, *lda*, *n*, *ipvt*, *iopt*, *rcond*, *det*, *aux*, *naux*)
C and C++	sgefcd | dgefcd (*a*, *lda*, *n*, *ipvt*, *iopt*, *rcond*, *det*, *aux*, *naux*);

On Entry

a
is a general matrix ***A*** of order *n*, whose factorization, reciprocal of condition number, and determinant are computed. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 132 on page 508.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

n
is the order of matrix ***A***.

Specified as: an integer; 0 ≤ *n* ≤ *lda*.

ipvt
See On Return.

iopt
indicates the type of computation to be performed, where:

If *iopt* = 0, the matrix is factored.

If *iopt* = 1, the matrix is factored, and the reciprocal of the condition number is computed.

If *iopt* = 2, the matrix is factored, and the determinant is computed.

If *iopt* = 3, the matrix is factored, and the reciprocal of the condition number and the determinant are computed.

Specified as: an integer; *iopt* = 0, 1, 2, or 3.

rcond
See On Return.

det
See On Return.

aux
has the following meaning:

If *naux = 0* and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is is a storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 132 on page 508.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SGEFCD and DGEFCD dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux* ≥ *n*.

On Return

a
is the transformed matrix ***A*** of order *n*, containing the results of the factorization. See "Function" on page 510. Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 132 on page 508.

ipvt
is the integer vector ***ipvt*** of length *n*, containing the pivot indices. Returned as: a one-dimensional array of (at least) length *n*, containing integers.

rcond
is an estimate of the reciprocal of the condition number, *rcond*, of matrix ***A***. Returned as: a number of the data type indicated in Table 132 on page 508; *rcond* ≥ 0.

det
is the vector ***det***, containing the two components, det_1 and det_2, of the determinant of matrix ***A***. The determinant is:

$$det_1\left(10^{det_2}\right)$$

where $1 \le det_1 < 10$. Returned as: an array of length 2, containing numbers of the data type indicated in Table 132 on page 508.

Notes

1. In your C program, argument *rcond* must be passed by reference.
2. When *iopt* = 0, these subroutines provide the same function as a call to SGEF or DGEF, respectively.
3. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 49.
4. On both input and output, matrix **A** conforms to LAPACK format.

Function

Matrix ***A*** is factored using Gaussian elimination with partial pivoting (**ipvt**) to compute the ***LU*** factorization of ***A***, where (***A=PLU***):

L is a unit lower triangular matrix.
U is an upper triangular matrix.
P is the permutation matrix.

On output, the transformed matrix ***A*** contains ***U*** in the upper triangle and ***L*** in the strict lower triangle where ***ipvt*** contains the pivots representing permutation ***P***, such that ***A*** = ***PLU***.

An estimate of the reciprocal of the condition number, *rcond*, and the determinant, ***det***, can also be computed by this subroutine. The estimate of the condition number uses an enhanced version of the algorithm described in references [“83” on page 1257] and [“84” on page 1257].

If *n* is 0, no computation is performed. See reference [“46” on page 1255].

These subroutines call SGEF and DGEF, respectively, to perform the factorization. ***ipvt*** is an output vector of SGEF and DGEF. It is returned for use by SGES/SGESM and DGES/DGESM, the solve subroutines.

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors
Matrix ***A*** is singular.

- If your program is not terminated by SGEF and DGEF, then SGEFCD and DGEFCD, respectively, return 0 for *rcond* and *det*.
- One or more columns of ***L*** and the corresponding diagonal of ***U*** contain all zeros (all columns of ***L*** are checked). The first column, *i*, of ***L*** with a corresponding ***U*** = 0 diagonal element is identified in the computational error message, issued by SGEF or DGEF, respectively.

- *i* can be determined at run time by using the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2103 in the ESSL error option table; otherwise, the default value causes your program to be terminated by SGEF or DGEF, respectively, when this error occurs. If your program is not terminated by SGEF or DGEF, respectively, the return code is set to 2. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

Input-Argument Errors

1. $lda \leq 0$
2. $n < 0$
3. $n > lda$
4. $iopt \neq 0, 1, 2$, or 3
5. Error 2015 is recoverable or $naux \neq 0$, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example

This example shows a factorization of matrix ***A*** of order 9. The input is the same as used in SGEF and DGEF. See Example 1. The reciprocal of the condition number and the determinant of matrix ***A*** are also computed. The values used to estimate the reciprocal of the condition number in this example are obtained with the following values:

$\|\boldsymbol{A}\|_1$ = max(6.0, 8.0, 10.0, 12.0, 13.0, 14.0, 15.0, 15.0, 15.0) = 15.0

Estimate of $\|\boldsymbol{A}^{-1}\|_1$ = 1091.87

This estimate is equal to the actual *rcond* of 5.436(10^{-5}), which is computed by SGEICD and DGEICD. (See Example 3.) On output, the value in ***det***, |***A***|, is equal to 336.

Call Statement and Input:

```
              A  LDA  N   IPVT  IOPT  RCOND   DET   AUX  NAUX
              |   |   |    |     |     |       |     |    |
CALL DGEFCD( A , 9 , 9 , IPVT , 3  , RCOND , DET , AUX , 9  )
```

A =(same as input A in
Example 1)

Output:

A = (same as output A in Example 1)

```
IPVT = (3, 4, 5, 6, 7, 8, 9, 8, 9)
RCOND = 0.00005436
DET = (3.36, 2.00)
```

SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)

Purpose

These subroutines find the inverse of general matrix ***A***.

Subroutines SGEICD and DGEICD also find the reciprocal of the condition number and the determinant of general matrix ***A***.

Table 133: Data Types.

A*, *aux*, *rcond*, *det*, *work	**Subroutine**
Short-precision real	SGETRI^ and SGEICD
Long-precision real	DGETRI^ and DGEICD
Short-precision complex	CGETRI^
Long-precision complex	ZGETRI^
^LAPACK	

Note: The input to SGETRI, DGETRI, CGETRI, and ZGETRI must be the output from the factorization subroutines SGETRF, DGETRF, CGETRF, and ZGETRF, respectively.

If you call subroutines SGEICD and DGEICD with *iopt* = 4, the input must be the output from the factorization subroutines SGEF/SGEFCD/SGETRF or DGEF/DGEFCD/DGEFP/DGETRF, respectively.

Syntax

Table 134: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SGETRI | DGETRI | CGETRI | ZGETRI (*n*, *a*, *lda*, *ipvt*, *work*, *lwork*, *info*) CALL SGEICD | DGEICD (*a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*)

C and C++	sgetri | dgetri | cgetri | zgetri (*n*, *a*, *lda*, *ipvt*, *work*, *lwork*, *info*);
	sgeicd | dgeicd (*a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*);

LAPACKE	info = LAPACKE_sgetri | LAPACKE_dgetri | LAPACKE_cgetri | LAPACKE_zgetri (*matrix_layout*, *n*, *a*, *lda*, *ipvt*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

a

has the following meaning, where:

For subroutines SGETRI, DGETRI, CGETRI, and ZGETRI:

It is the transformed matrix ***A*** of order *n*, resulting from the factorization performed in a previous call to SGETRF, DGETRF, CGETRF, or ZGETRF, respectively, whose inverse is computed.

For subroutines SGEICD and DGEICD:

If *iopt* = 0, 1, 2, or 3, it is matrix ***A*** of order *n*, whose inverse, reciprocal of condition number, and determinant are computed.

If *iopt* = 4, it is the transformed matrix ***A*** of order *n*, resulting from the factorization performed in a previous call to SGEF/SGEFCD or DGEF/DGEFCD/DGEFP, respectively, whose inverse is computed.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 133 on page 512.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

n
is the order of matrix ***A***.

Specified as: an integer; 0 ≤ *n* ≤ *lda*.

iopt
indicates the type of computation to be performed, where:

If *iopt* = 0, the inverse is computed for matrix ***A***.

If *iopt* = 1, the inverse and the reciprocal of the condition number are computed for matrix ***A***.

If *iopt* = 2, the inverse and the determinant are computed for matrix ***A***.

If *iopt* = 3, the inverse, the reciprocal of the condition number, and the determinant are computed for matrix ***A***.

If *iopt* = 4, the inverse is computed using the factored matrix ***A***.

Specified as: an integer; *iopt* = 0, 1, 2, 3, 4.

rcond
See On Return.

det
See On Return.

aux
has the following meaning, and its size is specified by *naux*:

If *iopt* = 0, 1, 2, or 3, then if *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored. Otherwise, it is the storage work area used by this subroutine.

If *iopt* = 4, *aux* has the following meaning:

- For SGEICD, the first *n* (32-bit integer arguments) or 2*n* (64-bit integer arguments) locations in *aux* must contain the ***ipvt*** integer vector of length *n*, resulting from a previous call to SGEF, SGETRF, or SGEFCD.
- For DGEICD, the first ceiling(*n*/2) (32-bit integer arguments) or *n* (64-bit integer arguments) locations in *aux* must contain the ***ipvt*** integer vector of length *n*, resulting from a previous call to DGEF, DGETRF, DGEFCD, or DGEFP.

Specified as: an area of storage, containing numbers of the data type indicated in Table 133 on page 512.

naux
is the size of the work area specified by *aux*; that is, the number of elements in *aux*.

Specified as: an integer, where:

If *iopt* ≠ 4, then if *naux* = 0 and error 2015 is unrecoverable, SGEICD and DGEICD dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux* ≥ 100*n*.

ipvt
is the integer vector *ipvt* of length *n*, containing the pivot indices resulting from a previous call to SGETRF, DGETRF, CGETRF, or ZGETRF.

Specified as: a one-dimensional array of (at least) length *n*, containing integers, where 1 ≤ *ipvt(i)* ≤ *n*.

work
has the following meaning:

If *lwork* = 0, *work* is ignored.

If *lwork* ≠ 0, *work* is the work area used by this subroutine, where:

- If *lwork* ≠ -1, its size is (at least) of length *lwork*.
- If *lwork* = -1, its size is (at least) of length 1.

Specified as: an area of storage containing numbers of data type indicated in Table 133 on page 512.

lwork
is the number of elements in array `WORK`.

Specified as: an integer; where:

- If *lwork* = 0, SGETRI/DGETRI/CGETRI/ZGETRI dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program. This option is an extension to the LAPACK standard.
- If *lwork* = -1, SGETRI/DGETRI/CGETRI/ZGETRI performs a work area query and returns the optimal size of *work* in $work_1$. No computation is performed and the subroutine returns after error checking is complete.
- Otherwise, it must be:

  lwork ≥ max(1, *n*)
- For optimal performance, *lwork* ≥ 100**n*.

info
See On Return.

On Return

a
is the resulting inverse of matrix ***A*** of order *n*. Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 133 on page 512.

rcond
is the reciprocal of the condition number, *rcond*, of matrix ***A***. Returned as: a real number of the data type indicated in Table 133 on page 512; *rcond* ≥ 0.

det
is the vector ***det***, containing the two components det_1 and det_2 of the determinant of matrix ***A***. The determinant is:

$$det_1\left(10^{det_2}\right)$$

where $1 \leq det_1 < 10$. Returned as: an array of length 2, containing numbers of the data type indicated in Table 133 on page 512.

work
is the work area used by this subroutine if *lwork* ≠ 0, where:

If *lwork* ≠ 0 and *lwork* ≠ -1, its size is (at least) of length *lwork*.

If *lwork* = -1, its size is (at least) of length 1.

Returned as: an area of storage, where:

If *lwork* ≥ 1 or *lwork* = -1, then $work_1$ is set to the optimal *lwork* value and contains numbers of the data type indicated in Table 133 on page 512. Except for $work_1$, the contents of *work* are overwritten on return.

info
has the following meaning:

If *info* = 0, the inverse completed successfully.

If *info* > 0, *info* is set equal to the first *i* where $\boldsymbol{U}_{ii}$ is exactly zero. The matrix is singular, and its inverse could not be computed.

Returned as:

- For SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD returned as: an integer; info ≥ 0.
- For LAPACKE_sgetri, LAPACKE_dgetri, LAPACKE_cgetri, and LAPACKE_zgetri, returned as an integer function value; info ≥ 0.

Notes

1. In your C program, arguments *rcond* and *info* must be passed by reference.
2. The input scalar arguments for SGETRI, DGETRI, CGETRI, and ZGETRI must be set to the same values as the corresponding input arguments in the previous call to SGETRF, DGETRF, CGETRF, and ZGETRF, respectively.

   If *iopt* = 4, the input scalar arguments for SGEICD and DGEICD must be set to the same values as the corresponding input arguments in the previous call to SGEF/SGEFCD or DGEF/DGEFCD/DGEFP, respectively.
3. You have the option of having the value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.
4. The way _GETRI subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the *info* argument to provide information about the computational error, but they also provide an error message.
5. On both input and output, matrix ***A*** conforms to LAPACK format.
6. For best performance, specify *lwork* = 0.

Function

These subroutines compute the inverse of general square matrix ***A***, where:

- $\boldsymbol{A}^{-1}$ is the inverse of matrix ***A***, where $\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}$, and ***I*** is the identity matrix.

Additionally, the subroutines SGEICD and DGEICD compute the reciprocal of the condition number and the determinant of a general square matrix ***A***, using partial pivoting to preserve accuracy, where:

- $1/(\|\boldsymbol{A}\|_1)(\|\boldsymbol{A}^{-1}\|_1)$ is the reciprocal of the condition number, where $\|\boldsymbol{A}\|_1$ is the one-norm of matrix ***A***.
- $|\boldsymbol{A}|$ is the determinant of matrix ***A***, where $|\boldsymbol{A}|$ is expressed as:

$$det_1\left(10^{det_2}\right)$$

The *iopt* argument is used to determine the combination of output items produced by SGEICD and DGEICD: the inverse, the reciprocal of the condition number, and the determinant.

If *n* is 0, no computation is performed. See references ["46" on page 1255], ["48" on page 1255], and ["54" on page 1256].

Error conditions

Resource Errors

1. Unable to allocate internal work area.
2. If *iopt* = 0, 1, 2, or 3, then error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors

Matrix ***A*** is singular or nearly singular.

For SGETRI, DGETRI, CGETRI, and ZGETRI:

- The index *i* of the first pivot element having a value equal to zero is identified in the computational error message.
- The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2149 is set to be unlimited in the ESSL error option table.

For SGEICD and DGEICD:

- The index *i* of the first pivot element having a value equal to 0 is identified in the computational error message.
- These subroutines return 0 for *rcond* and ***det***, if you requested them.
- The return code is set to 2.
- *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2105 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

Input-Argument Errors

1. $lda \le 0$
2. $n < 0$
3. $n > lda$
4. $iopt \ne 0, 1, 2, 3,$ or 4
5. $lwork \ne 0$, $lwork \ne -1$, and $lwork < \max(1, n)$
6. Error 2015 is recoverable or $naux \ne 0$, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example computes the inverse of matrix ***A***, where matrix ***A*** is the transformed matrix factored by SGETRF in Example 3 and the input contents of IPVT are the same as the output contents of IPVT in Example 3.

Note: Because *lwork* is 0, SGETRI dynamically allocates the work area used by this subroutine.

Call Statement and Input:

```
              N   A  LDA  IPVT   WORK  LWORK  INFO
              |   |   |    |      |      |     |
CALL SGETRI( 9 , A , 9 , IPVT , WORK ,  0   , INFO )
```

A = (same as output A in Example 3)
IPVT = (same as output IPVT in Example 3)

Output:

```
        |    0.333   -0.667   0.333  0.000  0.000  0.000   0.042 -0.042  0.000 |
        |   56.833  -52.167  -1.167 -0.500 -0.500 -0.357   6.836 -0.479 -0.500 |
        |  -55.167   51.833   0.833  0.500  0.500  0.214  -6.735  0.521  0.500 |
        |   -1.000    1.000   0.000  0.000  0.000  0.143  -0.143  0.000  0.000 |
A     = |   -1.000    1.000   0.000  0.000  0.000  0.000   0.000  0.000  0.000 |
        |   -1.000    1.000   0.000  0.000  0.000  0.000  -0.125  0.125  0.000 |
        | -226.000  206.000   5.000  3.000  2.000  1.429 -27.179  1.750  2.000 |
        |  560.000 -520.000 -10.000 -6.000 -4.000 -2.857  67.857 -5.000 -5.000 |
        | -325.000  305.000   5.000  3.000  2.000  1.429 -39.554  3.125  3.000 |
```

```
INFO =       0
```

Example 2

This example computes the inverse of matrix ***A***, where ***A*** is the transformed matrix factored by ZGETRF in Example 2 and the input contents of IPVT are the same as the output contents of IPVT in Example 2.

Note: Because *lwork* is 0, ZGETRI dynamically allocates the work area used by this subroutine.

Call Statement and Input:

```
             N   A  LDA  IPVT   WORK  LWORK  INFO
             |   |   |    |      |      |     |
CALL ZGETRI( 9 , A , 9 , IPVT , WORK ,  0   , INFO )
```

A = (same as output A in Example 2)
IPVT = (same as output IPVT in Example 2)

Output:

```
A     = | (-0.2, -0.4) (-0.1,  0.1) (-0.1,  0.1) ( 0.0,  0.1) ( 0.1,  0.1) ( 0.1,  0.1) ( 0.1,  0.0) ( 0.1,  0.0) ( 0.0, -0.3) |
        | ( 0.2,  0.4) ( 0.0, -0.6) (-0.1,  0.0) (-0.1,  0.0) (-0.1,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.1,  0.0) |
        | ( 0.0,  0.0) ( 0.2,  0.4) ( 0.0, -0.6) (-0.1,  0.0) (-0.1,  0.0) (-0.1,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.1,  0.0) |
        | ( 0.0,  0.0) ( 0.0,  0.0) ( 0.2,  0.4) ( 0.0, -0.6) (-0.1,  0.0) (-0.1,  0.0) (-0.1,  0.0) ( 0.0,  0.0) ( 0.1,  0.1) |
        | ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.2,  0.4) ( 0.0, -0.6) (-0.1,  0.0) (-0.1,  0.0) (-0.1,  0.0) ( 0.1,  0.1) |
        | ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.2,  0.4) ( 0.0, -0.6) (-0.1,  0.0) (-0.1,  0.0) ( 0.0,  0.1) |
        | ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.2,  0.4) ( 0.0, -0.6) (-0.1,  0.0) (-0.1,  0.1) |
        | ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.2,  0.4) ( 0.0, -0.6) (-0.1,  0.1) |
        | ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.0,  0.0) ( 0.2,  0.4) (-0.2, -0.4) |
```

```
INFO =       0
```

Example 3

This example computes the inverse, the reciprocal of the condition number, and the determinant of matrix ***A***. The values used to compute the reciprocal of the condition number in this example are obtained with the following values:

$\|\boldsymbol{A}\|_1$ = max(6.0, 8.0, 10.0, 12.0, 13.0, 14.0, 15.0, 15.0, 15.0) = 15.0
$\|\boldsymbol{A}^{-1}\|_1$ = 1226.33

On output, the value in ***det***, |***A***|, is equal to 336.

Call Statement and Input:

```
             A  LDA  N  IOPT  RCOND   DET   AUX   NAUX
             |   |   |   |      |      |     |     |
CALL DGEICD( A , 9 , 9 , 3  , RCOND , DET , AUX , 293 )


        | 1.0  1.0  1.0  1.0  0.0  0.0   0.0   0.0   0.0 |
        | 1.0  1.0  1.0  1.0  1.0  0.0   0.0   0.0   0.0 |
        | 4.0  1.0  1.0  1.0  1.0  1.0   0.0   0.0   0.0 |
        | 0.0  5.0  1.0  1.0  1.0  1.0   1.0   0.0   0.0 |
A    =  | 0.0  0.0  6.0  1.0  1.0  1.0   1.0   1.0   0.0 |
        | 0.0  0.0  0.0  7.0  1.0  1.0   1.0   1.0   1.0 |
        | 0.0  0.0  0.0  0.0  8.0  1.0   1.0   1.0   1.0 |
        | 0.0  0.0  0.0  0.0  0.0  9.0   1.0   1.0   1.0 |
        | 0.0  0.0  0.0  0.0  0.0  0.0  10.0  11.0  12.0 |
```

Output:

```
        |    0.333   -0.667   0.333  0.000  0.000  0.000   0.042 -0.042  0.000 |
        |   56.833  -52.167  -1.167 -0.500 -0.500 -0.357   6.836 -0.479 -0.500 |
        |  -55.167   51.833   0.833  0.500  0.500  0.214  -6.735  0.521  0.500 |
        |   -1.000    1.000   0.000  0.000  0.000  0.143  -0.143  0.000  0.000 |
A    =  |   -1.000    1.000   0.000  0.000  0.000  0.000   0.000  0.000  0.000 |
        |   -1.000    1.000   0.000  0.000  0.000  0.000  -0.125  0.125  0.000 |
        | -226.000  206.000   5.000  3.000  2.000  1.429 -27.179  1.750  2.000 |
        |  560.000 -520.000 -10.000 -6.000 -4.000 -2.857  67.857 -5.000 -5.000 |
        | -325.000  305.000   5.000  3.000  2.000  1.429 -39.554  3.125  3.000 |


RCOND    =  0.00005436
DET      =  (3.36, 2.00)
```

Example 4

This example computes the inverse of matrix ***A***, where: *iopt* = 4; matrix ***A*** is the transformed matrix factored by SGEF in Example 1; and the input contents of AUX are the same as the output contents of IPVT in Example 1.

Call Statement and Input:

```
             A  LDA  N  IOPT  RCOND   DET   AUX   NAUX
             |   |   |   |      |      |     |     |
CALL SGEICD( A , 9 , 9 , 4  , RCOND , DET , AUX , 300 )
```

A = (same as output A in Example 1)
AUX = (same as output IPVT in Example 1)

Output:

```
        |    0.333   -0.667   0.333  0.000  0.000  0.000   0.042 -0.042  0.000 |
        |   56.833  -52.167  -1.167 -0.500 -0.500 -0.357   6.836 -0.479 -0.500 |
        |  -55.167   51.833   0.833  0.500  0.500  0.214  -6.735  0.521  0.500 |
        |   -1.000    1.000   0.000  0.000  0.000  0.143  -0.143  0.000  0.000 |
A    =  |   -1.000    1.000   0.000  0.000  0.000  0.000   0.000  0.000  0.000 |
        |   -1.000    1.000   0.000  0.000  0.000  0.000  -0.125  0.125  0.000 |
        | -226.000  206.000   5.000  3.000  2.000  1.429 -27.179  1.750  2.000 |
        |  560.000 -520.000 -10.000 -6.000 -4.000 -2.857  67.857 -5.000 -5.000 |
        | -325.000  305.000   5.000  3.000  2.000  1.429 -39.554  3.125  3.000 |
```

SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)

Purpose

SLANGE, DLANGE, CLANGE, and ZLANGE compute the norm of general matrix ***A***.

Table 135: Data Types.

Data Types		
A	*work*, **Result**	**Subprogram**
Short-precision real	Short-precision real	SLANGE$^{\triangle}$
Long-precision real	Long-precision real	DLANGE$^{\triangle}$
Short-precision complex	Short-precision real	CLANGE$^{\triangle}$
Long-precision complex	Long-precision real	ZLANGE$^{\triangle}$

Syntax

Language	**Syntax**
Fortran	SLANGE | DLANGE | CLANGE | ZLANGE (*norm*, *m*, *n*, *a*, *lda*, *work*)
C and C++	slange | dlange | clange | zlange (*norm*, *m*, *n*, *a*, *lda*, *work*);
LAPACKE	LAPACKE_slange | LAPACKE_dlange | LAPACKE_clange | LAPACKE_zlange (*matrix_layout*, *norm*, *m*, *n*, *a*, *lda*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

norm

specifies the type of computation, where:

If *norm* = 'O' or '1', the one norm of ***A*** is computed.

If *norm* = 'I', the infinity norm of ***A*** is computed.

If *norm* = 'F' or 'E', the Frobenius or Euclidean norm of ***A*** is computed.

If *norm* = 'M', the absolute value of the matrix element having the largest absolute value, i.e., max (|***A***|), is returned.

Specified as: a single character; *norm* = 'O', '1', 'I', 'F', 'E', or 'M'.

m

the number of rows in matrix ***A***.

Specified as: an integer; $m \geq 0$.

n

the number of columns in matrix ***A***.

Specified as: an integer; $n \geq 0$.

a

is the general matrix ***A***, with *m* rows and *n* columns.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 135 on page 519.

lda

is the leading dimension of matrix ***A***.

Specified as: an integer; *lda* > 0 and *lda* ≥ *m*.

work
is the work area used by this subroutine, where:

- When *norm* = 'I', the size of *work* is (at least) of length *m*.
- Otherwise, *work* is not referenced.

Specified as: an area of storage containing numbers of data type indicated in Table 135 on page 519.

On Return

Function value
is the result of the norm computation, returned as a number of the data type indicated in Table 135 on page 519.

If *norm* = 'O' or '1', the one norm of ***A*** is returned.

If *norm* = 'I', the infinity norm of ***A*** is returned.

If *norm* = 'F' or 'E', the Frobenius or Euclidean norm of ***A*** is returned.

If *norm* = 'M', the absolute value of the matrix element having the largest absolute value, i.e., max (|***A***|), is returned.

If *m* = 0 or *n* = 0, the function returns zero.

Notes

1. Declare this function in your program as returning a value of the data type indicated in Table 135 on page 519.
2. This function accepts lowercase letters for the *norm* argument.

Function

One of the following computations is performed on general matrix ***A***, depending on the value specified for *norm*:

Value specified for *norm*	**Type of computation performed**
'O' or '1'	one norm
'I'	infinity norm
'F' or 'E'	Frobenius or Euclidean norm
'M'	absolute value of the matrix element having the largest absolute value, i.e., max (|***A***|)

If *m* = 0 or *n* = 0, the function returns zero.

Error conditions

Resource Errors
None.

Computational Errors
None.

Input-Argument Errors

1. *norm* ≠ 'O', '1', 'I', 'F', 'E', or 'M'
2. *m* < 0
3. *n* < 0

4. *m* > *lda*
5. *lda* ≤ 0

Examples

Example 1

This example computes the one norm of real general matrix ***A***.

Call Statements and Input:

```
                NORM  M   N   A  LDA  WORK
                  |   |   |   |   |     |
ANORM = DLANGE(  '1', 9 , 9 , A , 9 , WORK )
```

A = (same as input matrix A in Example 3)

Output:

ANORM = 15.0

Example 2

This example computes the one norm of complex general matrix ***A***.

Call Statements and Input:

```
                NORM  M   N   A  LDA  WORK
                  |   |   |   |   |     |
ANORM = ZLANGE(  '1', 4 , 4 , A , 4 , WORK )
```

A = (same as input matrix A in Example 2)

Output:

ANORM = 25.32

SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system of linear equations ***AX = B*** for ***X***, where ***X*** and ***B*** are general matrices and:

- for SPPSV and DPPSV, ***A*** is a positive definite real symmetric matrix.
- for CPPSV and ZPPSV, ***A*** is a positive definite complex Hermitian matrix.

The matrix ***A***, stored in upper- or lower-packed storage mode, is factored using Cholesky factorization.

Table 136: Data Types.

A*, *B	**Subroutine**
Short-precision real	SPPSV$^{\Delta}$
Long-precision real	DPPSV$^{\Delta}$
Short-precision complex	CPPSV$^{\Delta}$
Long-precision complex	ZPPSV$^{\Delta}$

Table 136: Data Types. (continued)	
A*, *B	**Subroutine**
^ LAPACK	

Syntax

Language	Syntax
Fortran	CALL SPPSV | DPPSV | CPPSV | ZPPSV (*uplo*, *n*, *nrhs*, *ap*, *bx*, *ldb*, *info*)
C and C++	sppsv | dppsv | cppsv | zppsv (*uplo*, *n*, *nrhs*, *ap*, *bx*, *ldb*, *info*);
LAPACKE	info = LAPACKE_sppsv | LAPACKE_dppsv | LAPACKE_cppsv | LAPACKE_zppsv (*matrix_layout*, *uplo*, *n*, *nrhs*, *ap*, *bx*, *ldb*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

uplo

indicates whether matrix ***A*** is stored in upper- or lower-packed storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper-packed storage mode.

If *uplo* = 'L', ***A*** is stored in lower-packed storage mode.

Specified as: a single character. It must be 'U' or 'L'.

n

is the order *n* of matrix ***A*** and the number of rows of matrix ***B***.

Specified as: an integer; $n \geq 0$.

nrhs

is the number of right-hand sides; that is, the number of columns of matrix ***B***.

Specified as: an integer; $nrhs \geq 0$.

ap

is an array, referred to as AP, in which matrix ***A***, to be factored, is stored in upper- or lower-packed storage mode.

Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 136 on page 521.

bx

is the general matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of ***B***.

Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 136 on page 521.

ldb

is the leading dimension of the array specified for ***B***.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

info

See On Return.

On Return

ap

is an array, referred to as AP, in which the transformed matrix ***A*** of order *n*, containing the results of the factorization, is stored in upper- or lower-packed storage mode.

Returned as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 136 on page 521. See "Function" on page 523.

bx

is the general matrix ***X***, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of ***X***.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 136 on page 521.

info

has the following meaning:

If *info* = 0, the subroutine completed successfully.

If *info* > 0, the factorization was unsuccessful. ***B*** is overwritten; that is, the original input is not preserved. *info* is set equal to the order *i* of the first minor encountered having a nonpositive determinant.

Returned as:

- For SPPSV, DPPSV, CPPSV, and ZPPSV, returned as: an integer; info ≥ 0.
- For LAPACKE_sppsv, LAPACKE_dppsv, LAPACKE_cppsv, and LAPACKE_zppsv, returned as an integer function value; info ≥ 0.

Notes

1. These subroutines accept lowercase letters for the *uplo* argument.
2. In your C program, argument *info* must be passed by reference.
3. The matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
4. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.
5. For a description of the storage modes used for the matrices, see:
   - For positive definite real symmetric matrices, see "Positive Definite or Negative Definite Symmetric Matrix" on page 84.
   - For positive definite complex Hermitian matrices, see "Positive Definite or Negative Definite Complex Hermitian Matrix" on page 85.
6. On both input and output, matrices ***A***, ***B***, and ***X*** conform to LAPACK format.
7. The way these subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the *info* argument to provide information about the computational error, but they also provide an error message.

Function

The system ***AX*** = ***B*** is solved for ***X***, where ***X*** and ***B*** are general matrices and:

- for SPPSV and DPPSV, ***A*** is a positive definite real symmetric matrix.

- for CPPSV and ZPPSV, ***A*** is a positive definite complex Hermitian matrix.

The matrix ***A***, stored in upper- or lower-packed storage mode, is factored using the Cholesky factorization method, where ***A*** is expressed as:

$\boldsymbol{A = LL^T}$ **or** $\boldsymbol{A = U^TU}$
for SPPSV and DPPSV

$\boldsymbol{A = LL^H}$ **or** $\boldsymbol{A = U^HU}$
for CPPSV and ZPPSV

where:

L is a lower triangular matrix.
U is an upper triangular matrix.

If *n* is 0, no computation is performed and the subroutine returns after doing some parameter checking. If *n* > 0 and *nrhs* is 0, no solutions are computed and the subroutine returns after factoring the matrix.

See references [“9” on page 1253], [“46” on page 1255], and [“48” on page 1255].

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
Matrix ***A*** is not positive definite.

- The order *i* of the **first** minor encountered having a nonpositive determinant is identified in the computational error message.
- The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2148 is set to be unlimited in the ESSL error option table.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. *n* < 0
3. *nrhs* < 0
4. *n* > *ldb*
5. *ldb* ≤ 0

Examples

Example 1

This example shows how to solve the system ***AX*** = ***B***, where matrix ***A*** is a positive definite real symmetric matrix of order 9, stored in lower-packed storage mode.

On input, matrix ***A*** is:

```
┌                                              ┐
| 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
| 1.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0 |
| 1.0  2.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0 |
| 1.0  2.0  3.0  4.0  4.0  4.0  4.0  4.0  4.0 |
| 1.0  2.0  3.0  4.0  5.0  5.0  5.0  5.0  5.0 |
| 1.0  2.0  3.0  4.0  5.0  6.0  6.0  6.0  6.0 |
| 1.0  2.0  3.0  4.0  5.0  6.0  7.0  7.0  7.0 |
| 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  8.0 |
| 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 |
└                                              ┘
```

On output, all elements of this matrix ***A*** are 1.0.

Note: The AP array is formatted in a triangular arrangement for readability; however, it is stored in lower-packed storage mode.

Call Statement and Input:

```
              UPLO   N  NRHS   AP   BX  LDB   INFO
               |     |   |     |    |    |     |
CALL SPPSV ( 'L',   9,  2,    AP, BX,  9,   INFO )
```

AP = (same as input AP in Example 5)
BX = (same as input BX in Example 5)

Output:

```
AP = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0,
      1.0, 1.0,
      1.0)
```

```
         ┌              ┐
         |   1.0    1.0  |
         |   1.0    2.0  |
         |   1.0    3.0  |
BX   =   |   1.0    4.0  |
         |   1.0    5.0  |
         |   1.0    6.0  |
         |   1.0    7.0  |
         |   1.0    8.0  |
         |   1.0    9.0  |
                       ┘
```

```
INFO = 0
```

Example 2

This example shows how to solve the system ***AX*** = ***B***, where matrix ***A*** is a positive definite real symmetric matrix of order 9, stored in upper-packed storage mode.

On input, matrix ***A*** is:

```
              ┌                                              ┐
              | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
              | 1.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0 |
              | 1.0  2.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0 |
              | 1.0  2.0  3.0  4.0  4.0  4.0  4.0  4.0  4.0 |
              | 1.0  2.0  3.0  4.0  5.0  5.0  5.0  5.0  5.0 |
              | 1.0  2.0  3.0  4.0  5.0  6.0  6.0  6.0  6.0 |
              | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  7.0  7.0 |
              | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  8.0 |
              | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 |
                                                           ┘
```

On output, all elements of this matrix ***A*** are 1.0.

Note: The AP array is formatted in a triangular arrangement for readability; however, it is stored in upper-packed storage mode.

Call Statement and Input:

```
              UPLO   N  NRHS   AP   BX  LDB   INFO
               |     |   |     |    |    |     |
CALL SPPSV ( 'U',   9,  2,    AP, BX,  9,   INFO )
```

AP = (same as input AP in Example 6)
BX = (same as input BX in Example 6)

Output:

```
AP  = (                                         1.0,
                                           1.0, 1.0,
                                      1.0, 1.0, 1.0,
                                 1.0, 1.0, 1.0, 1.0,
                            1.0, 1.0, 1.0, 1.0, 1.0,
                       1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                  1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
             1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

```
        ┌              ┐
        |   1.0    1.0 |
        |   1.0    2.0 |
        |   1.0    3.0 |
BX   =  |   1.0    4.0 |
        |   1.0    5.0 |
        |   1.0    6.0 |
        |   1.0    7.0 |
        |   1.0    8.0 |
        |   1.0    9.0 |
                      ┘
```

```
INFO = 0
```

Example 3

This example shows how to solve the system ***AX*** = ***B***, where matrix ***A*** is a positive definite complex Hermitian matrix of order 3, stored in lower-packed storage mode.

On input, matrix ***A*** is:

```
              ┌                                       ┐
              |  (25.0, 0.0)  (-5.0, -5.0)  (10.0, 5.0) |
              |  (-5.0, 5.0)   (51.0, 0.0)  (4.0, -6.0) |
              | (10.0, -5.0)    (4.0, 6.0)  (71.0, 0.0) |
                                                     ┘
```

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input:

```
             UPLO   N  NRHS  AP  BX  LDB  INFO
               |    |   |    |   |    |    |
CALL ZPPSV ( 'L',  3,  2,   AP, BX,  3,  INFO )
```

AP = (same as input AP in Example 7)
BX = (same as input BX in Example 7)

Output:

```
AP  = ((5.0, 0.0), (-1.0, 1.0), (2.0, -1.0), (7.0, 0.0), (1.0, 1.0), (8.0, 0.0))
```

```
        ┌                          ┐
        |   (2.0, -1.0) ( 2.0, 0.0)  |
BX   =  |   (1.0,  1.0) (-1.0, 2.0)  |
        |   (0.0, -2.0) ( 1.0, 1.0)  |
                                  ┘
```

```
INFO = 0
```

Example 4

This example shows how to solve the system ***AX*** = ***B***, where matrix ***A*** is a positive definite complex Hermitian matrix of order 3, stored in upper-packed storage mode.

On input, matrix ***A*** is:

```
                ┌                                      ┐
                | (9.0,  0.0)  (3.0, 3.0)  (3.0, -3.0) |
                | (3.0, -3.0) (18.0, 0.0)  (8.0, -6.0) |
                | (3.0,  3.0)  (8.0, 6.0) (43.0,  0.0) |
                └                                      ┘
```

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input:

```
             UPLO   N  NRHS   AP   BX   LDB   INFO
              |     |   |     |    |     |     |
CALL ZPPSV ( 'U',   3,  2,    AP, BX,   3,   INFO )
```

AP = (same as input AP in Example 8)
BX = (same as input BX in Example 8)

Output:

```
AP  = ((3.0, 0.0), (1.0, 1.0), (4.0, 0.0), (1.0, -1.0), (2.0, -1.0), (6.0, 0.0))
```

```
         ┌                             ┐
         |   (2.0, -1.0) ( 2.0,  0.0)  |
BX   =   |   (1.0, -1.0) ( 0.0,  1.0)  |
         |   (3.0,  0.0) ( 1.0, -1.0)  |
         └                             ┘
```

```
INFO = 0
```

SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system of linear equations ***AX*** = ***B*** for ***X***, where ***X*** and ***B*** are general matrices and:

- for SPOSV and DPOSV, ***A*** is a positive definite real symmetric matrix.
- for CPOSV and ZPOSV, ***A*** is a positive definite complex Hermitian matrix.

The matrix ***A***, stored in upper or lower storage mode, is factored using Cholesky factorization.

Table 137: Data Types.

A*, *B	**Subroutine**
Short-precision real	SPOSV^
Long-precision real	DPOSV^
Short-precision complex	CPOSV^
Long-precision complex	ZPOSV^
^LAPACK	

Syntax

Table 138: Syntax for supported computing languages.

Language	Syntax
Fortran	CALL SPOSV | DPOSV | CPOSV | ZPOSV (*uplo*, *n*, *nrhs*, *a*, *lda*, *bx*, *ldb*, *info*)
C and C++	sposv | dposv | cposv | zposv (*uplo*, *n*, *nrhs*, *a*, *lda*, *bx*, *ldb*, *info*);
LAPACKE	info = LAPACKE_sposv | LAPACKE_dposv | LAPACKE_cposv | LAPACKE_zposv (*matrix_layout*, *uplo*, *n*, *nrhs*, *a*, *lda*, *bx*, *ldb*);

On Entry

matrix_layout
indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

uplo
indicates whether matrix ***A*** is stored in upper or lower storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

n
is the order *n* of matrix ***A*** and the number of rows of matrix ***B***.

Specified as: an integer; $n \geq 0$.

nrhs
is the number of right-hand sides; that is, the number of columns of matrix ***B***.

Specified as: an integer; $nrhs \geq 0$.

a
is the positive definite matrix ***A*** to be factored.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 137 on page 527. See "Notes:" on page 529.

lda
is the leading dimension of the array specified for ***A***.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

bx
is the general matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of ***B***.

Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 137 on page 527.

ldb
is the leading dimension of the array specified for ***B***.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

info
See On Return.

On Return

a

is the transformed matrix ***A*** of order *n*, containing the results of the factorization.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 137 on page 527. See "Function" on page 529.

bx

is the general matrix ***X***, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of ***X***.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 137 on page 527.

info

has the following meaning:

If *info* = 0, the subroutine completed successfully.

If *info* > 0, the factorization was unsuccessful and the solution was not computed. *info* is set equal to the order *i* of the first minor encountered having a nonpositive determinant.

Returned as:

- For SPOSV, DPOSV, CPOSV, and ZPOSV, returned as: an integer; info ≥ 0.
- For LAPACKE_sposv, LAPACKE_dposv, LAPACKE_cposv, and LAPACKE_zposv, returned as an integer function value; info ≥ 0.

Notes:

1. In your C program, argument *info* must be passed by reference.
2. All subroutines accept lowercase letters for the *uplo* argument.
3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.
4. The way these subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the *info* argument to provide information about the computational error, but they also provide an error message.
5. On both input and output, matrices ***A***, ***B***, and ***X*** conform to LAPACK format.
6. For a description of the storage modes used for the matrices, see:
   - For positive definite real symmetric matrices, see "Positive Definite or Negative Definite Symmetric Matrix" on page 84.
   - For positive definite complex Hermitian matrices, see "Positive Definite or Negative Definite Complex Hermitian Matrix" on page 85.
7. The matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

These subroutines solve the system of linear equations ***AX*** = ***B*** for ***X***, where ***X*** and ***B*** are general matrices and:

- for SPOSV and DPOSV, ***A*** is a positive definite real symmetric matrix.
- for CPOSV and ZPOSV, ***A*** is a positive definite complex Hermitian matrix.

The matrix ***A*** is factored using Cholesky factorization, where ***A*** is expressed as:

A*=*LL*^{T} or *A*=*U*^{T}*U
for SPOSV and DPOSV

A*=*LL*^{H} or *A*=*U*^{H}*U
for CPOSV and ZPOSV

where:

L is a unit lower triangular matrix.
U is an upper triangular matrix.

If *n* is 0, no computation is performed and the subroutine returns after doing some parameter checking. If *n* > 0 and *nrhs* is 0, no solutions are computed and the subroutine returns after factoring the matrix.

See references [“9” on page 1253], [“46” on page 1255], and [“84” on page 1257].

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
Matrix ***A*** is not positive definite.

The order *i* of the **first** minor encountered having a nonpositive determinant is identified in the computational error message.

The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2148 is set to be unlimited in the ESSL error option table.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. *n* < 0
3. *nrhs* < 0
4. *n* > *lda*
5. *lda* ≤ 0
6. *n* > *ldb*
7. *ldb* ≤ 0

Examples

Example 1

This example shows how to solve the system ***AX*** = ***B***, where:

Matrix ***A*** is the same used as input in Example 1 for SPOTRF.
Matrix ***B*** is the same used as input in Example 1 for SPOTRS.

Call Statement and Input:

```
             UPLO  N  NRHS  A  LDA  BX  LDB  INFO
              |    |   |    |   |   |    |    |
CALL SPOSV( 'L',  9 , 2 ,  A , 9 , BX , 9 , INFO)
```

A = (same as input A in Example 1)
BX = (same as input BX in Example 1)

Output:

```

         | 1.0   .    .    .    .    .    .    .    .  |
         | 1.0  1.0   .    .    .    .    .    .    .  |
         | 1.0  1.0  1.0   .    .    .    .    .    .  |
        | 1.0  1.0  1.0  1.0   .    .    .    .    .  |
A    =  | 1.0  1.0  1.0  1.0  1.0   .    .    .    .  |
        | 1.0  1.0  1.0  1.0  1.0  1.0   .    .    .  |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0   .    .  |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0   .  |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
```

```
        | 1.0  1.0 |
        | 1.0  2.0 |
        | 1.0  3.0 |
        | 1.0  4.0 |
BX   =  | 1.0  5.0 |
        | 1.0  6.0 |
        | 1.0  7.0 |
        | 1.0  8.0 |
        | 1.0  9.0 |
```

```
INFO =  0
```

Example 2

This example shows how to solve the system ***AX*** = ***B***, where:

Matrix ***A*** is the same used as input in Example 2 for SPOTRF.
Matrix ***B*** is the same used as input in Example 2 for SPOTRS.

Call Statement and Input:

```
            UPLO   N  NRHS   A   LDA   BX    LDB  INFO
             |     |   |     |    |    |      |    |
CALL SPOSV( 'U' , 9 , 2 ,  A , 9 , BX ,   9 , INFO)
```

A = (same as input A in Example 2)
BX = (same as input BX in Example 2)

Output:

```
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        |  .   1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        |  .    .   1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        |  .    .    .   1.0  1.0  1.0  1.0  1.0  1.0 |
A    =  |  .    .    .    .   1.0  1.0  1.0  1.0  1.0 |
        |  .    .    .    .    .   1.0  1.0  1.0  1.0 |
        |  .    .    .    .    .    .   1.0  1.0  1.0 |
        |  .    .    .    .    .    .    .   1.0  1.0 |
        |  .    .    .    .    .    .    .    .   1.0 |
```

```
        | 1.0  1.0 |
        | 1.0  2.0 |
        | 1.0  3.0 |
        | 1.0  4.0 |
BX   =  | 1.0  5.0 |
        | 1.0  6.0 |
        | 1.0  7.0 |
        | 1.0  8.0 |
        | 1.0  9.0 |
```

```
INFO   =  0
```

Example 3

This example shows how to solve the system ***AX*** = ***B***, where:

Matrix ***A*** is the same used as input in Example 3 for CPOTRF.
Matrix ***BX*** is the same used as input in Example 3 for CPOTRS.

Call Statement and Input:

```
            UPLO  N  NRHS  A  LDA  BX  LDB  INFO
             |    |   |    |   |    |   |    |
CALL CPOSV( 'L',  3 , 2 ,  A , 3 , BX , 3 , INFO)
```

```
        | ( 25.0,  0.0) ( -5.0, -5.0) (10.0,  5.0) |
A    =  | ( -5.0,  5.0) ( 51.0,  0.0) ( 4.0, -6.0) |
        | ( 10.0, -5.0) (  4.0,  6.0) (71.0,  0.0) |
```

```
        | ( 60.0,  -55.0) ( 70.0,  10.0) |
BX   =  | ( 34.0,   58.0) (-51.0, 110.0) |
        | ( 13.0, -152.0) ( 75.0,  63.0) |
```

Output:

```
        | ( 5.0,  0.0) (-5.0, -5.0) (10.0,  5.0) |
A    =  | (-1.0,  1.0) ( 7.0,  0.0) ( 4.0, -6.0) |
        | ( 2.0, -1.0) ( 1.0,  1.0) ( 8.0,  0.0) |
```

```
        | ( 2.0, -1.0) ( 2.0, 0.0) |
BX   =  | ( 1.0,  1.0) (-1.0, 2.0) |
        | ( 0.0, -2.0) ( 1.0, 1.0) |
```

```
INFO =  0
```

Example 4

This example shows how to solve the system ***AX*** = ***B***, where:

Matrix ***A*** is the same used as input in Example 4 for CPOTRF.
Matrix ***BX*** is the same used as input in Example 4 for CPOTRS.

Call Statement and Input:

```
            UPLO  N  NRHS  A  LDA  BX  LDB  INFO
             |    |   |    |   |    |   |    |
CALL CPOSV( 'U',  3 , 2 ,  A , 3 , BX , 3 , INFO)
```

```
        | (9.0,  0.0) ( 3.0, 3.0) ( 3.0, -3.0) |
A    =  | (3.0, -3.0) (18.0, 0.0) ( 8.0, -6.0) |
        | (3.0,  3.0) ( 8.0, 6.0) (43.0,  0.0) |
```

```
        | ( 33.0, -18.0) (15.0,  -3.0) |
BX   =  | ( 45.0, -45.0) ( 8.0,  -2.0) |
        | (152.0,   1.0) (43.0, -29.0) |
```

Output:

Note: The strict lower part of ***A*** is not referenced.

```
        | (3.0,  0.0) (1.0, 1.0) (1.0, -1.0) |
A    =  | (3.0, -3.0) (4.0, 0.0) (2.0, -1.0) |
        | (3.0,  3.0) (8.0, 6.0) (6.0,  0.0) |
```

```
        | ( 2.0, -1.0) ( 2.0,  0.0) |
BX   =  | ( 1.0, -1.0) ( 0.0,  1.0) |
        | ( 3.0,  0.0) ( 1.0, -1.0) |
```

```
INFO =  0
```

SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)

Purpose

These subroutines factor matrix ***A*** as explained below:

SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, and ZPOF

The SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, and ZPOF subroutines factor matrix ***A*** stored in upper or lower storage mode, where:

- For SPOTRF, DPOTRF, SPOF, and DPOF, ***A*** is a positive definite real symmetric matrix.
- For CPOTRF, ZPOTRF, CPOF, and ZPOF, ***A*** is a positive definite complex Hermitian matrix.

Matrix ***A*** is factored using Cholesky factorization.

To solve the system of equations with one or more right-hand sides, follow the call to SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, or ZPOF with a call to SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, or ZPOSM, respectively.

To find the inverse of matrix ***A***, follow the call to SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, or DPOF with a call to SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, or DPOICD, respectively.

To estimate the reciprocal of the condition number of matrix ***A***, follow the call to SPOTRF, DPOTRF, CPOTRF, or ZPOTRF with a call to SPOCON, DPOCON, CPOCON, or ZPOCON, respectively.

SPPTRF, DPPTRF, CPPTRF, and ZPPTRF

The SPPTRF, DPPTRF, CPPTRF, and ZPPTRF subroutines factor matrix ***A***, stored in upper- or lower-packed storage mode, where:

- For SPPTRF and DPPTRF, ***A*** is a positive definite real symmetric matrix.
- For CPPTRF and ZPPTRF, ***A*** is a positive definite complex Hermitian matrix.

Matrix ***A*** is factored using Cholesky factorization.

To solve the system of equations with one or more right-hand sides, follow the call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF with a call to SPPTRS, DPPTRS, CPPTRS, or ZPPTRS, respectively.

To find the inverse of matrix ***A***, follow the call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF with a call to SPPTRI, DPPTRI, CPPTRI, or ZPPTRI, respectively.

To estimate the reciprocal of the condition number of matrix ***A***, follow the call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF with a call to SPPCON, DPPCON, CPPCON, or ZPPCON, respectively.

SPPF and DPPF

The SPPF and DPPF subroutines factor positive definite real symmetric matrix ***A***, stored in lower-packed storage mode, using Gaussian elimination ($\boldsymbol{LDL}^T$) or Cholesky factorization. To solve a system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SPPS or DPPS, respectively. To find the inverse of matrix ***A***, follow the call to these subroutines, performing Cholesky factorization, with a call to SPPICD or DPPICD, respectively.

Table 139: Data Types.	
A	**Subroutine**
Short-precision real	SPOTRF^, SPOF, SPPTRF^, and SPPF
Long-precision real	DPOTRF^, DPOF, DPPTRF^, and DPPF
Short-precision complex	CPOTRF^, CPOF, and CPPTRF^
Long-precision complex	ZPOTRF^, ZPOF, and ZPPTRF^
^LAPACK	

Note: The output from each of these subroutines should be used only as input for specific other subroutines, as shown in the table below.

Output from this subroutine:	**Should be used only as input to the following subroutines:**		
Solve	**Inverse**	**Reciprocal of the condition number**	
SPOTRF	SPOTRS	SPOTRI	SPOCON
DPOTRF	DPOTRS	DPOTRI	DPOCON
CPOTRF	CPOTRS	CPOTRI	CPOCON
ZPOTRF	ZPOTRS	ZPOTRI	ZPOCON
SPOF	SPOSM	SPOICD	SPOICD
DPOF	DPOSM	DPOICD	DPOICD
CPOF	CPOSM		
ZPOF	ZPOSM		
SPPTRF	SPPTRS	SPPTRI	SPPCON
DPPTRF	DPPTRS	DPPTRI	DPPCON
CPPTRF	CPPTRS	CPPTRI	CPPCON
ZPPTRF	ZPPTRS	ZPPTRI	ZPPCON
SPPF	SPPS	SPPICD	SPPICD
DPPF	DPPS	DPPICD	DPPICD

Syntax

Language	**Syntax**
Fortran	CALL SPOTRF | DPOTRF | CPOTRF | ZPOTRF (*uplo*, *n*, *a*, *lda*, *info*) CALL SPOF | DPOF | CPOF | ZPOF (*uplo*, *a*, *lda*, *n*) CALL SPPTRF | DPPTRF | CPPTRF | ZPPTRF (*uplo*, *n*, *ap*, *info*) CALL SPPF | DPPF (*ap*, *n*, *iopt*)

Language	Syntax

C and C++	spotrf | dpotrf | cpotrf | zpotrf (*uplo*, *n*, *a*, *lda*, *info*);
	spof | dpof | cpof | zpof (*uplo*, *a*, *lda*, *n*);
	spptrf | dpptrf | cpptrf | zpptrf (*uplo*, *n*, *ap*, *info*);
	sppf | dppf (*ap*, *n*, *iopt*);

LAPACKE	info = LAPACKE_spotrf | LAPACKE_dpotrf | LAPACKE_cpotrf | LAPACKE_zpotrf (*matrix_layout*, *uplo*, *n*, *a*, *lda*);
	nfo = LAPACKE_spptrf | LAPACKE_dpptrf | LAPACKE_cpptrf | LAPACKE_zpptrf (*matrix_layout*, *uplo*, *n*, *ap*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

uplo

indicates whether matrix ***A*** is stored in upper or lower storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

ap

is an array, referred to as AP, in which matrix ***A***, to be factored, is stored as follows:

SPPTRF, DPPTRF, CPPTRF, and ZPPTRF

Upper-packed or lower-packed storage mode

SPPF and DPPF

Lower-packed storage mode

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 139 on page 534. See “Notes” on page 537.

For SPPTRF, DPPTRF, CPPTRF, and ZPPTRF:

The array must have at least $n(n+1)/2$ elements.

For SPPF and DPPF:

If *iopt* = 0 or 10, the array must have at least $n(n+1)/2+n$ elements.

If *iopt* = 1 or 11, the array must have at least $n(n+1)/2$ elements.

a

is the positive definite matrix ***A***, to be factored.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 139 on page 534.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

n

is the order *n* of matrix ***A***.

Specified as: an integer; $n \geq 0$.

iopt

determines the type of computation to be performed, where:

If *iopt* = 0, the matrix is factored using the $\boldsymbol{LDL}^{\mathrm{T}}$ method, and the output is stored in an internal format.

If *iopt* = 1, the matrix is factored using Cholesky factorization, and the output is stored in an internal format.

If *iopt* = 10, the matrix is factored using the $\boldsymbol{LDL}^{\mathrm{T}}$ method, and the output is stored in lower-packed storage mode.

If *iopt* = 11, the matrix is factored using Cholesky factorization, and the output is stored in lower-packed storage mode.

Specified as: an integer; *iopt* = 0, 1, 10, or 11.

info

See On Return.

On Return

ap

is an array, referred to as AP, in which the transformed matrix ***A*** of order *n*, containing the results of the factorization, is stored.

For SPPTRF, DPPTRF, CPPTRF, and ZPPTRF:
The transformed matrix is stored in upper-packed or lower-packed storage mode.

For SPPF and DPPF:
If *iopt* is 0 or 1, the transformed matrix is stored in an internal format and should only be used as input to the corresponding solve or inverse subroutine.

If *iopt* is 10 or 11, the transformed matrix is stored in lower-packed storage mode.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 139 on page 534.

For SPPTRF, DPPTRF, CPPTRF, and ZPPTRF:
The array contains at least $n(n+1)/2$ elements.

For SPPF and DPPF:
If *iopt* = 0 or 10, the array contains $n(n+1)/2+n$ elements.

If *iopt* = 1 or 11, the array contains $n(n+1)/2$ elements.

See "Notes" on page 537 and see "Function" on page 537.

a

is the transformed matrix ***A*** of order *n*, containing the results of the factorization. See "Function" on page 537.

Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 139 on page 534.

info

has the following meaning:

If *info* = 0, the factorization completed successfully.

If *info* > 0, *info* is set equal to the order *i* of the first minor encountered having a nonpositive determinant.

Returned as:

- For SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF, returned as: an integer; info ≥ 0.
- For LAPACKE_spotrf, LAPACKE_dpotrf, LAPACKE_cpotrf, LAPACKE_zpotrf, LAPACKE_spptrf, LAPACKE_dpptrf, LAPACKE_cpptrf, and LAPACKE_zpptrf, returned as an integer function value; info ≥ 0.

Specified as: an integer; *info* ≥ 0.

Notes

1. In your C program, argument *info* must be passed by reference.
2. All subroutines accept lowercase letters for the *uplo* argument.
3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.
4. In the input and output arrays specified for *ap*, the first *n*(*n*+1)/2 elements are matrix elements. The additional *n* locations, required in the array when *iopt* = 0 or 10, are used for working storage by this subroutine and should not be altered between calls to the factorization and solve subroutines.
5. If *iopt* = 0 or 1, SPPF and DPPF in some cases utilize algorithms based on recursive packed storage format. As a result, on output, if *iopt* = 0 or 1, the array specified for AP may be stored in this new format rather than the conventional lower packed format. (See references [“63” on page 1256], [“79” on page 1257], and [“81” on page 1257]).

   The array specified for AP should not be altered between calls to the factorization and solve subroutines; otherwise unpredictable results may occur.
6. The way _POTRF and _PPTRF subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the *info* argument to provide information about the computational error, but they also provide an error message.
7. On both input and output, matrix ***A*** conforms to LAPACK format.
8. For a description of the storage modes used for the matrices, see:
   - For positive definite symmetric matrices, see “Positive Definite or Negative Definite Symmetric Matrix” on page 84.
   - For positive definite complex Hermitian matrices, see “Positive Definite or Negative Definite Complex Hermitian Matrix” on page 85.

Function

The functions for these subroutines are described.

For SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, and ZPOF

The positive definite matrix ***A***, stored in upper or lower storage mode, is factored using Cholesky factorization, where ***A*** is expressed as:

$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^{\mathrm{T}}$ **or** $\boldsymbol{A} = \boldsymbol{U}^{\mathrm{T}}\boldsymbol{U}$
for SPOTRF, DPOTRF, SPOF, and DPOF

$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^{\mathrm{H}}$ **or** $\boldsymbol{A} = \boldsymbol{U}^{\mathrm{H}}\boldsymbol{U}$
for CPOTRF, ZPOTRF, CPOF, and ZPOF

where:

L is a lower triangular matrix.
U is an upper triangular matrix.

If *n* is 0, no computation is performed. See references [“9” on page 1253] and [“46” on page 1255].

For SPPTRF, DPPTRF, CPPTRF, and ZPPTRF:

The positive definite matrix ***A***, stored in upper-packed or lower-packed storage mode, is factored using Cholesky factorization, where ***A*** is expressed as:

$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^{\mathrm{T}}$ **or** $\boldsymbol{A} = \boldsymbol{U}^{\mathrm{T}}\boldsymbol{U}$
for SPPTRF and DPPTRF

A* = *LL*^{H} or *A* = *U*^{H}*U
for CPPTRF and ZPPTRF

where:

L is a lower triangular matrix.
U is an upper triangular matrix.

If *n* is 0, no computation is performed. See references [“9” on page 1253], [“46” on page 1255], and [“80” on page 1257].

For SPPF and DPPF:

If *iopt* = 0 or 10, the positive definite symmetric matrix ***A***, stored in lower-packed storage mode, is factored using Gaussian elimination, where ***A*** is expressed as:

A = ***LDL***^{T}

where:

L is a unit lower triangular matrix.
D is a diagonal matrix.

If *iopt* = 1 or 11, the positive definite symmetric matrix ***A***, stored in lower-packed storage mode, is factored using Cholesky factorization, where ***A*** is expressed as:

A = ***LL***^{T}

where ***L*** is a lower triangular matrix.

If *n* is 0, no computation is performed. See references [“9” on page 1253] and [“46” on page 1255].

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors

1. Matrix ***A*** is not positive definite (for SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPPTRF, DPPTRF, CPPTRF, and ZPPTRF).
   - The order *i* of the **first** minor encountered having a nonpositive determinant is identified in the computational error message.
   - The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2148 is set to be unlimited in the ESSL error option table.
2. Matrix ***A*** is not positive definite (for SPPF and DPPF when *iopt* = 0 or 10).
   - Processing continues to the end of the matrix.
   - One or more elements of ***D*** contain values less than or equal to 0; all elements of ***D*** are checked. The index *i* of the **last** nonpositive element encountered is identified in the computational error message.
   - The return code is set to 1.
   - *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2104 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see “What Can You Do about ESSL Computational Errors?” on page 65.
3. Matrix ***A*** is not positive definite (for SPPF and DPPF when *iopt* = 1 or 11 and for SPOF, DPOF, CPOF, and ZPOF).
   - Processing stops at the first occurrence of a nonpositive definite diagonal element.

- The order *i* of the **first** minor encountered having a nonpositive determinant is identified in the computational error message.
- The return code is set to 1.
- *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2115 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. *n* < 0
3. *n* > *lda*
4. *lda* ≤ 0
5. *iopt* ≠ 0, 1, 10, or 11

Examples

Example 1

This example shows a factorization of the same positive definite symmetric matrix ***A*** of order 9 used in Example 9, but stored in lower storage mode.

Call Statement and Input:

```
             UPLO  N   A  LDA  INFO
               |   |   |   |     |
CALL SPOTRF( 'L' , 9 , A , 9 , INFO )
```

or

```
           UPLO  A  LDA  N
             |   |   |   |
CALL SPOF( 'L' , A , 9 , 9 )
```

```
         | 1.0   .    .    .    .    .    .    .    .  |
         | 1.0  2.0   .    .    .    .    .    .    .  |
         | 1.0  2.0  3.0   .    .    .    .    .    .  |
         | 1.0  2.0  3.0  4.0   .    .    .    .    .  |
A    =   | 1.0  2.0  3.0  4.0  5.0   .    .    .    .  |
         | 1.0  2.0  3.0  4.0  5.0  6.0   .    .    .  |
         | 1.0  2.0  3.0  4.0  5.0  6.0  7.0   .    .  |
         | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0   .  |
         | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 |
```

Output:

```
         | 1.0   .    .    .    .    .    .    .    .  |
         | 1.0  1.0   .    .    .    .    .    .    .  |
         | 1.0  1.0  1.0   .    .    .    .    .    .  |
         | 1.0  1.0  1.0  1.0   .    .    .    .    .  |
A    =   | 1.0  1.0  1.0  1.0  1.0   .    .    .    .  |
         | 1.0  1.0  1.0  1.0  1.0  1.0   .    .    .  |
         | 1.0  1.0  1.0  1.0  1.0  1.0  1.0   .    .  |
         | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0   .  |
         | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
```

```
INFO  =   0
```

Example 2

This example shows a factorization of the same positive definite symmetric matrix ***A*** of order 9 used in Example 9, but stored in upper storage mode.

Call Statement and Input:

```
             UPLO   N   A  LDA  INFO
              |     |   |   |    |
CALL SPOTRF( 'U' , 9 , A , 9 , INFO )
```

or

```
           UPLO  A  LDA  N
            |    |   |   |
CALL SPOF( 'U' , A , 9 , 9 )
```

```
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        |  .   2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0 |
        |  .    .   3.0  3.0  3.0  3.0  3.0  3.0  3.0 |
        |  .    .    .   4.0  4.0  4.0  4.0  4.0  4.0 |
A    =  |  .    .    .    .   5.0  5.0  5.0  5.0  5.0 |
        |  .    .    .    .    .   6.0  6.0  6.0  6.0 |
        |  .    .    .    .    .    .   7.0  7.0  7.0 |
        |  .    .    .    .    .    .    .   8.0  8.0 |
        |  .    .    .    .    .    .    .    .   9.0 |
```

Output:

```
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        |  .   1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        |  .    .   1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        |  .    .    .   1.0  1.0  1.0  1.0  1.0  1.0 |
A    =  |  .    .    .    .   1.0  1.0  1.0  1.0  1.0 |
        |  .    .    .    .    .   1.0  1.0  1.0  1.0 |
        |  .    .    .    .    .    .   1.0  1.0  1.0 |
        |  .    .    .    .    .    .    .   1.0  1.0 |
        |  .    .    .    .    .    .    .    .   1.0 |
```

```
INFO  =   0
```

Example 3

This example shows a factorization of positive definite complex Hermitian matrix ***A*** of order 3, stored in lower storage mode, where on input matrix ***A*** is:

```
              |  (25.0, 0.0)  (-5.0, -5.0)  (10.0, 5.0) |
              |  (-5.0, 5.0)   (51.0, 0.0)  (4.0, -6.0) |
              | (10.0, -5.0)    (4.0, 6.0)  (71.0, 0.0) |
```

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input:

```
             UPLO   N   A  LDA  INFO
              |     |   |   |    |
CALL CPOTRF( 'L' , 3 , A , 3 , INFO )
```

or

```
           UPLO  A  LDA  N
            |    |   |   |
CALL CPOF( 'L' , A , 3 , 3 )
```

```
        | (25.0,   . )       .          .      |
A    =  | (-5.0,  5.0) (51.0,  . )      .      |
        | (10.0, -5.0) ( 4.0, 6.0) (71.0, . )  |
```

Output:

```
        | ( 5.0,  0.0)       .          .      |
A    =  | (-1.0,  1.0)  (7.0, 0.0)      .      |
        | ( 2.0, -1.0)  (1.0, 1.0)  (8.0, 0.0) |
```

```
INFO =      0
```

Example 4

This example shows a factorization of positive definite complex Hermitian matrix ***A*** of order 3, stored in upper storage mode, where on input matrix ***A*** is:

```
               | (9.0,  0.0)  (3.0, 3.0)  (3.0, -3.0) |
               | (3.0, -3.0) (18.0, 0.0)  (8.0, -6.0) |
               | (3.0,  3.0)  (8.0, 6.0) (43.0,  0.0) |
```

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input:

```
             UPLO  N   A  LDA  INFO
              |    |   |   |    |
CALL CPOTRF( 'U' , 3 , A , 3 , INFO )
```

or

```
           UPLO  A  LDA  N
            |    |   |   |
CALL CPOF( 'U' , A , 3 , 3 )
```

```
        | (9.0,  . )  ( 3.0,3.0)  ( 3.0,-3.0) |
A    =  |     .       (18.0, . )  ( 8.0,-6.0) |
        |     .            .      (43.0,  . ) |
```

Output:

```
        | (3.0, 0.0)    (1.0, 1.0)  (1.0, -1.0) |
A    =  |     .         (4.0, 0.0)  (2.0, -1.0) |
        |     .              .      (6.0,  0.0) |
```

```
INFO =     0
```

Example 5

This example shows a factorization (using the Cholesky factorization method) of the same positive definite symmetric matrix ***A*** of order 9 used in Example 9, but stored in lower-packed storage mode.

Note: The AP arrays are formatted in a triangular arrangement for readability; however, they are stored in lower-packed storage mode.

Call Statement and Input:

```
             UPLO  N   AP   INFO
              |    |   |     |
CALL SPPTRF( 'L' , 9 , AP , INFO )
```

```
AP = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0,
      3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0,
      4.0, 4.0, 4.0, 4.0, 4.0, 4.0,
      5.0, 5.0, 5.0, 5.0, 5.0,
      6.0, 6.0, 6.0, 6.0,
      7.0, 7.0, 7.0,
      8.0, 8.0,
      9.0)
```

Output:

```
AP  = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
       1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
       1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
       1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
       1.0, 1.0, 1.0, 1.0, 1.0,
       1.0, 1.0, 1.0, 1.0,
       1.0, 1.0, 1.0,
       1.0, 1.0,
       1.0)

INFO = 0
```

Example 6

This example shows a factorization (using the Cholesky factorization method) of the same positive definite symmetric matrix ***A*** of order 9 used in Example 9, but stored in upper-packed storage mode.

Note: The AP arrays are formatted in a triangular arrangement for readability; however, they are stored in upper-packed storage mode.

Call Statement and Input:

```
             UPLO  N   AP  INFO
              |    |   |    |
CALL SPPTRF( 'U',  9,  AP, INFO )
```

```
AP = (                                     1.0,
                                      1.0, 2.0,
                                 1.0, 2.0, 3.0,
                            1.0, 2.0, 3.0, 4.0,
                       1.0, 2.0, 3.0, 4.0, 5.0,
                  1.0, 2.0, 3.0, 4.0, 5.0, 6.0,
             1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0,
        1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
   1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0)
```

Output:

```
AP  = (                                     1.0,
                                       1.0, 1.0,
                                  1.0, 1.0, 1.0,
                             1.0, 1.0, 1.0, 1.0,
                        1.0, 1.0, 1.0, 1.0, 1.0,
                   1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
              1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
         1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

```
INFO = 0
```

Example 7

This example shows a factorization (using the Cholesky factorization method) of the same positive definite complex Hermitian matrix ***A*** of order 3 used in Example 3, but stored in lower-packed storage mode.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input:

```
             UPLO  N   AP   INFO
              |    |   |     |
CALL ZPPTRF( 'L' , 3 , AP , INFO )
```

```
AP = ((25.0, .), (-5.0, 5.0), (10.0, -5.0), (51.0, .), (4.0, 6.0), (71.0, .))
```

Output:

```
AP = ((5.0, 0.0), (-1.0, 1.0), (2.0, -1.0), (7.0, 0.0), (1.0, 1.0), (8.0, 0.0))

INFO = 0
```

Example 8

This example shows a factorization (using the Cholesky factorization method) of the same positive definite complex Hermitian matrix ***A*** of order 3 used in Example 4, but stored in upper-packed storage mode.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input:

```
             UPLO  N   AP   INFO
              |    |   |     |
CALL ZPPTRF( 'U' , 3 , AP , INFO )
```

```
AP = ((9.0, .), (3.0, 3.0), (18.0, .), (3.0, -3.0), (8.0, -6.0), (43.0, .))
```

Output:

```
AP = ((3.0, 0.0), (1.0, 1.0), (4.0, 0.0), (1.0, -1.0), (2.0, -1.0), (6.0, 0.0))

INFO = 0
```

Example 9

This example shows a factorization (using the Gaussian elimination method) of positive definite symmetric matrix ***A*** of order 9, stored in lower-packed storage mode, where on input matrix ***A*** is:

```
              | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
              | 1.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0 |
              | 1.0  2.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0 |
              | 1.0  2.0  3.0  4.0  4.0  4.0  4.0  4.0  4.0 |
              | 1.0  2.0  3.0  4.0  5.0  5.0  5.0  5.0  5.0 |
              | 1.0  2.0  3.0  4.0  5.0  6.0  6.0  6.0  6.0 |
              | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  7.0  7.0 |
              | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  8.0 |
              | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 |
```

On output, all elements of this matrix ***A*** are 1.0.

Note: The AP arrays are formatted in a triangular arrangement for readability; however, they are stored in lower-packed storage mode.

Call Statement and Input:

```
           AP   N  IOPT
           |    |   |
CALL SPPF( AP, 9,  0 )
```

```
AP = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0,
      3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0,
      4.0, 4.0, 4.0, 4.0, 4.0, 4.0,
      5.0, 5.0, 5.0, 5.0, 5.0,
      6.0, 6.0, 6.0, 6.0,
      7.0, 7.0, 7.0,
      8.0, 8.0,
      9.0,
       . ,  . ,  . ,  . ,  . ,  . ,  . ,  . ,  .  )
```

Output:

```
AP = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0,
      1.0, 1.0,
      1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

Example 10

This example shows a factorization (using the Cholesky factorization method) of the same positive definite symmetric matrix ***A*** of order 9 used in Example 9, stored in lower-packed storage mode.

Note: The AP arrays are formatted in a triangular arrangement for readability; however, they are stored in lower-packed storage mode.

Call Statement and Input:

```
           AP   N  IOPT
           |    |   |
CALL SPPF( AP, 9,  1 )
```

```
AP = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0,
      3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0,
      4.0, 4.0, 4.0, 4.0, 4.0, 4.0,
      5.0, 5.0, 5.0, 5.0, 5.0,
      6.0, 6.0, 6.0, 6.0,
      7.0, 7.0, 7.0,
      8.0, 8.0,
      9.0)
```

Output:

```
AP = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0,
      1.0, 1.0,
      1.0)
```

SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS, CPPTRS, and ZPPTRS (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system ***AX*** = ***B*** for ***X***, where ***X*** and ***B*** are general matrices and:

- For SPOTRS, DPOTRS, SPOSM, DPOSM, SPPTRS, and DPPTRS, ***A*** is a positive definite real symmetric matrix.
- For CPOTRS, ZPOTRS, CPOSM, ZPOSM, CPPTRS, and ZPPTRS, ***A*** is a positive definite complex Hermitian matrix.

SPOTRS, DPOTRS, CPOTRS, and ZPOTRS use the results of the factorization of matrix ***A***, produced by a preceding call to SPOTRF, DPOTRF, CPOTRF, or ZPOTRF, respectively.

SPOSM, DPOSM, CPOSM, and ZPOSM use the results of the factorization of matrix ***A***, produced by a preceding call to SPOF/SPOFCD, DPOF/DPOFCD, CPOF, or ZPOF, respectively.

SPPTRS, DPPTRS, CPPTRS, and ZPPTRS use the results of the factorization of matrix ***A***, produced by a preceding call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF, respectively.

Table 140: Data Types.

A, ***B***, ***X***	**Subroutine**
Short-precision real	SPOTRS▵, SPOSM, and SPPTRS▵
Long-precision real	DPOTRS▵, DPOSM, and DPPTRS▵
Short-precision complex	CPOTRS▵, CPOSM, and CPPTRS▵
Long-precision complex	ZPOTRS▵, ZPOSM, and ZPPTRS▵
▵LAPACK	

Note: The input to these solve subroutines must be the output from the corresponding factorization subroutines.

Syntax

Language	Syntax
Fortran	CALL SPOTRS | DPOTRS | CPOTRS | ZPOTRS (*uplo*, *n*, *nrhs*, *a*, *lda*, *bx*, *ldb*, *info*) CALL SPOSM | DPOSM | CPOSM | ZPOSM (*uplo*, *a*, *lda*, *n*, *bx*, *ldb*, *nrhs*) CALL SPPTRS | DPPTRS | CPPTRS | ZPPTRS (*uplo*, *n*, *nrhs*, *ap*, *bx*, *ldb*, *info*)

C and C++	spotrs | dpotrs | cpotrs | zpotrs (*uplo*, *n*, *nrhs*, *a*, *lda*, *bx*, *ldb*, *info*);
	sposm | dposm | cposm | zposm (*uplo*, *a*, *lda*, *n*, *bx*, *ldb*, *nrhs*);
	spptrs | dpptrs | cpptrs | zpptrs (*uplo*, *n*, *nrhs*, *ap*, *bx*, *ldb*, *info*);

LAPACKE	info = LAPACKE_spotrs | LAPACKE_dpotrs | LAPACKE_cpotrs | LAPACKE_zpotrs (*matrix_layout*, *uplo*, *n*, *nrhs*, *a*, *lda*, *bx*, *ldb*);
	info = LAPACKE_spptrs | LAPACKE_dpptrs | LAPACKE_cpptrs | LAPACKE_zpptrs (*matrix_layout*, *uplo*, *n*, *nrhs*, *ap*, *bx*, *ldb*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

uplo

indicates whether the original matrix ***A*** is stored in upper or lower storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

a

is the factorization of positive definite matrix ***A***, produced by a preceding call to SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF/SPOFCD, DPOF/DPOFCD, CPOF, or ZPOF.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 140 on page 545.

ap

is an array, referred to as AP, in which the factorization of positive definite matrix ***A***, produced by a preceding call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF, is stored in upper-packed or lower-packed storage mode.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 140 on page 545.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

n

is the order of matrix ***A*** and the number of rows of matrix ***B***.

Specified as: an integer; *n* ≥ 0.

bx

is the general matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of ***B***.

Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 140 on page 545.

ldb

is the leading dimension of the array specified for *b*.

Specified as: an integer; *ldb* > 0 and *ldb* ≥ *n*.

nrhs

is the number of right-hand sides; that is, the number of columns of matrix ***B***.

Specified as: an integer; *nrhs* ≥ 0.

info

See On Return.

On Return

bx

is the general matrix ***X***, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of ***X***.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 140 on page 545.

info

info has the following meaning:

If *info* = 0, the solve completed successfully.

Returned as:

- For SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS, CPPTRS, and ZPPTRS, returned as: an integer; info ≥ 0.
- For LAPACKE_spotrs, LAPACKE_dpotrs, LAPACKE_cpotrs, LAPACKE_zpotrs, LAPACKE_spptrs, LAPACKE_dpptrs, LAPACKE_cpptrs, and LAPACKE_zpptrs, returned as an integer function value; info ≥ 0.

Notes

1. In your C program, argument *info* must be passed by reference.
2. All subroutines accept lowercase letters for the *uplo* argument.
3. The scalar data specified for input arguments *uplo*, *lda*, and *n* for these subroutines must be the same as the corresponding input arguments specified for SPOTRF/SPOF/SPOFCD/SPPTRF, DPOTRF/DPOF/DPOFCD/DPPTRF, CPOTRF/CPOF/CPPTRF, and ZPOTRF/ZPOF/ZPPTRF, respectively.
4. The array data specified for input argument *a* for these subroutines must be the same as the corresponding output arguments for SPOTRF/SPOF/SPOFCD, DPOTRF/DPOF/DPOFCD, CPOTRF/CPOF, and ZPOTRF/ZPOF, respectively.
5. The array data specified for input argument *ap* for these subroutines must be the same as the corresponding output arguments for SPPTRF, DPPTRF, CPPTRF, and ZPPTRF, respectively.
6. The matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
7. For a description of how the matrices are stored:
   - For positive definite real symmetric matrices, see "Positive Definite or Negative Definite Symmetric Matrix" on page 84.
   - For positive definite complex Hermitian matrices, see "Positive Definite or Negative Definite Complex Hermitian Matrix" on page 85.

Function

The system ***AX*** = ***B*** is solved for ***X***, where ***X*** and ***B*** are general matrices and ***A*** is a positive definite real symmetric matrix for SPOTRS/SPOSM/SPPTRS and DPOTRS/DPOSM/DPPTRS, and a positive definite complex Hermitian matrix for CPOTRS/CPOSM/CPPTRS and ZPOTRS/ZPOSM/ZPPTRS. These subroutines use the results of the factorization of matrix ***A***, produced by a preceding call to SPOTRF/SPOF/SPOFCD/SPPTRF, DPOTRF/DPOF/DPOFCD/DPPTRF, CPOTRF/CPOF/CPPTRF, or ZPOTRF/ZPOF/ZPPTRF, respectively. For a description of how ***A*** is factored, see "SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)" on page 533.

If *n* or *nrhs* is 0, no computation is performed. See references ["9" on page 1253] and ["46" on page 1255].

Error conditions

Computational Errors

None

Note: If the factorization performed by _POTRF, _POF, _POFCD, or _PPTRF failed because matrix ***A*** was not positive definite, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. *n* < 0
3. *nrhs* < 0
4. *n* > *lda*
5. *lda* ≤ 0
6. *n* > *ldb*
7. *ldb* ≤ 0

Examples

Example 1

This example shows how to solve the system ***AX*** = ***B*** for two right-hand sides, where matrix ***A*** is the same matrix factored in the Example 1 for SPOTRF and SPOF.

Call Statement and Input:

```
             UPLO   N  NRHS   A  LDA  BX  LDB  INFO
               |    |   |     |   |    |    |    |
CALL SPOTRS( 'L' , 9 , 2 ,   A , 9 , BX , 9 , INFO  )
```

or

```
            UPLO  A  LDA  N   BX  LDB  NRHS
              |   |   |   |   |    |    |
CALL SPOSM( 'L' , A , 9 , 9 , BX , 9 ,  2  )
```

A = (same as output A in Example 1)

```
         ┌             ┐
         |  9.0   45.0 |
         | 17.0   89.0 |
         | 24.0  131.0 |
         | 30.0  170.0 |
BX   =   | 35.0  205.0 |
         | 39.0  235.0 |
         | 42.0  259.0 |
         | 44.0  276.0 |
         | 45.0  285.0 |
         └             ┘
```

Output:

```
         ┌           ┐
         | 1.0  1.0 |
         | 1.0  2.0 |
         | 1.0  3.0 |
         | 1.0  4.0 |
BX   =   | 1.0  5.0 |
         | 1.0  6.0 |
         | 1.0  7.0 |
         | 1.0  8.0 |
         | 1.0  9.0 |
         └           ┘
```

```
INFO =  0
```

Example 2

This example shows how to solve the system ***AX*** = ***B*** for two right-hand sides, where matrix ***A*** is the input matrix factored in Example 2 for SPOTRF and SPOF.

Call Statement and Input:

```
             UPLO  N  NRHS A  LDA  BX  LDB  INFO
              |    |   |   |   |   |    |    |
CALL SPOTRS( 'U' , 9 , 2 , A , 9 , BX , 9 , INFO  )
```

or

```
            UPLO  A  LDA  N   BX  LDB  NRHS
             |    |   |   |   |    |    |
CALL SPOSM( 'U' , A , 9 , 9 , BX , 9 ,  2  )
```

A = (same as output A in Example 2)

```
        ┌              ┐
        |  9.0   45.0 |
        | 17.0   89.0 |
        | 24.0  131.0 |
        | 30.0  170.0 |
BX   =  | 35.0  205.0 |
        | 39.0  235.0 |
        | 42.0  259.0 |
        | 44.0  276.0 |
        | 45.0  285.0 |
        └              ┘
```

Output:

```
        ┌           ┐
        | 1.0  1.0 |
        | 1.0  2.0 |
        | 1.0  3.0 |
        | 1.0  4.0 |
BX   =  | 1.0  5.0 |
        | 1.0  6.0 |
        | 1.0  7.0 |
        | 1.0  8.0 |
        | 1.0  9.0 |
        └           ┘
```

```
INFO =  0
```

Example 3

This example shows how to solve the system ***AX*** = ***B*** for two right-hand sides, where matrix ***A*** is the same matrix factored in the Example 3 for CPOTRF and CPOF.

Call Statement and Input:

```
             UPLO  N  NRHS A  LDA  BX  LDB  INFO
              |    |   |   |   |   |    |    |
CALL CPOTRS( 'L' , 3 , 2 , A , 3 , BX , 3 , INFO  )
```

or

```
            UPLO  A  LDA  N   BX  LDB  NRHS
             |    |   |   |   |    |    |
CALL CPOSM( 'L' , A , 3 , 3 , BX , 3 ,  2  )
```

A = (same as output A in Example 3)

```
        ┌                                  ┐
        |  (60.0, -55.0)    (70.0, 10.0) |
BX   =  |   (34.0, 58.0)  (-51.0, 110.0) |
        | (13.0, -152.0)    (75.0, 63.0) |
        └                                  ┘
```

Output:

```
         ┌                        ┐
         | (2.0, -1.0)  ( 2.0, 0.0) |
BX   =   | (1.0,  1.0)  (-1.0, 2.0) |
         | (0.0, -2.0)  ( 1.0, 1.0) |
         └                        ┘
```

```
INFO =  0
```

Example 4

This example shows how to solve the system ***AX*** = ***B*** for two right-hand sides, where matrix ***A*** is the input matrix factored in Example 4 for CPOTRF and CPOF.

Call Statement and Input:

```
              UPLO   N  NRHS A   LDA   BX  LDB  INFO
               |     |   |   |    |    |    |    |
CALL CPOTRS( 'U' , 3 , 2 , A , 3 , BX , 3 , INFO  )
```

or

```
             UPLO   A  LDA  N   BX  LDB  NRHS
              |     |   |   |    |    |    |
CALL CPOSM( 'U' , A , 3 , 3 , BX , 3 ,  2  )
```

A = (same as output A in Example 4)

```
         ┌                              ┐
         | ( 33.0, -18.0)  (15.0,  -3.0) |
BX   =   | ( 45.0, -45.0)  ( 8.0,  -2.0) |
         | (152.0,   1.0)  (43.0, -29.0) |
         └                              ┘
```

Output:

```
         ┌                        ┐
         | (2.0, -1.0)  (2.0,  0.0) |
BX   =   | (1.0, -1.0)  (0.0,  1.0) |
         | (3.0,  0.0)  (1.0, -1.0) |
         └                        ┘
```

```
INFO =  0
```

Example 5

This example shows how to solve the system ***AX*** = ***B*** for two right-hand sides, where matrix ***A*** is the input matrix factored in Example 5 for SPPTRF and DPPTRF.

Call Statement and Input:

```
              UPLO   N  NRHS  AP    BX  LDB  INFO
               |     |   |     |     |    |    |
CALL SPPTRS( 'L' , 9 , 2 ,  AP , BX , 9 , INFO )
```

A = (same as output A in Example 5)

```
         ┌             ┐
         |  9.0   45.0 |
         | 17.0   89.0 |
         | 24.0  131.0 |
         | 30.0  170.0 |
BX   =   | 35.0  205.0 |
         | 39.0  235.0 |
         | 42.0  259.0 |
         | 44.0  276.0 |
         | 45.0  285.0 |
         └             ┘
```

Output:

```
         ┌          ┐
         | 1.0  1.0 |
         | 1.0  2.0 |
         | 1.0  3.0 |
         | 1.0  4.0 |
BX    =  | 1.0  5.0 |
         | 1.0  6.0 |
         | 1.0  7.0 |
         | 1.0  8.0 |
         | 1.0  9.0 |
         └          ┘
```

```
INFO =  0
```

Example 6

This example shows how to solve the system ***AX*** = ***B*** for two right-hand sides, where matrix ***A*** is the same matrix factored in Example 6 for SPPTRF.

Call Statement and Input:

```
             UPLO   N  NRHS   AP    BX  LDB  INFO
              |     |   |     |     |    |    |
CALL SPPTRS( 'U' , 9 , 2 ,  AP  , BX , 9 , INFO  )
```

A = (same as output A in Example 6)

```
         ┌             ┐
         |  9.0   45.0 |
         | 17.0   89.0 |
         | 24.0  131.0 |
         | 30.0  170.0 |
BX    =  | 35.0  205.0 |
         | 39.0  235.0 |
         | 42.0  259.0 |
         | 44.0  276.0 |
         | 45.0  285.0 |
         └             ┘
```

Output:

```
         ┌          ┐
         | 1.0  1.0 |
         | 1.0  2.0 |
         | 1.0  3.0 |
         | 1.0  4.0 |
BX    =  | 1.0  5.0 |
         | 1.0  6.0 |
         | 1.0  7.0 |
         | 1.0  8.0 |
         | 1.0  9.0 |
         └          ┘
```

```
INFO =  0
```

Example 7

This example shows how to solve the system ***AX*** = ***B*** for two right-hand sides, where matrix ***A*** is the same matrix factored in Example 7 for ZPPTRF.

Call Statement and Input:

```
             UPLO   N  NRHS   AP    BX  LDB  INFO
              |     |   |     |     |    |    |
CALL ZPPTRS( 'L' , 3 , 2 ,  AP  , BX , 3 , INFO )
```

AP = (same as output AP in Example 7)

```
        ┌                                  ┐
        | (60.0,  -55.0)  ( 70.0,  10.0) |
BX   =  | (34.0,   58.0)  (-51.0, 110.0) |
        | (13.0, -152.0)  ( 75.0,  63.0) |
        └                                  ┘
```

Output:

```
        ┌                              ┐
        | (2.0, -1.0)  ( 2.0, 0.0) |
BX   =  | (1.0,  1.0)  (-1.0, 2.0) |
        | (0.0, -2.0)  ( 1.0, 1.0) |
        └                              ┘
```

```
INFO =  0
```

Example 8

This example shows how to solve the system ***AX*** = ***B*** for two right-hand sides, where matrix ***A*** is the same matrix factored in Example 8 for ZPPTRF.

Call Statement and Input:

```
             UPLO  N  NRHS  AP   BX  LDB  INFO
              |    |   |    |    |    |    |
CALL ZPPTRS( 'U' , 3 , 2 ,  AP , BX , 3 , INFO )
```

AP = (same as output AP in Example 8)

```
        ┌                                 ┐
        | ( 33.0, -18.0) (15.0,  -3.0) |
BX   =  | ( 45.0, -45.0) ( 8.0,  -2.0) |
        | (152.0,   1.0) (43.0, -29.0) |
        └                                 ┘
```

Output:

```
        ┌                            ┐
        | (2.0, -1.0) (2.0,  0.0) |
BX   =  | (1.0, -1.0) (0.0,  1.0) |
        | (3.0,  0.0) (1.0, -1.0) |
        └                            ┘
```

```
INFO =  0
```

SPPS and DPPS (Positive Definite Real Symmetric Matrix Solve)

Purpose

These subroutines solve the system ***Ax*** = ***b*** for ***x***, where ***A*** is a positive definite symmetric matrix, and ***x*** and ***b*** are vectors. The subroutines use the results of the factorization of matrix ***A***, produced by a preceding call to SPPF/SPPFCD or DPPF/DPPFP/DPPFCD, respectively.

Table 141: Data Types.

Data Types	
A, ***b***, ***x***	**Subroutine**
Short-precision real	SPPS
Long-precision real	DPPS

Note: The input to these solve subroutines must be the output from the factorization subroutines SPPF/SPPFCD and DPPF/DPPFP/DPPFCD, respectively.

Syntax

Language	Syntax
Fortran	CALL SPPS | DPPS (*ap*, *n*, *bx*, *iopt*)
C and C++	spps | dpps (*ap*, *n*, *bx*, *iopt*);

On Entry

ap

is the factorization of matrix ***A***, produced by a preceding call to SPPF/SPPFCD or DPPF/DPPFP/DPPFCD, respectively.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 141 on page 552, where:

If *iopt* = 0, the array must contain $n(n+1)/2+n$ elements.

If *iopt* = 1, the array must contain $n(n+1)/2$ elements.

n

is the order of matrix ***A*** used in the factorization, and the lengths of vectors ***b*** and ***x***.

Specified as: an integer; $n \ge 0$.

bx

is the vector ***b*** of length *n*, containing the right-hand side of the system.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 141 on page 552.

iopt

indicates the type of factorization that was performed on matrix ***A***, where:

If *iopt* = 0, the matrix was factored using the $\boldsymbol{LDL}^{\mathrm{T}}$ method.

If *iopt* = 1, the matrix was factored using Cholesky factorization.

Specified as: an integer; *iopt* = 0 or 1.

On Return

bx

is the solution vector ***x*** of length *n*, containing the results of the computation.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 141 on page 552.

Notes

1. The array data specified for input argument *ap* for these subroutines must be the same as the corresponding output argument for SPPF/SPPFCD and DPPF/DPPFP/DPPFCD, respectively.
2. The scalar data specified for input argument *n* for these subroutines must be the same as that specified for SPPF/SPPFCD and DPPF/DPPFP/DPPFCD, respectively.
3. When you call these subroutines after calling SPPF or DPPF, the value of input argument *iopt* must be as follows:

SPPF/DPPF Input *iopt*	SPPS/DPPS Input *iopt*
0 or 10	0

SPPF/DPPF Input *iopt*	**SPPS/DPPS Input *iopt***
1 or 11	1

4. When you call these subroutines after calling SPPFCD or DPPFCD, the value of input argument *iopt* must be 0.
5. When you call these subroutines after calling DPPFP, the value of input argument *iopt* must be 1.
6. In the input array specified for *ap*, the first $n(n+1)/2$ elements are matrix elements. The additional *n* locations, required in the array when *iopt* = 0, are used for working storage by this subroutine and should not be altered between calls to the factorization and solve subroutines.
7. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
8. For a description of how a positive definite symmetric matrix is stored in lower-packed storage mode in an array, see "Symmetric Matrix" on page 80.

Function

The system $\boldsymbol{Ax} = \boldsymbol{b}$ is solved for $\boldsymbol{x}$, where $\boldsymbol{A}$ is a positive definite symmetric matrix, stored in lower-packed storage mode in array AP, and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors. These subroutines use the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SPPF/SPPFCD or DPPF/DPPFP/DPPFCD, respectively.

If *n* is 0, no computation is performed. See references ["46" on page 1255] and ["48" on page 1255].

Error conditions

Computational Errors

None

Note: If a call to SPPF, DPPF, SPPFCD, DPPFCD, or DPPFP resulted in a nonpositive definite matrix, error 2104 or 2115, SPPS or DPPS results may be unpredictable or numerically unstable.

Input-Argument Errors

1. $n < 0$
2. *iopt* ≠ 0 or 1

Examples

Example 1

This example shows how to solve the system $\boldsymbol{Ax} = \boldsymbol{b}$, where matrix $\boldsymbol{A}$ is the same matrix factored in the Example 9 for SPPF and DPPF.

Call Statement and Input:

```
            AP   N   BX   IOPT
            |    |   |     |
CALL SPPS ( AP , 9 , BX ,  0 )
```

AP = (same as output AP in Example 9 for SPPF and DPPF)

```
BX = (9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0)
```

Output:

```
BX       =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

Example 2

This example shows how to solve the same system as in Example 1, where matrix $\boldsymbol{A}$ is the same matrix factored in the Example 10 for SPPF and DPPF.

Call Statement and Input:

```
           AP   N   BX  IOPT
           |    |   |    |
CALL SPPS( AP , 9 , BX , 1 )
```

AP = (same as output AP in Example 10 for SPPF and DPPF)
BX = (9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0)

Output:

```
BX       =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

SPOCON, DPOCON, CPOCON, ZPOCON, SPPCON, DPPCON, CPPCON, and ZPPCON (Estimate the Reciprocal of the Condition Number of a Positive Definite Real Symmetric or Complex Hermitian Matrix)

Purpose

These subroutines estimate the reciprocal of the condition number of matrix ***A*** as explained below:

SPOCON, DPOCON, CPOCON, and ZPOCON

The SPOCON, DPOCON, CPOCON, and ZPOCON subroutines estimate the reciprocal of the condition number of matrix ***A***, stored in upper or lower storage mode, where:

- For SPOCON and DPOCON, ***A*** is a positive definite real symmetric matrix.
- For CPOCON and ZPOCON, ***A*** is a positive definite complex Hermitian matrix.

These subroutines use the results of the factorization of matrix ***A*** produced by a preceding call to SPOTRF, DPOTRF, CPOTRF, or ZPOTRF, respectively.

SPPCON, DPPCON, CPPCON, and ZPPCON

The SPPCON, DPPCON, CPPCON, and ZPPCON subroutines estimate the reciprocal of the condition number of matrix ***A***, stored in upper-packed or lower-packed storage mode, where:

- For SPPCON and DPPCON, ***A*** is a positive definite real symmetric matrix.
- For CPPCON and ZPPCON, ***A*** is a positive definite complex Hermitian matrix.

These subroutines use the results of the factorization of matrix ***A*** produced by a preceding call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF, respectively.

For details about the factorization subroutines, see "SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)" on page 533.

Table 142: Data Types.

Data Types		
A, *work*	*anorm*, *rcond*, *rwork*	**Subroutine**
Short-precision real	Short-precision real	SPOCON^, SPPCON^
Long-precision real	Long-precision real	DPOCON^, DPPCON^
Short-precision complex	Short-precision real	CPOCON^, CPPCON^
Long-precision complex	Long-precision real	ZPOCON^, ZPPCON^

Syntax

Language	Syntax
Fortran	CALL SPOCON | DPOCON (*uplo*, *n*, *a*, *lda*, *anorm*, *rcond*, *work*, *iwork*, *info*) CALL CPOCON | ZPOCON (*uplo*, *n*, *a*, *lda*, *anorm*, *rcond*, *work*, *rwork*, *info*) CALL SPPCON | DPPCON (*uplo*, *n*, *ap*, *anorm*, *rcond*, *work*, *iwork*, *info*) CALL CPPCON | ZPPCON (*uplo*, *n*, *ap*, *anorm*, *rcond*, *work*, *rwork*, *info*)

C and C++	spocon | dpocon (*uplo*, *n*, *a*, *lda*, *anorm*, *rcond*, *work*, *iwork*, *info*);
	cpocon | zpocon (*uplo*, *n*, *a*, *lda*, *anorm*, *rcond*, *work*, *rwork*, *info*);
	sppcon | dppcon (*uplo*, *n*, *ap*, *anorm*, *rcond*, *work*, *iwork*, *info*);
	cppcon | zppcon (*uplo*, *n*, *ap*, *anorm*, *rcond*, *work*, *rwork*, *info*);

LAPACKE	info = LAPACKE_spocon | LAPACKE_dpocon (*matrix_layout*, *uplo*, *n*, *a*, *lda*, *anorm*, *rcond*);
	info = LAPACKE_cpocon | LAPACKE_zpocon (*matrix_layout*, *uplo*, *n*, *a*, *lda*, *anorm*, *rcond*);
	info = LAPACKE_sppcon | LAPACKE_dppcon (*matrix_layout*, *uplo*, *n*, *ap*, *anorm*, *rcond*);
	info = LAPACKE_cppcon | LAPACKE_zppcon (*matrix_layout*, *uplo*, *n*, *ap*, *anorm*, *rcond*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

uplo

indicates whether matrix ***A*** is stored in upper or lower storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

n

the order of the factored matrix ***A*** used in the computation.

Specified as: an integer; $n \geq 0$.

ap

is an array, referred to as AP, containing the factorization of the positive definite matrix ***A*** produced by a preceding call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF, respectively, stored in upper-packed or lower-packed storage mode.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 142 on page 555. See "Notes" on page 558.

a

the factorization of positive definite matrix ***A*** produced by a preceding call to SPOTRF, DPOTRF, CPOTRF, or ZPOTRF, respectively, stored in upper or lower storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 142 on page 555.

lda

is the leading dimension of matrix ***A***.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

anorm
is the one norm of the original matrix ***A***.

For SPOCON, DPOCON, CPOCON, and ZPOCON

To obtain the value of *anorm*, make a preceding call to SLANSY, DLANSY, CLANHE, or ZLANHE, respectively.

For SPPCON, DPPCON, CPPCON, and ZPPCON

To obtain the value of *anorm*, make a preceding call to SLANSP, DLANSP, CLANHP, or ZLANHP, respectively.

Refer to "SLANSY, DLANSY, CLANSY, ZLANSY, CLANHE, ZLANHE, SLANSP, DLANSP, CLANHP, and ZLANHP (Real Symmetric or Complex Hermitian Matrix Norm)" on page 579.

Specified as: a number ≥ ≥ 0.0, of the data type indicated in Table 142 on page 555.

rcond
See On Return.

work
is the work area used by this subroutine, where:

For SPOCON, DPOCON, SPPCON, and DPPCON
The size of *work* is (at least) of length 3*n*.

For CPOCON, ZPOCON, CPPCON, and ZPPCON
The size of *work* is (at least) of length 2*n*.

Specified as: an area of storage containing numbers of data type indicated in Table 142 on page 555.

iwork
is a work area used by this subroutine whose size is (at least) of length *n*.

Specified as: an area of storage containing integers.

rwork
is a work area used by this subroutine whose size is (at least) of length *n*.

Specified as: an area of storage containing numbers of the data type indicated in Table 142 on page 555.

info
See On Return.

On Return

rcond
has the following meaning:

If *info* = 0, an estimate of the reciprocal of the condition number of matrix ***A*** is returned; i.e., *rcond* = 1.0/(NORM(A) × NORM(A^{-1})).

If *n* = 0, the subroutines return with *rcond* = 1.0.

If *n* ≠ 0 and *anorm* = 0.0, the subroutines return with *rcond* = 0.0.

Returned as: a number ≥ ≥ 0.0, of the data type indicated in Table 142 on page 555.

info
has the following meaning:

If *info* = 0, the computation completed normally.

Returned as:

- For SPOCON, DPOCON, CPOCON, ZPOCON, SPPCON, DPPCON, CPPCON, and ZPPCON, returned as: an integer; info ≥ 0.

- For LAPACKE_spocon, LAPACKE_dpocon, LAPACKE_cpocon, LAPACKE_zpocon, LAPACKE_sppcon, LAPACKE_dppcon, LAPACKE_cppcon, and LAPACKE_zppcon, returned as an integer function value; info ≥ 0.

Returned as: an integer; *info* = 0.

Notes

1. In your C program, arguments *rcond* and *info* must be passed by reference.
2. This subroutine accepts lowercase letters for the *uplo* argument.
3. For input arguments *uplo*, *lda*, and *n*, the following must be true:

   For SPOCON/DPOCON/CPOCON/ZPOCON
   The scalar data specified for *uplo*, *lda*, and *n* must be the same as the scalar data specified for SLANSY/DLANSY/CLANHE/ZLANHE and SPOTRF/DPOTRF/CPOTRF/ZPOTRF.

   For SPPCON/DPPCON/CPPCON/ZPPCON
   The scalar data specified for *uplo* and *n* must be the same as the scalar data specified for SLANSP/DLANSP/CLANHP/ZLANHP and SPPTRF/DPPTRF/CPPTRF/ZPPTRF.
4. For matrix ***A***, the following must be true:

   For SPOCON/DPOCON/CPOCON/ZPOCON
   The matrix ***A*** input to SLANSY/DLANSY/CLANHE/ZLANHE must be the same as the corresponding input argument for SPOTRF/DPOTRF/CPOTRF/ZPOTRF.

   For SPPCON/DPPCON/CPPCON/ZPPCON
   The matrix ***A*** input to SLANSP/DLANSP/CLANHP/ZLANHP must be the same as the corresponding input argument for SPPTRF/DPPTRF/CPPTRF/ZPPTRF.
5. On both input and output, matrix ***A*** conforms to LAPACK format.
6. For a description of the storage modes used for the matrices, see:
   - For positive definite symmetric matrices, see "Positive Definite or Negative Definite Symmetric Matrix" on page 84.
   - For positive definite complex Hermitian matrices, see "Positive Definite or Negative Definite Complex Hermitian Matrix" on page 85.

Function

The reciprocal of the condition number of general matrix ***A*** is estimated, using the results of the factorization of matrix ***A*** produced by a preceding factorization call.

rcond = 1.0/(NORM(A) × NORM(A^{-1})).

If $n = 0$, the subroutines return with *rcond* = 1.0.

If $n \neq 0$ and *anorm* = 0.0, the subroutines return with *rcond* = 0.0.

See reference ["84" on page 1257].

Error conditions

Resource Errors

None.

Computational Errors

None.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. $n < 0$
3. $n > lda$

4. *lda* ≤ 0
5. *anorm* < 0
6. *anorm* ≠ 0 and *anorm* > *big* or *anorm* < *tiny*

   Where:

   For SPOCON, SPPCON, CPOCON, and CPPCON
   big and *tiny* have the following values:

   $big = 2^{127} \times (1 - \text{ULP})$
   $tiny = 2^{-126} \times 2^{21}$

   For DPOCON, DPPCON, ZPOCON, and ZPPCON
   big and *tiny* have the following values:

   $big = 2^{1023} \times (1 - \text{ULP})$
   $tiny = 2^{-1022} \times 2^{50}$

   Where ULP = unit in last place.

   Note: To avoid this error, scale matrix ***A*** so that *tiny* ≤ *anorm* ≤ *big*.

Examples

Example 1

This example estimates the reciprocal of the condition number of positive definite real symmetric matrix ***A*** stored in lower storage mode. The input matrix ***A*** to DLANSY and DPOTRF is the same as input matrix ***A*** in Example 1.

Call Statements and Input:

```
                 NORM  UPLO  N   A  LDA  WORK
                   |     |   |   |   |     |
ANORM = DLANSY(  '1',  'L', 9 , A , 9 , WORK )
```

```
              UPLO  N   A  LDA   INFO
                |   |   |   |      |
CALL DPOTRF(  'L', 9 , A , 9 ,  INFO )
```

```
              UPLO  N   A  LDA  ANORM  RCOND  WORK  IWORK   INFO
                |   |   |   |     |      |      |     |       |
CALL DPOCON(  'L', 9 , A , 9 , ANORM, RCOND, WORK, IWORK , INFO )
```

A = (same as output A in Example 1)

ANORM = (same as output ANORM in Example 1)

Output:

RCOND = 5.56×10^{-3}

INFO = 0

Example 2

This example estimates the reciprocal of the condition number of positive definite real symmetric matrix ***A*** stored in upper storage mode. The input matrix ***A*** to DLANSY and DPOTRF is the same as input matrix ***A*** in Example 2.

Call Statements and Input:

```
                 NORM  UPLO  N   A  LDA  WORK
                   |     |   |   |   |     |
ANORM = DLANSY(  '1',  'U' ,9 , A , 9 , WORK )
```

```
              UPLO  N   A  LDA   INFO
                |   |   |   |      |
CALL DPOTRF(  'U', 9 , A , 9 ,  INFO )
```

```
              UPLO  N   A  LDA  ANORM  RCOND  WORK  IWORK   INFO
                |   |   |   |     |      |      |     |       |
CALL DPOCON(  'U', 9 , A , 9 , ANORM, RCOND, WORK, IWORK , INFO )
```

A = (same as output A in Example 2)

ANORM = (same as output ANORM in Example 2)

Output:

RCOND = 5.56×10^{-3}

INFO = 0

Example 3

This example estimates the reciprocal of the condition number of positive definite complex Hermitian matrix ***A*** stored in lower storage mode. The input matrix ***A*** to ZLANHE and ZPOTRF is the same as input matrix ***A*** in Example 3.

Call Statements and Input:

```
                 NORM  UPLO  N   A  LDA  WORK
                   |     |   |   |   |     |
ANORM = ZLANHE(  '1',  'L', 3 , A , 3 , RWORK )
```

```
              UPLO  N   A  LDA  INFO
                |   |   |   |     |
CALL ZPOTRF(  'L', 3 , A , 3 , INFO )
```

```
              UPLO  N   A  LDA  ANORM  RCOND  WORK  RWORK  INFO
                |   |   |   |     |      |      |     |      |
CALL ZPOCON(  'L', 3 , A , 3 , ANORM, RCOND, WORK, RWORK, INFO )
```

A = (same as output A in Example 3)

ANORM = (same as output ANORM in Example 3)

Output:

RCOND = 1.85×10^{-1}

INFO = 0

Example 4

This example estimates the reciprocal of the condition number of positive definite complex Hermitian matrix ***A*** stored in upper storage mode. The input matrix ***A*** to ZLANHE and ZPOTRF is the same as input matrix ***A*** in Example 4.

Call Statements and Input:

```
                 NORM  UPLO  N   A  LDA  WORK
                  |     |    |   |   |     |
ANORM = ZLANHE(  '1',  'U', 3 , A , 3 , RWORK )
```

```
              UPLO  N   A  LDA  INFO
               |    |   |   |     |
CALL ZPOTRF(  'U', 3 , A , 3 , INFO )
```

```
              UPLO  N   A  LDA  ANORM  RCOND  WORK  RWORK  INFO
               |    |   |   |     |      |      |     |      |
CALL ZPOCON(  'U', 3 , A , 3 , ANORM, RCOND, WORK, RWORK, INFO )
```

A = (same as output A in Example 3)

ANORM = (same as output ANORM in Example 4)

Output:

RCOND = 1.01×10^{-1}

INFO = 0

Example 5

This example estimates the reciprocal of the condition number of positive definite real symmetric matrix ***A*** stored in lower-packed storage mode. The input matrix ***A*** to DLANSP and DPPTRF is the same as input matrix ***A*** in Example 5.

Call Statements and Input:

```
                 NORM  UPLO  N   AP   WORK
                  |     |    |    |     |
ANORM = DLANSP(  '1',  'L', 9 , AP , WORK )
```

```
              UPLO  N   AP   INFO
               |    |    |     |
CALL DPPTRF(  'L', 9 , AP , INFO )
```

```
              UPLO  N   AP  ANORM  RCOND  WORK  IWORK   INFO
               |    |    |    |      |      |     |       |
CALL DPPCON(  'L', 9 , AP, ANORM, RCOND, WORK, IWORK , INFO )
```

AP = (same as output AP in Example 5)

ANORM = (same as output ANORM in Example 5)

Output:

RCOND = 5.56×10^{-3}

INFO = 0

Example 6

This example estimates the reciprocal of the condition number of positive definite real symmetric matrix ***A*** stored in upper-packed storage mode. The input matrix ***A*** to DLANSP and DPPTRF is the same as input matrix ***A*** in Example 6.

Call Statements and Input:

```
                NORM  UPLO   N   AP   WORK
                  |     |    |    |     |
ANORM = DLANSP(  '1',  'U' , 9 , AP , WORK )
```

```
             UPLO  N   AP   INFO
               |   |    |     |
CALL DPPTRF(  'U', 9 , AP , INFO )
```

```
             UPLO  N   AP  ANORM  RCOND  WORK  IWORK   INFO
               |   |    |    |      |     |     |       |
CALL DPPCON(  'U', 9 , AP, ANORM, RCOND, WORK, IWORK , INFO )
```

AP = (same as output AP in Example 6)

ANORM = (same as output ANORM in Example 6)

Output:

RCOND = 5.56×10^{-3}

INFO = 0

Example 7

This example estimates the reciprocal of the condition number of positive definite complex Hermitian matrix ***A*** stored in lower-packed storage mode. The input matrix ***A*** to ZLANHP and ZPPTRF is the same as input matrix ***A*** in Example 7.

Call Statements and Input:

```
                NORM  UPLO   N   AP   WORK
                  |     |    |    |     |
ANORM = ZLANHP(  '1',  'L' , 3 , AP , RWORK )
```

```
             UPLO   N   AP   INFO
               |    |    |     |
CALL ZPPTRF(  'L' , 3 , AP , INFO )
```

```
             UPLO  N   AP  ANORM  RCOND  WORK  RWORK  INFO
               |   |    |    |      |     |     |      |
CALL ZPPCON(  'L', 3 , AP ,ANORM, RCOND, WORK, RWORK, INFO )
```

AP = (same as output AP in Example 7)

ANORM = (same as output ANORM in Example 7)

Output:

RCOND = 1.85×10^{-1}

INFO = 0

Example 8

This example estimates the reciprocal of the condition number of positive definite complex Hermitian matrix ***A*** stored in upper-packed storage mode. The input matrix ***A*** to ZLANHP and ZPPTRF is the same as input matrix ***A*** in Example 8.

Call Statements and Input:

```
                NORM  UPLO   N   AP   WORK
                  |     |    |    |     |
ANORM = ZLANHP(  '1',  'U' , 3 , AP , RWORK )
```

```
             UPLO  N   AP   INFO
               |   |    |     |
CALL ZPPTRF(  'U', 3 , AP , INFO )
```

```
             UPLO  N   AP  ANORM  RCOND  WORK  RWORK  INFO
               |   |    |    |      |      |     |      |
CALL ZPPCON(  'U', 3 , AP ,ANORM, RCOND, WORK, RWORK, INFO )
```

AP = (same as output AP in Example 8)

ANORM = (same as output ANORM in Example 8)

Output:

RCOND = 1.01×10^{-1}

INFO = 0

SPPFCD, DPPFCD, SPOFCD, and DPOFCD (Positive Definite Real Symmetric Matrix Factorization, Condition Number Reciprocal, and Determinant)

Purpose

The SPPFCD and DPPFCD subroutines factor positive definite symmetric matrix ***A***, stored in lower-packed storage mode, using Gaussian elimination ($\boldsymbol{LDL}^T$). The reciprocal of the condition number and the determinant of matrix ***A*** can also be computed. To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SPPS or DPPS, respectively.

The SPOFCD and DPOFCD subroutines factor positive definite symmetric matrix ***A***, stored in upper or lower storage mode, using Cholesky factorization ($\boldsymbol{LL}^T$ or $\boldsymbol{U}^T\boldsymbol{U}$). The reciprocal of the condition number and the determinant of matrix ***A*** can also be computed. To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with a call to SPOSM or DPOSM, respectively. To find the inverse of matrix ***A***, follow the call to these subroutines with a call to SPOICD or DPOICD, respectively.

Table 143: Data Types.

Data Types	
A, *aux*, *rcond*, ***det***	**Subroutine**
Short-precision real	SPPFCD and SPOFCD
Long-precision real	DPPFCD and DPOFCD

Note: The output factorization from SPPFCD and DPPFCD should be used only as input to the solve subroutines SPPS and DPPS, respectively. The output from SPOFCD and DPOFCD should be used only as input to the following subroutines for performing a solve or inverse: SPOSM/SPOICD and DPOSM/DPOICD, respectively.

Syntax

Language	**Syntax**
Fortran	CALL SPPFCD | DPPFCD (*ap*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*) CALL SPOFCD | DPOFCD (*uplo*, *a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*)

C and C++	sppfcd | dppfcd (*ap*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*);
	spofcd | dpofcd (*uplo*, *a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*);

On Entry

uplo

indicates whether matrix ***A*** is stored in upper or lower storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

ap

is the array, referred to as AP, in which the matrix ***A***, to be factored, is stored in lower-packed storage mode.

Specified as: a one-dimensional array of (at least) length $n(n+1)/2+n$, containing numbers of the data type indicated in Table 143 on page 563.

a

is the positive definite symmetric matrix ***A***, to be factored.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 143 on page 563.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

n

is the order *n* of matrix ***A***.

Specified as: an integer, where:

For SPPFCD and DPPFCD, $n \geq 0$.

For SPOFCD and DPOFCD, $0 \leq n \leq lda$.

iopt

indicates the type of computation to be performed, where:

If *iopt* = 0, the matrix is factored.

If *iopt* = 1, the matrix is factored, and the reciprocal of the condition number is computed.

If *iopt* = 2, the matrix is factored, and the determinant is computed.

If *iopt* = 3, the matrix is factored and the reciprocal of the condition number and the determinant are computed.

Specified as: an integer; *iopt* = 0, 1, 2, or 3.

rcond

See On Return.

det

See On Return.

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, is the storage work area used by these subroutines. Its size is specified by *naux*. Specified as: an area of storage, containing numbers of the data type indicated in Table 143 on page 563.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SPPFCD, DPPFCD, SPOFCD, and DPOFCD dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux* ≥ *n*.

On Return

ap

is the transformed matrix ***A*** of order *n*, containing the results of the factorization. See "Function" on page 566. Returned as: a one-dimensional array of (at least) length $n(n+1)/2+n$, containing numbers of the data type indicated in Table 143 on page 563.

a

is the transformed matrix ***A*** of order *n*, containing the results of the factorization. See "Function" on page 566. Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 143 on page 563.

rcond

is the estimate of the reciprocal of the condition number, *rcond*, of matrix ***A***. Returned as: a number of the data type indicated in Table 143 on page 563; *rcond* ≥ 0.

det

is the vector ***det***, containing the two components det_1 and det_2 of the determinant of matrix ***A***. The determinant is:

$$det_1\left(10^{det_2}\right)$$

where $1 \le det_1 < 10$. Returned as: an array of length 2, containing numbers of the data type indicated in Table 143 on page 563.

Notes

1. All subroutines accept lowercase letters for the *uplo* argument.
2. In your C program, argument *rcond* must be passed by reference.
3. When *iopt* = 0, SPPFCD and DPPFCD provide the same function as a call to SPPF or DPPF, respectively. When *iopt* = 0, SPOFCD and DPOFCD provide the same function as a call to SPOF or DPOF, respectively.
4. SPPFCD and DPPFCD in many cases utilize new algorithms based on recursive packed storage format. As a result, on output, the array specified for AP may be stored in this new format rather than the conventional lower packed format. (See references ["63" on page 1256], ["79" on page 1257], and ["81" on page 1257]).

   The array specified for AP should not be altered between calls to the factorization and solve subroutines; otherwise unpredictable results may occur.
5. See "Notes" on page 553 for information on specifying a value for *iopt* in the SPPS and DPPS subroutines after calling SPPFCD and DPPFCD, respectively.

6. In the input and output arrays specified for *ap*, the first $n(n+1)/2$ elements are matrix elements. The additional *n* locations in the array are used for working storage by this subroutine and should not be altered between calls to the factorization and solve subroutines.
7. For a description of how a positive definite symmetric matrix is stored in lower-packed storage mode in an array, see “Symmetric Matrix” on page 80. For a description of how a positive definite symmetric matrix is stored in upper or lower storage mode, see “Positive Definite or Negative Definite Symmetric Matrix” on page 84.
8. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 49.

Function

The functions for these subroutines are described.

For SPPFCD and DPPFCD

The positive definite symmetric matrix $\boldsymbol{A}$, stored in lower-packed storage mode, is factored using Gaussian elimination, where $\boldsymbol{A}$ is expressed as:

$\boldsymbol{A} = \boldsymbol{LDL}^{T}$

where:

$\boldsymbol{L}$ is a unit lower triangular matrix.
$\boldsymbol{L}^{T}$ is the transpose of matrix $\boldsymbol{L}$.
$\boldsymbol{D}$ is a diagonal matrix.

An estimate of the reciprocal of the condition number, *rcond*, and the determinant, ***det***, can also be computed by this subroutine. The estimate of the condition number uses an enhanced version of the algorithm described in references [“83” on page 1257] and [“84” on page 1257].

If *n* is 0, no computation is performed. See references [“46” on page 1255] and [“48” on page 1255].

These subroutines call SPPF and DPPF, respectively, to perform the factorization using Gaussian elimination ($\boldsymbol{LDL}^{T}$). If you want to use the Cholesky factorization method, you must call SPPF and DPPF directly.

For SPOFCD and DPOFCD

The positive definite symmetric matrix $\boldsymbol{A}$, stored in upper or lower storage mode, is factored using Cholesky factorization, where $\boldsymbol{A}$ is expressed as:

$\boldsymbol{A} = \boldsymbol{LL}^{T}$ or $\boldsymbol{A} = \boldsymbol{U}^{T}\boldsymbol{U}$

where:

$\boldsymbol{L}$ is a lower triangular matrix.
$\boldsymbol{L}^{T}$ is the transpose of matrix $\boldsymbol{L}$.
$\boldsymbol{U}$ is an upper triangular matrix.
$\boldsymbol{U}^{T}$ is the transpose of matrix $\boldsymbol{U}$.

If specified, the estimate of the reciprocal of the condition number and the determinant can also be computed. The estimate of the condition number uses an enhanced version of the algorithm described in references [“83” on page 1257] and [“84” on page 1257].

If *n* is 0, no computation is performed. See references [“9” on page 1253] and [“46” on page 1255].

Error conditions

Resource Errors

Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors

1. Matrix ***A*** is not positive definite (for SPPFCD and DPPFCD).
   - If matrix ***A*** is singular (at least one of the diagonal elements are 0), then *rcond* and *det*, if you requested them, are set to 0.
   - If matrix ***A*** is nonsingular and nonpositive definite (none of the diagonal elements are 0 and at least one diagonal element is negative), then *rcond* and ***det***, if you requested them, are computed.
   - One or more elements of ***D*** contain values less than or equal to 0; all elements of ***D*** are checked. The index *i* of the last nonpositive element encountered is identified in the computational error message, issued by SPPF or DPPF, respectively.
   - *i* can be determined at run time by using the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2104 in the ESSL error option table; otherwise, the default value causes your program to be terminated by SPPF or DPPF, respectively, when this error occurs. If your program is not terminated by SPPF or DPPF, respectively, the return code is set to 2. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.
2. Matrix ***A*** is not positive definite (for SPOFCD and DPOFCD).
   - If matrix ***A*** is singular (at least one of the diagonal elements are 0), then *rcond* and *det*, if you requested them, are set to 0.
   - If matrix ***A*** is nonsingular and nonpositive definite (none of the diagonal elements are 0 and at least one diagonal element is negative), then *rcond* and ***det***, if you requested them, are computed.
   - Processing stops at the first occurrence of a nonpositive definite diagonal element.
   - The order *i* of the **first** minor encountered having a nonpositive determinant is identified in the computational error message.
   - *i* can be determined at run time by using the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2115 in the ESSL error option table; otherwise, the default value causes your program to be terminated by SPPF or DPPF, respectively, when this error occurs. If your program is not terminated by SPPF or DPPF, respectively, the return code is set to 2. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. $lda \leq 0$
3. $lda < n$
4. $n < 0$
5. *iopt* ≠ 0, 1, 2, or 3
6. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example computes the factorization, reciprocal of the condition number, and determinant of matrix ***A***. The input is the same as used in Example 9 for SPPF.

The values used to estimate the reciprocal of the condition number are obtained with the following values:

$\|\boldsymbol{A}\|_1$ = max(9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0) = 45.0
Estimate of $\|\boldsymbol{A}\|$ = `4.0`

On output, the value in ***det***, |***A***|, is equal to 1.

Call Statement and Input:

```
              AP    N  IOPT  RCOND   DET   AUX  NAUX
              |     |   |      |      |     |    |
CALL DPPFCD( AP , 9 , 3  , RCOND , DET , AUX , 9  )
```

AP =(same as input AP in
Example 9)

Output:

AP =(same as output AP in Example 9)
`RCOND = 0.0055555`
`DET = (1.0, 0.0)`

Example 2

This example computes the factorization, reciprocal of the condition number, and determinant of matrix ***A***. The input is the same as used in Example 1 for SPOF.

The values used to estimate the reciprocal of the condition number are obtained with the following values:

$\|\boldsymbol{A}\|_1$ = max(9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0) = 45.0
Estimate of $\|\boldsymbol{A}\|$ = `4.0`

On output, the value in ***det***, |***A***|, is equal to 1.

Call Statement and Input:

```
             UPLO A  LDA  N IOPT  RCOND   DET   AUX  NAUX
              |   |   |   |   |     |      |     |    |
CALL SPOFCD( 'L', A , 9 , 9 , 3 , RCOND , DET , AUX , 9  )
```

A =(same as input A in
Example 1)

Output:

A =(same as output A in Example 1)
`RCOND = 0.0055555`
`DET = (1.0, 0.0)`

Example 3

This example computes the factorization, reciprocal of the condition number, and determinant of matrix ***A***. The input is the same as used in Example 2 for SPOF.

The values used to estimate the reciprocal of the condition number are obtained with the following values:

$\|\boldsymbol{A}\|_1$ = max(9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0) = 45.0
Estimate of $\|\boldsymbol{A}\|$ = `4.0`

On output, the value in ***det***, |***A***|, is equal to 1.

Call Statement and Input:

```
             UPLO A  LDA  N IOPT  RCOND   DET   AUX  NAUX
              |   |   |   |   |     |      |     |    |
CALL SPOFCD( 'U', A , 9 , 9 , 3 , RCOND , DET , AUX , 9  )
```

A =(same as input A in
Example 2)

Output:

```
A =(same as output A in Example 2)
RCOND = 0.0055555
DET = (1.0, 0.0)
```

SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPPTRI, DPPTRI, CPPTRI, ZPPTRI, SPPICD, and DPPICD (Positive Definite Real Symmetric or Complex Hermitian Matrix Inverse, Condition Number Reciprocal, and Determinant)

Purpose

These subroutines find the inverse of a positive definite real symmetric or complex Hermitian matrix ***A*** using Cholesky factorization, where:

- For SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, and DPOICD, ***A*** is stored in upper or lower storage mode.
- For SPPTRI, DPPTRI, CPPTRI, and ZPPTRI, ***A*** is stored in upper- or lower-packed storage mode.
- For SPPICD and DPPICD, ***A*** is stored in lower-packed storage mode.

Subroutines SPOICD, DPOICD, SPPICD, and DPPICD also find the reciprocal of the condition number and the determinant of matrix ***A***.

Table 144: Data Types.

A*, *aux*, *rcond*, *det	**Subroutine**
Short-precision real	SPOTRI^, SPOICD, SPPTRI^, and SPPICD
Long-precision real	DPOTRI^, DPOICD, DPPTRI^, and DPPICD
Short-precision complex	CPOTRI^ and CPPTRI^
Long-precision complex	ZPOTRI^ and ZPPTRI^
^LAPACK	

Note: For each of the _POTRI and _PPTRI subroutines, the input must be the output from the corresponding _POTRF or _PPTRF Cholesky factorization subroutine.

If you call the subroutines SPOICD, DPOICD, SPPICD, and DPPICD with *iopt* = 4, the input must be the output from SPPF, DPPF, SPOF/SPOFCD, or DPOF/DPOFCD, respectively, where Cholesky factorization was performed.

Syntax

Table 145: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SPOTRI | DPOTRI | CPOTRI | ZPOTRI (*uplo*, *n*, *a*, *lda*, *info*) CALL SPOICD | DPOICD (*uplo*, *a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*) CALL SPPTRI | DPPTRI | CPPTRI | ZPPTRI (*uplo*, *n*, *ap*, *info*) CALL SPPICD | DPPICD (*ap*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*)

C and C++	spotri | dpotri | cpotri | zpotri (*uplo*, *n*, *a*, *lda*, *info*);
	spoicd | dpoicd (*uplo*, *a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*);
	spptri | dpptri | cpptri | zpptri (*uplo*, *n*, *ap*, *info*);
	sppicd | dppicd (*ap*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*);

Table 145: Syntax for supported computing languages. (continued)

Language	Syntax

LAPACKE	info = LAPACKE_spotri | LAPACKE_dpotri | LAPACKE_cpotri | LAPACKE_zpotri (*matrix_layout*, *uplo*, *n*, *a*, *lda*);
	info = LAPACKE_spptri | LAPACKE_dpptri | LAPACKE_cpptri | LAPACKE_zpptri (*matrix_layout*, *uplo*, *n*, *ap*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

uplo

indicates whether matrix ***A*** is stored in upper or lower storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

ap

is the array, referred to as AP, where:

For SPPTRI, DPPTRI, CPPTRI, and ZPPTRI:

AP contains the transformed matrix ***A*** of order *n*, resulting from the Cholesky factorization performed in a previous call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF, respectively, whose inverse is computed.

For SPPICD and DPPICD:

If *iopt* = 0, 1, 2, or 3, then AP contains the positive definite real symmetric matrix ***A***, whose inverse, condition number reciprocal, and determinant are computed, where matrix ***A*** is stored in lower-packed storage mode.

If *iopt* = 4, then AP contains the transformed matrix ***A*** of order *n*, resulting from the Cholesky factorization performed in a previous call to SPPF or DPPF, respectively, whose inverse is computed.

Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 144 on page 569.

a

has the following meaning, where:

For SPOTRI, DPOTRI, CPOTRI, and ZPOTRI:

It is the transformed matrix ***A*** of order *n*, containing results of the factorization from a previous call to SPOTRF, DPOTRF, CPOTRF, or ZPOTRF, respectively, whose inverse is computed.

For SPOICD and DPOICD:

If *iopt* = 0, 1, 2, or 3, it is the positive definite real symmetric matrix ***A***, whose inverse, condition number reciprocal, and determinant are computed, where matrix ***A*** is stored in upper or lower storage mode.

If *iopt* = 4, it is the transformed matrix ***A*** of order *n*, containing results of the factorization from a previous call to SPOF/SPOFCD or DPOF/DPOFCD, respectively, whose inverse is computed.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 144 on page 569.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

n

is the order *n* of matrix ***A***.

Specified as: an integer; *n* ≥ 0.

iopt

indicates the type of computation to be performed, where:

If *iopt* = 0, the inverse is computed for matrix ***A***.

If *iopt* = 1, the inverse and the reciprocal of the condition number are computed for matrix ***A***.

If *iopt* = 2, the inverse and the determinant are computed for matrix ***A***.

If *iopt* = 3, the inverse, the reciprocal of the condition number, and the determinant are computed for matrix ***A***.

If *iopt* = 4, the inverse is computed for the Cholesky factored matrix ***A***.

Specified as: an integer; *iopt* = 0, 1, 2, 3, or 4.

rcond

See On Return.

det

See On Return.

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*. Specified as: an area of storage, containing numbers of the data type indicated in Table 144 on page 569.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SPOICD, DPOICD, SPPICD, and DPPICD dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux* ≥ *n*.

info

See On Return.

On Return

ap

is an array, referred to as AP, in which the transformed matrix ***A*** of order *n*, containing the inverse of the matrix, is stored.

For SPPTRI, DPPTRI, CPPTRI, and ZPPTRI:
The transformed matrix is stored in upper- or lower-packed storage mode.

For SPPICD and DPPICD:
The transformed matrix is stored in lower-packed storage mode.

Returned as: a one-dimensional array of at least length $n(n+1)/2$, containing numbers of the data type indicated in Table 144 on page 569.

a

is the transformed matrix ***A*** of order *n*, containing the inverse of the matrix in upper or lower storage mode. Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 144 on page 569.

rcond

is the reciprocal of the condition number, *rcond*, of matrix ***A***. Returned as: a real number of the data type indicated in Table 144 on page 569; *rcond* ≥ 0.

det

is the vector ***det***, containing the two components det_1 and det_2 of the determinant of matrix ***A***. The determinant is:

$$det_1\left(10^{det_2}\right)$$

where $1 \le det_1 < 10$. Returned as: an array of length 2, containing numbers of the data type indicated in Table 144 on page 569.

info

has the following meaning:

If *info* = 0, the inverse completed successfully.

If *info* > 0, *info* is set equal to the first *i* where $\boldsymbol{A}_{ii}$ is zero; the matrix is not positive definite, and its inverse could not be completed.

Returned as:

- For SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPPTRI, DPPTRI, CPPTRI, ZPPTRI, SPPICD, and DPPICD, returned as: an integer; info ≥ 0.
- For LAPACKE_LAPACKE_spotri, LAPACKE_dpotri, LAPACKE_cpotri, LAPACKE_zpotri, LAPACKE_spptri, LAPACKE_dpptri, LAPACKE_cpptri, and LAPACKE_zppti, returned as an integer function value; info ≥ 0.

Specified as: an integer; *info* ≥ 0.

Notes:

1. In your C program, the arguments *info* and *rcond* must be passed by reference.
2. For SPOICD, DPOICD, SPPICD, and DPPICD, when you specify *iopt* = 4, you must do the following:
   - For SPOICD and DPOICD, specify the same storage mode for matrix ***A*** that was specified in the previous call to SPOF/SPOFCD or DPOF/DPOFCD, respectively.
   - For SPPICD and DPPICD, use Cholesky factorization in the previous call to SPPF or DPPF, respectively.
3. The scalar data specified for input arguments *uplo*, *lda*, and *n* for these subroutines must be the same as the input arguments specified for the corresponding factorization subroutines.
4. All subroutines accept lowercase letters for the *uplo* argument.
5. SPPICD and DPPICD in some cases utilize algorithms based on recursive packed storage format. (See references [“63” on page 1256], [“79” on page 1257], and [“81” on page 1257]).
6. The way _POTRI and _PPTRI subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the *info* argument to provide information about the computational error, but they also provide an error message.
7. On both input and output, matrix ***A*** conforms to LAPACK format.
8. For a description of how a positive definite symmetric matrix is stored in upper- or lower-packed storage mode in an array or in upper or lower storage mode, see “Symmetric Matrix” on page 80.
9. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 49.

Function

These subroutines find the inverse of positive definite matrix ***A***, where:

- $\boldsymbol{A}^{-1}$ is the inverse of matrix ***A***, where $\boldsymbol{AA}^{-1} = \boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}$.
- For positive definite real symmetric matrix ***A***:

  $\boldsymbol{A} = \boldsymbol{LL}^{\mathrm{T}}$ or $\boldsymbol{U}^{\mathrm{T}}\boldsymbol{U}$

  $\boldsymbol{A}^{-1} = \boldsymbol{L}^{-\mathrm{T}}\boldsymbol{L}^{-1}$ or $\boldsymbol{U}^{-1}\boldsymbol{U}^{-\mathrm{T}}$
- For positive definite complex Hermitian matrix ***A***:

  $\boldsymbol{A} = \boldsymbol{LL}^{\mathrm{H}}$ or $\boldsymbol{U}^{\mathrm{H}}\boldsymbol{U}$

  $\boldsymbol{A}^{-1} = \boldsymbol{L}^{-\mathrm{H}}\boldsymbol{L}^{-1}$ or $\boldsymbol{U}^{-1}\boldsymbol{U}^{-\mathrm{H}}$

Note: SPPICD and DPPICD only support a matrix in lower-packed storage mode.

Additionally, the subroutines SPOICD, DPOICD, SPPICD, and DPPICD find the reciprocal of the condition number and the determinant of positive definite symmetric matrix ***A*** using Cholesky factorization, where:

- $1/(\|\boldsymbol{A}\|_1)(\|\boldsymbol{A}^{-1}\|_1)$ is the reciprocal of the condition number, where $\|\boldsymbol{A}\|_1$ is the one-norm of matrix ***A***.
- $|\boldsymbol{A}|$ is the determinant of matrix ***A***, where $|\boldsymbol{A}|$ is expressed as:

$$det_1\left(10^{det_2}\right)$$

- The *iopt* argument is used to determine the combination of output items produced: the inverse, the reciprocal of the condition number, and the determinant.

If *n* is 0, no computation is performed. See references [“46” on page 1255], [“48” on page 1255], and [“54” on page 1256].

Error conditions

Resource Errors

- Unable to allocate internal work area.
- Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors

Note: If the Cholesky factorization performed by one of the _POTRF and _PPTRF subroutines failed because matrix ***A*** was not positive definite, the results returned by the corresponding _POTRI or _PPTRI subroutine are unpredictable.

If the Cholesky factorization performed by SPPF, DPPF, SPOF/SPOFCD, or DPOF/DPOFCD failed because matrix ***A*** was not positive definite, the results returned by SPOICD, DPOICD, SPPICD, or DPPICD, respectively, with *iopt* = 4, are unpredictable.

For SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPPTRI, DPPTRI, CPPTRI, and ZPPTRI:

The inverse of matrix ***A*** could not be computed.

- One or more of the diagonal elements of the factored matrix ***A*** are zero. The first diagonal element that is found to be exactly zero is identified in the computational error message and returned in *info*. If one or more of the diagonal elements of the factored matrix ***A*** are negative, the results are unpredictable.
- The computational error message may occur multiple times with processing continuing after each error because the default for the number of allowable errors for error code 2151 is set to be unlimited in the ESSL error option table.

For SPOICD, DPOICD, SPPICD, and DPPICD:

Matrix ***A*** is not positive definite.

- These subroutines do not perform the inverse, determinant, and reciprocal of the condition number computations.
- For *iopt* = 1, 2, or 3, the leading minor of order *i* has a nonpositive determinant. The order *i* is identified in the computational error message.
- *i* can be determined at run time by using the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2115 in the error option table; otherwise, the default value causes your program to terminate. If your program is not terminated, the return code is set to 2. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

The inverse of matrix ***A*** could not be computed.

- For *iopt* = 4, for _POICD and _PPICD, one or more of the diagonal elements of the factored matrix ***A*** are zero. *i* is the first diagonal element that is found to be exactly zero and is identified in the computational error message. If one or more of the diagonal elements of the factored matrix ***A*** are negative, the results are unpredictable.
- *i* can be determined at run time by using the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2150 in the error option table; otherwise, the default value causes your program to terminate. If your program is not terminated, the return code is set to 3. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. $n < 0$
3. $lda \le 0$
4. $lda < n$
5. *iopt* ≠ 0, 1, 2, 3, or 4
6. Error 2015 is recoverable or *naux* ≠ 0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example uses SPOTRI to compute the inverse of matrix ***A***, where *iopt* = 4, and matrix ***A*** is the transformed matrix factored by SPOTRF in Example 9.

Call Statement and Input:

```
             UPLO  N   A   LDA  INFO
              |    |   |    |    |
CALL SPOTRI( 'U' , 9 , A ,  9 , INFO )
```

A =(same as output A in Example 2 for SPOTRF)

Output:

```
        | 2.0  -1.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0 |
        |  .    2.0  -1.0   0.0   0.0   0.0   0.0   0.0   0.0 |
        |  .     .    2.0  -1.0   0.0   0.0   0.0   0.0   0.0 |
        |  .     .     .    2.0  -1.0   0.0   0.0   0.0   0.0 |
A    =  |  .     .     .     .    2.0  -1.0   0.0   0.0   0.0 |
        |  .     .     .     .     .    2.0  -1.0   0.0   0.0 |
        |  .     .     .     .     .     .    2.0  -1.0   0.0 |
        |  .     .     .     .     .     .     .    2.0  -1.0 |
        |  .     .     .     .     .     .     .     .    1.0 |
```

```
INFO  =   0
```

Example 2

This example uses CPOTRI to compute the inverse of the matrix ***A***, stored in lower storage mode. Matrix ***A*** is the transformed matrix factored by CPOTRF in Example 3.

Call Statement and Input:

```
             UPLO  N   A   LDA  INFO
              |    |   |    |    |
CALL CPOTRI( 'L' , 3 , A ,  3 , INFO )
```

A = (same as output A in Example 3 for CPOTRF)

Output:

```
        | ( .05,  .00)      .          .       |
A    =  | ( .00, -.01) (.02, .00)      .       |
        | (-.01,  .00) (.00, .00) (.02, .00) |
```

```
INFO  =   0
```

Example 3

This example uses CPOTRI to compute the inverse of the matrix ***A***, stored in upper storage mode. Matrix ***A*** is the transformed matrix factored by CPOTRF in Example 4.

Call Statement and Input:

```
             UPLO  N   A   LDA  INFO
              |    |   |    |    |
CALL CPOTRI( 'U' , 3 , A ,  3 , INFO )
```

A = (same as output A in Example 4 for CPOTRF)

Output:

```
        | ( .13,  .00) (-.02, -.03) ( .00, .01) |
A    =  |      .       ( .07,  .00) (-.01, .01) |
        |      .            .       ( .03, .00) |
```

```
INFO  =   0
```

Example 4

This example uses SPOICD to compute the inverse, reciprocal of the condition number, and determinant of matrix ***A***, stored in upper storage mode. Matrix ***A*** is:

```
              | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
              | 1.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0 |
              | 1.0  2.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0 |
              | 1.0  2.0  3.0  4.0  4.0  4.0  4.0  4.0  4.0 |
              | 1.0  2.0  3.0  4.0  5.0  5.0  5.0  5.0  5.0 |
              | 1.0  2.0  3.0  4.0  5.0  6.0  6.0  6.0  6.0 |
              | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  7.0  7.0 |
              | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  8.0 |
              | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 |
```

The values used to compute the reciprocal of the condition number in this example are obtained with the following values:

$\|\boldsymbol{A}\|_1$ = max(9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0) = 45.0
$\|\boldsymbol{A}^{-1}\|_1$ = 4.0

On output, the value in ***det***, |***A***|, is equal to 1, and RCOND = 1/180.

Call Statement and Input:

```
             UPLO   A  LDA   N  IOPT RCOND    DET   AUX  NAUX
              |     |   |    |   |     |       |     |    |
CALL SPOICD( 'U' , A , 9 ,  9 , 3 , RCOND , DET , AUX , 9  )


        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        |  .   2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0 |
        |  .    .   3.0  3.0  3.0  3.0  3.0  3.0  3.0 |
        |  .    .    .   4.0  4.0  4.0  4.0  4.0  4.0 |
A    =  |  .    .    .    .   5.0  5.0  5.0  5.0  5.0 |
        |  .    .    .    .    .   6.0  6.0  6.0  6.0 |
        |  .    .    .    .    .    .   7.0  7.0  7.0 |
        |  .    .    .    .    .    .    .   8.0  8.0 |
        |  .    .    .    .    .    .    .    .   9.0 |
```

Output:

```
        | 2.0  -1.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0 |
        |  .    2.0  -1.0   0.0   0.0   0.0   0.0   0.0   0.0 |
        |  .     .    2.0  -1.0   0.0   0.0   0.0   0.0   0.0 |
        |  .     .     .    2.0  -1.0   0.0   0.0   0.0   0.0 |
A    =  |  .     .     .     .    2.0  -1.0   0.0   0.0   0.0 |
        |  .     .     .     .     .    2.0  -1.0   0.0   0.0 |
        |  .     .     .     .     .     .    2.0  -1.0   0.0 |
        |  .     .     .     .     .     .     .    2.0  -1.0 |
        |  .     .     .     .     .     .     .     .    1.0 |


RCOND    =  0.005555556
DET      =  (1.0, 0.0)
```

Example 5

This example uses SPPTRI to compute the inverse of matrix ***A***, where matrix ***A*** is the transformed matrix factored in Example 5 by SPPTRF.

Note: The AP arrays are formatted in a triangular arrangement for readability; however, they are stored in lower-packed storage mode.

Call Statement and Input:

```
             UPLO  N   AP   INFO
              |    |   |     |
CALL SPPTRI( 'L' , 9 , AP , INFO )
```

AP =(same as output AP in Example 5 for SPPTRF)

Output:

```
AP   =  (2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
         2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
         2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
         2.0, -1.0, 0.0, 0.0, 0.0, 0.0,
         2.0, -1.0, 0.0, 0.0, 0.0,
         2.0, -1.0, 0.0, 0.0,
         2.0, -1.0, 0.0,
         2.0, -1.0,
         1.0)

INFO =  0
```

Example 6

This example uses SPPTRI to compute the inverse of matrix ***A***, where matrix ***A*** is the transformed matrix factored in Example 6 by SPPTRF.

Note: The AP arrays are formatted in a triangular arrangement for readability; however, they are stored in upper-packed storage mode.

Call Statement and Input:

```
             UPLO  N   AP   INFO
              |    |   |     |
CALL SPPTRI( 'U' , 9 , AP , INFO )
```

AP =(same as output AP in Example 6 for SPPTRF)

Output:

```
AP   =  (                                          2.0,
                                             -1.0, 2.0,
                                        0.0, -1.0, 2.0,
                                   0.0, 0.0, -1.0, 2.0,
                              0.0, 0.0, 0.0, -1.0, 2.0,
                         0.0, 0.0, 0.0, 0.0, -1.0, 2.0,
                    0.0, 0.0, 0.0, 0.0, 0.0, -1.0, 2.0,
               0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0, 2.0,
          0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0, 1.0)

INFO  =  0
```

Example 7

This example uses ZPPTRI to compute the inverse of matrix ***A***, where matrix ***A***, stored in lower-packed storage mode, is the transformed matrix factored in Example 7 by ZPPTRF.

Call Statement and Input:

```
             UPLO  N   AP   INFO
              |    |   |     |
CALL ZPPTRI( 'L' , 3 , AP , INFO )
```

AP =(same as output AP in Example 7 for ZPPTRF)

Output:

```
AP   =  ((0.05, 0.00) (0.00, -0.01) (-0.01, 0.00) (0.02, 0.00) (0.00, 0.00) (0.02, 0.00))

INFO =  0
```

Example 8

This example uses ZPPTRI to compute the inverse of matrix ***A***, where matrix ***A***, stored in upper-packed storage mode, is the transformed matrix factored in Example 8 by ZPPTRF.

Call Statement and Input:

```
             UPLO  N   AP   INFO
              |    |   |     |
CALL ZPPTRI( 'U' , 3 , AP , INFO )
```

AP =(same as output AP in Example 8 for ZPPTRF)

Output:

```
AP   =  ((0.13, 0.0) (-0.02, -0.03) (0.07, 0.00) (0.00, 0.01) (-0.01, 0.01) (0.03, 0.00))

INFO  =  0
```

Example 9

This example uses SPPICD to compute the inverse, reciprocal of the condition number, and determinant of the same matrix ***A*** used in Example 4; however, matrix ***A*** is stored in lower-packed storage mode in this example.

The values used to compute the reciprocal of the condition number in this example are obtained with the following values:

$\|\boldsymbol{A}\|_1$ = max(9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0) = 45.0
$\|\boldsymbol{A}^{-1}\|_1$ = 4.0

On output, the value in ***det***, |***A***|, is equal to 1, and RCOND = 1/180.

Note: The AP arrays are formatted in a triangular arrangement for readability; however, they are stored in lower-packed storage mode.

Call Statement and Input:

```
             AP   N  IOPT RCOND   DET   AUX  NAUX
             |    |   |     |      |     |    |
CALL SPPICD( AP , 9 , 3 , RCOND , DET , AUX , 9  )

AP   =   (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
          2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0,
          3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0,
          4.0, 4.0, 4.0, 4.0, 4.0, 4.0,
          5.0, 5.0, 5.0, 5.0, 5.0,
          6.0, 6.0, 6.0, 6.0,
          7.0, 7.0, 7.0,
          8.0, 8.0,
          9.0)
```

Output:

```
AP   =   (2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0,
          2.0, -1.0, 0.0,
          2.0, -1.0,
          1.0)

RCOND    =  0.005556
DET      =  (1.0, 0.0)
```

Example 10

This example uses SPPICD to compute the inverse of matrix ***A***, where *iopt* = 4, and matrix ***A*** is the transformed matrix factored in Example 10 by SPPF.

Note: The AP arrays are formatted in a triangular arrangement for readability; however, they are stored in lower-packed storage mode.

Call Statement and Input:

```
             AP    N  IOPT RCOND    DET    AUX NAUX
             |     |   |     |       |      |    |
CALL SPPICD(AP , 9 , 4 , RCOND , DET , AUX , 9)
```

AP =(same as output AP in Example 10 for SPPF)

Output:

```
AP    =   (2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           2.0, -1.0, 0.0, 0.0, 0.0, 0.0,
           2.0, -1.0, 0.0, 0.0, 0.0,
           2.0, -1.0, 0.0, 0.0,
           2.0, -1.0, 0.0,
           2.0, -1.0,
           1.0)
```

SLANSY, DLANSY, CLANSY, ZLANSY, CLANHE, ZLANHE, SLANSP, DLANSP, CLANHP, and ZLANHP (Real Symmetric or Complex Hermitian Matrix Norm)

Purpose

These subprograms compute the norm of matrix ***A*** as explained below:

SLANSY, DLANSY, >|CLANSY, ZLANSY,|< CLANHE, and ZLANHE

These subprograms compute the norm of matrix ***A***, stored in upper or lower storage mode, where:

- For SLANSY and DLANSY, ***A*** is a real symmetric matrix.
- >|For CLANSY and ZLANSY, ***A*** is a complex symmetric matrix.|<
- For CLANHE, ZLANHE, ***A*** is a complex Hermitian matrix.

SLANSP, DLANSP, CLANHP, and ZLANHP

These subroutines compute the norm of matrix ***A***, stored in upper-packed or lower-packed storage mode, where:

- For SLANSP and DLANSP, ***A*** is a real symmetric matrix.
- For CLANHP and ZLANHP, ***A*** is a complex Hermitian matrix.

Table 146: Data Types.

A	***work*, Result**	**Subprogram**
Short-precision real	Short-precision real	SLANSY$^{\Delta}$, SLANSP$^{\Delta}$
Long-precision real	Long-precision real	DLANSY$^{\Delta}$, DLANSP$^{\Delta}$
Short-precision complex	Short-precision real	CLANHE$^{\Delta}$, CLANHP$^{\Delta}$, >|CLANSY$^{\Delta}$|<
Long-precision complex	Long-precision real	ZLANHE$^{\Delta}$, ZLANHP$^{\Delta}$, >|ZLANSY$^{\Delta}$|<
$^{\Delta}$ LAPACK		

Syntax

Language	Syntax
Fortran	SLANSY | DLANSY | >|CLANSY | ZLANSY ||< CLANHE | ZLANHE (*norm*, *uplo*, *n*, *a*, *lda*, *work*) SLANSP | DLANSP | CLANHP | ZLANHP (*norm*, *uplo*, *n*, *ap*, *work*)

C and C++	slansy | dlansy | >|clansy | zlansy ||< clanhe | zlanhe (*norm*, *uplo*, *n*, *a*, *lda*, *work*);
	slansp | dlansp | clanhp | zlanhp (*norm*, *uplo*, *n*, *ap*, *work*);

LAPACKE	LAPACKE_slansy | LAPACKE_dlansy | >|LAPACKE_clansy | LAPACKE_zlansy ||< LAPACKE_clanhe | LAPACKE_zlanhe (*matrix_layout*, *norm*, *uplo*, *n*, *a*, *lda*);
	LAPACKE_slansp | LAPACKE_dlansp | LAPACKE_clanhp | LAPACKE_zlanhp (*matrix_layout*, *norm*, *uplo*, *n*, *ap*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

norm

specifies the type of computation, where:

If *norm* = 'O' or '1', the one norm of ***A*** is computed.

If *norm* = 'I', the infinity norm of ***A*** is computed.

If *norm* = 'F' or 'E', the Frobenius or Euclidean norm of ***A*** is computed.

If *norm* = 'M', the absolute value of the matrix element having the largest absolute value, i.e., max (|***A***|), is returned.

Specified as: a single character; *norm* = 'O', '1', 'I', 'F', 'E', or 'M'.

uplo

indicates whether matrix ***A*** is stored in upper or lower storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

n

is the order of matrix ***A***.

Specified as: an integer; $n \geq 0$.

ap

is the matrix ***A***, stored in upper-packed or lower-packed storage mode.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 146 on page 579.

a

is the matrix ***A***, stored in upper or lower storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 146 on page 579.

lda
is the leading dimension of matrix ***A***.

Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

work
is the work area used by this subroutine, where:

- When *norm* = 'O', '1', or 'I', the size the of *work* argument is (at least) of length *n*.
- Otherwise, *work* is not referenced.

Specified as: an area of storage containing numbers of data type indicated in Table 146 on page 579.

On Return

Function value
is the result of the norm computation, returned as a number of the data type indicated in Table 146 on page 579.

If *norm* = 'O' or '1', the one norm of ***A*** is returned.

If *norm* = 'I', the infinity norm of ***A*** is returned.

If *norm* = 'F' or 'E', the Frobenius or Euclidean norm of ***A*** is returned.

If *norm* = 'M', the absolute value of the matrix element having the largest absolute value, i.e., max (|***A***|), is returned.

If *n* = 0, the function returns zero.

Notes

1. Declare this function in your program as returning a value of the data type indicated in Table 146 on page 579.
2. This function accepts lowercase letters for the *norm* and *uplo* arguments.
3. For real symmetric and complex Hermitian matrices, the one norm and the infinity norm are identical.

Function

One of the following computations is performed on matrix ***A***, depending on the value specified for the *norm* argument:

Value specified for *norm*	Type of computation performed
'O' or '1'	One norm
'I'	Infinity norm
'F' or 'E'	Frobenius or Euclidean norm
'M'	Absolute value of the matrix element having the largest absolute value, that is, max (|***A***|)

If *n* = 0, the function returns zero.

Error conditions

Resource Errors
None.

Computational Errors
None.

Input-Argument Errors

1. *norm* ≠ 'O', '1', 'I', 'F', 'E', or 'M'
2. *uplo* ≠ 'U' or 'L'
3. *n* < 0
4. *n* > *lda*
5. *lda* ≤ 0

Examples

Example 1

This example computes the one norm of real symmetric matrix ***A*** stored in lower storage mode.

Call Statements and Input:

```
                 NORM  UPLO  N   A  LDA  WORK
                   |     |   |   |   |     |
ANORM = DLANSY(  '1',  'L', 9 , A , 9 , WORK )
```

A = (same as input matrix A in Example 1)

```

```

Output:

ANORM = 45.0

Example 2

This example computes the one norm of real symmetric matrix ***A*** stored in upper storage mode.

Call Statements and Input:

```
                 NORM  UPLO  N   A  LDA  WORK
                   |     |   |   |   |     |
ANORM = DLANSY(  '1',  'U', 9 , A , 9 , WORK )
```

A = (same as input matrix A in Example 2)

```

```

Output:

ANORM = 45.0

Example 3

This example computes the one norm of complex symmetric matrix ***A*** stored in upper storage mode.

Call Statements and Input:

```
                 NORM  UPLO  N   A  LDA  WORK
                   |     |   |   |   |     |
ANORM = ZLANSY(  '1',  'U', 3 , A , 3 , WORK )
```

```
       ┌                                          ┐
       | ( 9.0,   .0)  ( 3.0,  3.0)  ( 3.0, -3.0) |
  A =  |       .       (18.0,   .0)  ( 8.0, -6.0) |
       |       .              .      (43.0,   .0) |
       └                                          ┘
```

Output:

ANORM = 57.2426406871192839

Example 4

This example computes the one norm of complex Hermitian matrix ***A*** stored in lower storage mode.

Call Statements and Input:

```
                 NORM  UPLO  N   A  LDA  WORK
                   |     |   |   |   |     |
ANORM = ZLANHE(  '1',  'L', 3 , A , 3 , WORK )
```

A = (same as input matrix A in Example 3)

Output:

ANORM = 89.39

Example 5

This example computes the one norm of complex Hermitian matrix ***A*** stored in upper storage mode.

Call Statements and Input:

```
                 NORM  UPLO  N   A  LDA  WORK
                   |     |   |   |   |     |
ANORM = ZLANHE(  '1',  'U', 3 , A , 3 , WORK )
```

A = (same as input matrix A in Example 4)

Output:

ANORM = 57.24

Example 6

This example computes the one norm of real symmetric matrix ***A*** stored in lower-packed storage mode.

Call Statements and Input:

```
                 NORM  UPLO  N   AP   WORK
                   |     |   |    |     |
ANORM = DLANSP(  '1',  'L', 9 , AP , WORK )
```

AP = (same as input matrix AP in Example 5)

Output:

ANORM = 45.0

Example 7

This example computes the one norm of real symmetric matrix ***A*** stored in upper-packed storage mode.

Call Statements and Input:

```
                 NORM  UPLO  N   AP   WORK
                   |     |   |    |     |
ANORM = DLANSP(  '1',  'U', 9 , AP , WORK )
```

AP = (same as input matrix AP in Example 6)

Output:

ANORM = 45.0

Example 8

This example computes the one norm of complex Hermitian matrix ***A*** stored in lower-packed storage mode.

Call Statements and Input:

```
                 NORM  UPLO   N   AP   WORK
                   |     |    |    |     |
ANORM = ZLANHP(  '1',  'L' , 3 , AP , WORK )
```

AP = (same as input matrix AP in Example 7)

Output:

ANORM = 89.39

Example 9

This example computes the one norm of complex Hermitian matrix ***A*** stored in upper-packed storage mode.

Call Statements and Input:

```
                 NORM  UPLO   N   AP   WORK
                   |     |    |    |     |
ANORM = ZLANHP(  '1',  'U' , 3 , AP , WORK )
```

AP = (same as input matrix AP in Example 8)

Output:

ANORM = 57.24

SSYSV, DSYSV, CSYSV, ZSYSV, CHESV, ZHESV, SSPSV, DSPSV, CSPSV, ZSPSV, CHPSV, and ZHPSV (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system ***AX*** = ***B*** for ***X***, where ***A*** is an indefinite real or complex symmetric or complex Hermitian matrix and ***X*** and ***B*** are general matrices.

Table 147: Data Types.

A*, *B*, *work	**Subroutine**
Short-precision real	SSYSV$^{\Delta}$, SSPSV$^{\Delta}$
Long-precision real	DSYSV$^{\Delta}$, DSPSV$^{\Delta}$
Short-precision complex	CSYSV$^{\Delta}$, CHESV$^{\Delta}$, CSPSV$^{\Delta}$, CHPSV$^{\Delta}$
Long-precision complex	ZSYSV$^{\Delta}$, ZHESV$^{\Delta}$, ZSPSV$^{\Delta}$, ZHPSV$^{\Delta}$
$^{\Delta}$LAPACK	

Syntax

Language	Syntax
Fortran	CALL SSYSV | DSYSV | CSYSV | ZSYSV | CHESV | ZHESV (*uplo*, *n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*, *work*, *lwork*, *info*) CALL SSPSV | DSPSV | CSPSV | ZSPSV | CHPSV | ZHPSV (*uplo*, *n*, *nrhs*, *ap*, *ipiv*, *b*, *ldb*, *info*)

C and C++	ssysv | dsysv | csysv | zsysv | chesv | zhesv (*uplo*, *n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*, *work*, *lwork*, *info*);
	sspsv | dspsv | cspsv | zspsv | chpsv | zhpsv (*uplo*, *n*, *nrhs*, *ap*, *ipiv*, *b*, *ldb*, *info*);

LAPACKE	info = LAPACKE_ssysv | LAPACKE_dsysv | LAPACKE_csysv | LAPACKE_zsysv | LAPACKE_chesv | LAPACKE_zhesv (*matrix_layout*, *uplo*, *n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*);
	info = LAPACKE_sspsv | LAPACKE_dspsv | LAPACKE_cspsv | LAPACKE_zspsv | LAPACKE_chpsv | LAPACKE_zhpsv (*matrix_layout*, *uplo*, *n*, *nrhs*, *ap*, *ipiv*, *b*, *ldb*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

uplo

indicates whether the upper or lower triangular part of the matrix ***A*** is referenced, where:

If *uplo* = 'U', the upper triangular part is referenced.

If *uplo* = 'L', the lower triangular part is referenced.

Specified as: a single character; *uplo* = 'U' or 'L'.

n

is the order of matrix ***A*** and the number of rows in matrix ***B***.

Specified as: an integer; $n \geq 0$.

nrhs

is the number of right-hand sides; that is, the number of columns in matrix ***B***.

Specified as: an integer; $nrhs \geq 0$.

a

is the indefinite real symmetric, complex symmetric, or complex Hermitian matrix ***A*** of order *n*.

If *uplo* = 'U', it is stored in upper storage mode.

If *uplo* = 'L', it is stored in lower storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 147 on page 584.

ap

is the indefinite real symmetric, complex symmetric, or complex Hermitian matrix ***A*** of order *n*. It is stored in an array, referred to as AP, where:

If *uplo* = 'U', it is stored in upper-packed storage mode.

If *uplo* = 'L', it is stored in lower-packed storage mode.

Specified as: one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 147 on page 584.

lda

is the leading dimension of the array specified for ***A***.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

ipiv

See On Return.

b

is the general matrix ***B*** containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of matrix ***B***.

Specified as: an *ldb* by *nrhs* array, containing numbers of the data type indicated in Table 147 on page 584.

ldb

is the leading dimension of the array specified for ***B***.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

work

is a work area used by these subroutines, where:

If $lwork = 0$, *work* is ignored.

If $lwork \neq 0$, the size of *work* is determined as follows:

- If $lwork \neq -1$, *work* is (at least) of length *lwork*.
- If $lwork = -1$, *work* is (at least) of length 1.

Specified as: an area of storage containing numbers of the data type indicated in Table 147 on page 584.

lwork

is the number of elements in array `WORK`.

Specified as: an integer; where:

- If $lwork = 0$, the subroutine dynamically allocates the workspace needed for use during this computation. The work area is deallocated before control is returned to the calling program.
- If $lwork = -1$, subroutine performs a workspace query and returns the optimal required size of *work* in $work_1$. No computation is performed and the subroutine returns after error checking is complete.
- Otherwise, $lwork \geq 1$. It is suggested that the user specify $lwork \geq 8n$.

info

See On Return.

On Return

a

is the transformed matrix ***A*** containing the results of the factorization. See "Function" on page 596.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 147 on page 584.

ap

is the transformed matrix ***A*** containing the results of the factorization. See "Function" on page 596.

Returned as: a one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 147 on page 584.

ipiv

If $info = 0$, *ipiv* contains the pivot indices.

If $ipiv_k > 0$, then rows and columns k and $ipiv_k$ were interchanged and $\boldsymbol{D}_{k,k}$ is a 1 × 1 diagonal block.

If *uplo* = 'U' and $ipiv_k = ipiv_{k-1} < 0$, then rows and columns k-1 and $-ipiv_k$ were interchanged and $\boldsymbol{D}_{k-1:k,k-1:k}$ is a 2 × 2 diagonal block.

If *uplo* = 'L' and $ipiv_k = ipiv_{k+1} < 0$, then rows and columns k+1 and $-ipiv_k$ were interchanged and $\boldsymbol{D}_{k:k+1,k:k+1}$ is a 2 × 2 diagonal block.

Returned as: a one-dimensional integer array of (at least) length *n*, containing integers.

b

If *info* = 0, *b* is the matrix ***X***, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of ***X***.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 147 on page 584.

work

is a work area used by this subroutine if *lwork* ≠ 0, where:

If *lwork* ≠ 0 and *lwork* ≠ -1, its size is (at least) of length *lwork*.

If *lwork* = -1, its size is (at least) of length 1.

Returned as: an area of storage, where:

If *lwork* ≥ 1 or *lwork* = -1, then $work_1$ is set to the optimal *lwork* value and all other elements of *work* are overwritten.

info

has the following meaning:

- If *info* = 0, the factorization completed successfully.
- If *info* > 0, the factorization was unsuccessful and *info* is set to *i* where d_{ii} is exactly zero.

Returned as:

- For SSYSV, DSYSV, CSYSV, ZSYSV, CHESV, ZHESV, SSPSV, DSPSV, CSPSV, ZSPSV, CHPSV, and ZHPSV, returned as: an integer; info ≥ 0.
- For LAPACKE_ssysv, LAPACKE_dsysv, LAPACKE_csysv, LAPACKE_zsysv, LAPACKE_chesv, LAPACKE_zhesv, LAPACKE_sspsv, LAPACKE_dspsv, LAPACKE_cspsv, LAPACKE_zspsv, LAPACKE_chpsv, and LAPACKE_zhpsv, returned as an integer function value; info ≥ 0.

Notes

1. These subroutines accept lowercase letters for the *uplo* argument.
2. In your C program, argument *info* must be passed by reference.
3. *a*, *ap*, *b*, *ipiv*, and *work* must have no common elements; otherwise, results are unpredictable.
4. For a description of how real and complex symmetric matrices are stored in lower or upper storage mode, see "Lower Storage Mode" on page 83 or "Upper Storage Mode" on page 83, respectively.

   For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see "Complex Hermitian Matrix" on page 84.
5. For a description of how real and complex symmetric matrices are stored in lower- or upper-packed storage mode, see "Lower-Packed Storage Mode" on page 80 or "Upper-Packed Storage Mode" on page 82, respectively.

   For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see "Complex Hermitian Matrix" on page 84.
6. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.
7. For best performance, specify *lwork* = 0.

Function

These subroutines solve the system ***AX*** = ***B*** for ***X***, where ***A*** is an indefinite real or complex symmetric or complex Hermitian indefinite matrix and ***X*** and ***B*** are general matrices.

For SSYSV, DSYSV, CSYSV, ZSYSV, SSPSV, DSPSV, CSPSV, and ZSPSV:

The indefinite real or complex symmetric indefinite matrix $\boldsymbol{A}$ is factored using the Bunch-Kaufman diagonal pivoting method, where $\boldsymbol{A}$ is expressed as one of the following:

$\boldsymbol{A} = \boldsymbol{UDU}^T$

$\boldsymbol{A} = \boldsymbol{LDL}^T$

where:

$\boldsymbol{U}$ is a product of permutation and unit upper triangular matrices.

$\boldsymbol{L}$ is a product of permutation and unit lower triangular matrices.

$\boldsymbol{D}$ is a symmetric block diagonal matrix, consisting of 1 × 1 and 2 × 2 diagonal blocks.

Matrix $\boldsymbol{A}$ is stored as follows:

- For SSYSV, DSYSV, CSYSV, and ZSYSV, matrix $\boldsymbol{A}$ is stored in upper or lower storage mode.
- For SSPSV, DSPSV, CSPSV, and ZSPSV, matrix $\boldsymbol{A}$ is stored in upper- or lower-packed storage mode.

For CHESV, ZHESV, CHPSV, and ZHPSV:

The indefinite complex Hermitian indefinite matrix $\boldsymbol{A}$ is factored using the Bunch-Kaufman diagonal pivoting method, where $\boldsymbol{A}$ is expressed as one of the following:

$\boldsymbol{A} = \boldsymbol{UDU}^H$

$\boldsymbol{A} = \boldsymbol{LDL}^H$

where:

$\boldsymbol{U}$ is a product of permutation and unit upper triangular matrices.

$\boldsymbol{L}$ is a product of permutation and unit lower triangular matrices.

$\boldsymbol{D}$ is a complex Hermitian block diagonal matrix, consisting of 1 × 1 and 2 × 2 diagonal blocks.

Matrix $\boldsymbol{A}$ is stored as follows:

- For CHESV and ZHESV, matrix $\boldsymbol{A}$ is stored in upper or lower storage mode.
- For CHPSV and ZHPSV, matrix $\boldsymbol{A}$ is stored in upper- or lower-packed storage mode.

If *n* is 0, no computation is performed and the subroutine returns after doing some parameter checking. If *n* > 0 and *nrhs* is 0, no solutions are computed and the subroutine returns after factoring the matrix. See references [“9” on page 1253] and [“19” on page 1254].

Error conditions

Resource Errors
lwork = 0 and unable to allocate work area

Computational Errors
Matrix $\boldsymbol{A}$ is singular.

- The factorization completed but the block diagonal matrix $\boldsymbol{D}$ is exactly singular. *info* is set to *i*, where d_{ii} is exactly zero. This diagonal element is identified in the computational error message.
- The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2147 is set to be unlimited in the ESSL error option table. For details, see “What Can You Do about ESSL Computational Errors?” on page 65.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. *n* < 0
3. *nrhs* < 0
4. *lda* ≤ 0

5. *n* > *lda*
6. *ldb* ≤ 0
7. *n* > *ldb*
8. *lwork* ≠ 0 and *lwork* ≠ -1 and *lwork* < the minimum required value

Examples

Example 1

This example shows how to solve the system ***AX*** = ***B***, for three right-hand sides, where indefinite real symmetric matrix ***A*** is the same matrix factored in the Example 1 for DSYTRF.

Note: Because *lwork* = 0, the subroutine dynamically allocates `WORK`.

Call Statement and Input:

```
            UPLO  N  NRHS  A  LDA  IPIV   B  LDB  WORK LWORK INFO
             |    |    |   |   |     |    |   |     |    |     |
CALL DSYSV( 'L' , 8 ,  3 , A , 8 , IPIV , B , 8 , WORK,  0 , INFO )
```

A = (same as input A in Example 1)
B = (same as input B in Example 1)

Output:

```
                                                   ┐
         |  3.0   .    .    .    .    .    .    .  |
         |  5.0  3.0   .    .    .    .    .    .  |
         |  1.0 -1.0  4.0   .    .    .    .    .  |
 A   =   | -1.0  1.0 -1.0  8.0   .    .    .    .  |
         |  1.0  0.0  0.0 -1.0  1.0   .    .    .  |
         |  0.0 -1.0  1.0 -1.0  3.0  1.0   .    .  |
         |  1.0 -1.0  1.0 -1.0 -1.0  1.0  2.0   .  |
         | -1.0  0.0  1.0 -1.0  1.0  0.0  1.0 16.0 |
                                                   ┘
```

```
IPIV = ( -2 -2 3 4 -6 -6 7 8 )
```

```
                         ┐
         | 1.0   1.0   8.0 |
         | 1.0   2.0   7.0 |
         | 1.0   3.0   6.0 |
 B   =   | 1.0   4.0   5.0 |
         | 1.0   5.0   4.0 |
         | 1.0   6.0   3.0 |
         | 1.0   7.0   2.0 |
         | 1.0   8.0   1.0 |
                         ┘
```

```
INFO  =   0
```

Example 2

This example shows how to solve the system ***AX*** = ***B*** for three right-hand sides, where indefinite complex symmetric matrix ***A*** is the same matrix factored in the Example 2 for ZSYTRF.

Note: Because *lwork* = 0, the subroutine dynamically allocates `WORK`.

Call Statement and Input:

```
             UPLO  N  NRHS  A  LDA  IPIV   B  LDB  WORK LWORK INFO
              |    |    |   |   |     |    |   |     |    |     |
CALL ZSYSV( 'L' , 4 ,  3 , A , 4 , IPIV , B , 4 , WORK,  0 , INFO )
```

A = (same as input A in Example 2)
B = (same as input B in Example 2)

Output:

```
         ┌                                                          ┐
         | ( 0.368,-0.319)        .               .               .        |
 A   =   | (-0.062, 0.006) ( 0.258,-0.147)        .               .        |
         | ( 0.625, 0.257) ( 1.085,-0.335) ( 0.333, 0.315)        .        |
         | (-0.462, 0.314) (-0.444, 1.248) (-0.437,-1.386) ( 0.841, 0.431) |
         └                                                          ┘
```

```
IPIV = ( 1 2 4 4 )
```

```
                                                        ┐
         | ( 0.409,-0.663) (-0.582,-1.410) ( 2.484, 2.216) |
 B   =   | (-1.664,-0.552) (-1.503,-4.837) (-3.577, 2.575) |
         | ( 2.388, 4.010) ( 1.260,-0.430) (-1.273, 0.177) |
         | ( 1.562, 0.164) ( 6.213, 1.471) (-0.980,-2.551) |
                                                        ┘
```

```
INFO  =   0
```

Example 3

This example shows how to solve the system ***AX*** = ***B*** for three right-hand sides, where indefinite complex Hermitian matrix ***A*** is the same matrix factored in the Example 3 for ZHETRF.

Notes:

1. Because *lwork* = 0, the subroutine dynamically allocates WORK.
2. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input:

```
             UPLO  N  NRHS  A  LDA  IPIV   B  LDB  WORK LWORK INFO
              |    |    |   |   |     |    |   |     |    |     |
CALL ZHESV( 'L' , 4 ,  3 , A , 4 , IPIV , B , 4 , WORK,  0 , INFO )
```

A = (same as input A in Example 3)
B = (same as input B in Example 3)

Output:

```
         ┌                                                              ┐
         | (-0.550, 0.000)        .                .                .        |
 A   =   | (-0.027, 0.476) (-0.483,  0.000)        .                .        |
         | ( 0.062, 0.244) (-0.002, -0.269) (-0.490,  0.000)        .        |
         | (-0.249, 0.022) ( 0.152, -0.091) ( 0.244, -0.002) (-0.479, 0.000) |
                                                                                   ┘
```

```
IPIV = ( 1 2 3 4 )
```

```
        ┌                                                    ┐
        | (-1.623,-4.385) ( 2.635,-1.111) (-2.436,-0.306) |
 B   =  | (-3.533,-0.212) ( 1.865,-2.830) (-0.866,-0.308) |
        | (-1.742,-1.724) (-1.576, 1.575) (-0.478,-1.172) |
        | (-1.537,-2.115) (-1.047, 1.608) ( 3.093, 0.365) |
                                                         ┘
```

```
INFO  =   0
```

Example 4

This example shows how to solve the system ***AX*** = ***B*** for three right-hand sides, where indefinite real symmetric matrix ***A*** is the same matrix factored in the Example 1 for DSYTRF.

Call Statement and Input:

```
            UPLO  N  NRHS  AP   IPIV   B  LDB  INFO
             |    |    |    |     |    |   |     |
CALL DSPSV( 'L' , 8 ,  3 , AP , IPIV , B , 8 , INFO )
```

AP = (same as input AP in Example 4)
B = (same as input B in Example 4)

Output:

```
AP = (   3.0   5.0   1.0  -1.0   1.0   0.0   1.0  -1.0,
               3.0  -1.0   1.0   0.0  -1.0  -1.0   0.0,
                     4.0  -1.0   0.0   1.0   1.0   1.0,
                           8.0  -1.0  -1.0  -1.0  -1.0,
                                 1.0   3.0  -1.0   1.0,
                                       1.0   1.0   0.0,
                                             2.0   1.0,
                                                  16.0 )
```

```
IPIV = ( 1 2 -2 -2 5 6 7 8 )
```

```
                                 ┐
        | 1.0   1.0   8.0 |
        | 1.0   2.0   7.0 |
        | 1.0   3.0   6.0 |
 B   =  | 1.0   4.0   5.0 |
        | 1.0   5.0   4.0 |
        | 1.0   6.0   3.0 |
        | 1.0   7.0   2.0 |
        | 1.0   8.0   1.0 |
                         ┘
```

```
INFO  =   0
```

Example 5

This example shows how to solve the system ***AX*** = ***B*** for three right-hand sides, where indefinite complex symmetric matrix ***A*** is the same matrix factored in the Example 2 for ZSYTRF.

Call Statement and Input:

```
            UPLO  N  NRHS  AP   IPIV   B  LDB  INFO
             |    |    |    |     |    |   |     |
CALL ZSPSV( 'L' , 4 ,  3 , AP , IPIV , B , 4 , INFO )
```

AP = (same as input AP in Example 5)
B = (same as input B in Example 5)

Output:

```
AP = ( (0.368, -0.319), (-0.062, 0.006), (0.625,  0.257), (-0.462,  0.314),
                        ( 0.258,-0.147), (1.085, -0.335), (-0.444,  1.248),
                                         (0.333,  0.315), (-0.437, -1.386),
                                                          ( 0.841,  0.431) )
```

```
IPIV = (  1 2 4 4 )
```

```
         ┌                                                 ┐
         | ( 0.409,-0.663) (-0.582,-1.410) ( 2.484, 2.216) |
 B   =   | (-1.664,-0.552) (-1.503,-4.837) (-3.577, 2.575) |
         | ( 2.388, 4.010) ( 1.260,-0.430) (-1.273, 0.177) |
         | ( 1.562, 0.164) ( 6.213, 1.471) (-0.980,-2.551) |
                                                          ┘
```

```
INFO  =   0
```

Example 6

This example shows how to solve the system ***AX*** = ***B*** for three right-hand sides, where indefinite complex Hermitian matrix ***A*** is the same matrix factored in the Example 3 for ZHETRF.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input:

```
            UPLO  N  NRHS  AP   IPIV   B  LDB  INFO
             |    |    |    |     |    |   |     |
CALL ZHPSV( 'L' , 4 ,  3 , AP , IPIV , B , 4 , INFO )
```

AP = (same as input AP in Example 6)
B = (same as input B in Example 6)

Output:

```
 AP = ( (-0.550, 0.000), (-0.027, 0.476), ( 0.062,  0.244), (-0.249,  0.022),
                         (-0.484, 0.000), (-0.002, -0.269), ( 0.152, -0.091),
                                          (-0.490,  0.000), ( 0.245, -0.002),
                                                            (-0.479,  0.000) )
```

```
IPIV = (  1 2 3 4 )
```

```
         ┌                                                 ┐
         | (-1.623,-4.385) ( 2.635,-1.111) (-2.436,-0.306) |
 B   =   | (-3.533,-0.212) ( 1.865,-2.830) (-0.866,-0.308) |
         | (-1.742,-1.724) (-1.576, 1.575) (-0.478,-1.172) |
         | (-1.537,-2.115) (-1.047, 1.608) ( 3.093, 0.365) |
                                                          ┘
```

```
INFO  =   0
```

SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, and ZHPTRF (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization)

Purpose

These subroutines factor an indefinite real or complex symmetric or complex Hermitian matrix ***A***. The matrix ***A*** is factored using the Bunch-Kaufman diagonal pivoting method.

To solve the system of equations with one or more right-hand sides, follow the call to SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, or ZHETRF with a call to SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, or ZHETRS respectively.

To solve the system of equations with one or more right-hand sides, follow the call to SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, or ZHPTRF with a call to SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, or ZHPTRS respectively.

Table 148: Data Types.

A*, *B*, *work	**Subroutine**
Short-precision real	SSYTRF$^{\Delta}$, SSPTRF$^{\Delta}$
Long-precision real	DSYTRF$^{\Delta}$, DSPTRF$^{\Delta}$
Short-precision complex	CSYTRF$^{\Delta}$, CHETRF$^{\Delta}$, CSPTRF$^{\Delta}$, CHPTRF$^{\Delta}$
Long-precision complex	ZSYTRF$^{\Delta}$, ZHETRF$^{\Delta}$, ZSPTRF$^{\Delta}$, ZHPTRF$^{\Delta}$
$^{\Delta}$LAPACK	

Syntax

Language	Syntax
Fortran	CALL SSYTRF | DSYTRF | CSYTRF | ZSYTRF | CHETRF | ZHETRF (*uplo*, *n*, *a*, *lda*, *ipiv*, *work*, *lwork*, *info*) CALL SSPTRF | DSPTRF | CSPTRF | ZSPTRF | CHPTRF | ZHPTRF (*uplo*, *n*, *ap*, *ipiv*, *info*)

C and C++	ssytrf | dsytrf | csytrf | zsytrf | chetrf | zhetrf (*uplo*, *n*, *a*, *lda*, *ipiv*, *work*, *lwork*, *info*);
	ssptrf | dsptrf | csptrf | zsptrf | chptrf | zhptrf (*uplo*, *n*, *ap*, *ipiv*, *info*);

LAPACKE	info = LAPACKE_ssytrf | LAPACKE_dsytrf | LAPACKE_csytrf | LAPACKE_zsytrf | LAPACKE_chetrf | LAPACKE_zhetrf (*matrix_layout*, *uplo*, *n*, *a*, *lda*, *ipiv*);
	info = LAPACKE_ssptrf | LAPACKE_dsptrf | LAPACKE_csptrf | LAPACKE_zsptrf | LAPACKE_chptrf | LAPACKE_zhptrf (*matrix_layout*, *uplo*, *n*, *ap*, *ipiv*);

Note:

- The output from the SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, or ZHETRF factorization routines should only be used as input to SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, or ZHETRS respectively.
- The output from the SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, or ZHPTRF factorization routines should only be used as input to SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, or ZHPTRS respectively.

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

uplo

indicates whether the upper or lower triangular part of the matrix ***A*** is referenced, where:

If *uplo* = 'U', the upper triangular part is referenced.

If *uplo* = 'L', the lower triangular part is referenced.

Specified as: a single character; *uplo* = 'U' or 'L'.

n

is the order of matrix ***A*** used in the computation.

Specified as: an integer; $n \geq 0$.

a

is the indefinite real symmetric, complex symmetric, or complex Hermitian matrix ***A*** of order *n*.

If *uplo* = 'U', it is stored in upper storage mode.

If *uplo* = 'L', it is stored in lower storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 148 on page 593.

ap

is the indefinite real symmetric, complex symmetric, or complex Hermitian matrix ***A*** of order *n*. It is stored in an array, referred to as AP, where:

If *uplo* = 'U', it is stored in upper-packed storage mode.

If *uplo* = 'L', it is stored in lower-packed storage mode.

Specified as: one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 148 on page 593.

lda

is the leading dimension of the array specified for ***A***.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

ipiv

See On Return.

work

is a work area used by these subroutines, where:

If *lwork* = 0, *work* is ignored.

If $lwork \neq 0$, the size of *work* is determined as follows:

- If $lwork \neq -1$, *work* is (at least) of length *lwork*.
- If $lwork = -1$, *work* is (at least) of length 1.

Specified as: an area of storage containing numbers of the data type indicated in Table 148 on page 593.

lwork

is the number of elements in array `WORK`.

Specified as: an integer; where:

- If *lwork* = 0, the subroutine dynamically allocates the workspace needed for use during this computation. The work area is deallocated before control is returned to the calling program.
- If *lwork* = -1, subroutine performs a workspace query and returns the optimal required size of *work* in $work_1$. No computation is performed and the subroutine returns after error checking is complete.
- Otherwise, $lwork \geq 1$. It is suggested that the user specify $lwork \geq 8n$.

info

See On Return.

On Return

a

is the transformed matrix ***A*** containing the results of the factorization. See "Function" on page 596.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 148 on page 593.

ap

is the transformed matrix ***A*** containing the results of the factorization. See "Function" on page 596.

Returned as: a one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 148 on page 593.

ipiv

If *info* = 0, *ipiv* contains the pivot indices.

If $ipiv_k > 0$, then rows and columns *k* and $ipiv_k$ were interchanged and $\boldsymbol{D}_{k,k}$ is a 1 × 1 diagonal block.

If *uplo* = 'U' and $ipiv_k = ipiv_{k-1} < 0$, then rows and columns *k*-1 and -$ipiv_k$ were interchanged and $\boldsymbol{D}_{k-1:k,k-1:k}$ is a 2 × 2 diagonal block.

If *uplo* = 'L' and $ipiv_k = ipiv_{k+1} < 0$, then rows and columns *k*+1 and -$ipiv_k$ were interchanged and $\boldsymbol{D}_{k:k+1,k:k+1}$ is a 2 × 2 diagonal block.

Returned as: a one-dimensional integer array of (at least) length *n*, containing integers.

work

is a work area used by this subroutine if *lwork* ≠ 0, where:

If *lwork* ≠ 0 and *lwork* ≠ -1, its size is (at least) of length *lwork*.

If *lwork* = -1, its size is (at least) of length 1.

Returned as: an area of storage, where:

If *lwork* ≥ 1 or *lwork* = -1, then $work_1$ is set to the optimal *lwork* value and all other elements of *work* are overwritten.

info

has the following meaning:

- If *info* = 0, the factorization completed successfully.
- If *info* > 0, the factorization was unsuccessful and *info* is set to *i* where d_{ii} is exactly zero.

Returned as:

- For SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, and ZHPTRF, returned as: an integer; info ≥ 0.
- For LAPACKE_ssytrf, LAPACKE_dsytrf, LAPACKE_csytrf, LAPACKE_zsytrf, LAPACKE_chetrf, LAPACKE_zhetrf, LAPACKE_ssptrf, LAPACKE_dsptrf, LAPACKE_csptrf, LAPACKE_zsptrf, LAPACKE_chptrf, and LAPACKE_zhptrf, returned as an integer function value; info ≥ 0.

Specified as: an integer; *info* ≥ 0.

Notes

1. These subroutines accept lowercase letters for the *uplo* argument.
2. In your C program, argument *info* must be passed by reference.
3. *a*, *ap*, *ipiv*, and *work* must have no common elements; otherwise, results are unpredictable.
4. For a description of how real and complex symmetric matrices are stored in lower or upper storage mode, see "Lower Storage Mode" on page 83 or "Upper Storage Mode" on page 83, respectively.

   For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see "Complex Hermitian Matrix" on page 84.

5. For a description of how real and complex symmetric matrices are stored in lower- or upper-packed storage mode, see "Lower-Packed Storage Mode" on page 80 or "Upper-Packed Storage Mode" on page 82, respectively.

   For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see "Complex Hermitian Matrix" on page 84.
6. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.
7. For best performance, specify *lwork* = 0.

Function

For SSYTRF, DSYTRF, CSYTRF, ZSYTRF, SSPTRF, DSPTRF, CSPTRF, and ZSPTRF:

The indefinite real or complex symmetric matrix ***A*** is factored using the Bunch-Kaufman diagonal pivoting method, where ***A*** is expressed as one of the following:

$\boldsymbol{A} = \boldsymbol{UDU}^{\mathrm{T}}$

$\boldsymbol{A} = \boldsymbol{LDL}^{\mathrm{T}}$

where:

U is a product of permutation and unit upper triangular matrices.

L is a product of permutation and unit lower triangular matrices.

D is a symmetric block diagonal matrix, consisting of 1 × 1 and 2 × 2 diagonal blocks.

Matrix ***A*** is stored as follows:

- For SSYTRF, DSYTRF, CSYTRF, and ZSYTRF, matrix ***A*** is stored in upper or lower storage mode.
- For SSPTRF, DSPTRF, CSPTRF, and ZSPTRF, matrix ***A*** is stored in upper- or lower-packed storage mode.

For CHETRF, ZHETRF, CHPTRF, and ZHPTRF:

The indefinite complex Hermitian matrix ***A*** is factored using the Bunch-Kaufman diagonal pivoting method, where ***A*** is expressed as one of the following:

$\boldsymbol{A} = \boldsymbol{UDU}^{\mathrm{H}}$

$\boldsymbol{A} = \boldsymbol{LDL}^{\mathrm{H}}$

where:

U is a product of permutation and unit upper triangular matrices.

L is a product of permutation and unit lower triangular matrices.

D is a complex Hermitian block diagonal matrix, consisting of 1 × 1 and 2 × 2 diagonal blocks.

Matrix ***A*** is stored as follows:

- For CHETRF and ZHETRF, matrix ***A*** is stored in upper or lower storage mode.
- For CHPTRF and ZHPTRF, matrix ***A*** is stored in upper- or lower-packed storage mode.

If *n* is 0, no computation is performed. See references ["9" on page 1253] and ["19" on page 1254].

Error conditions

Resource Errors
   lwork = 0 and unable to allocate work area

Computational Errors
   Matrix ***A*** is singular.

- The factorization completed but the block diagonal matrix ***D*** is exactly singular. *info* is set to *i*, where d_{ii} is exactly zero. This diagonal element is identified in the computational error message.
- The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2147 is set to be unlimited in the ESSL error option table. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. *n* < 0
3. *lda* ≤ 0
4. *n* > *lda*
5. *lwork* ≠ 0 and *lwork* ≠ -1 and *lwork* < the minimum required value

Examples

Example 1

This example shows a factorization of the indefinite real symmetric matrix ***A*** of order 8.

Matrix ***A*** is the same matrix factored in the Example 1 for DBSTRF.

Note: Because *lwork* = 0, the subroutine dynamically allocates WORK.

Call Statement and Input:

```
              UPLO   N    A   LDA   IPIV    WORK LWORK INFO
               |     |    |    |     |        |     |     |
CALL DSYTRF( 'L' , 8 , A , 8 , IPIV , WORK,  0 , INFO )
```

```
                                                          ┐
         |  3.0    .     .     .     .     .     .     . |
         |  5.0   3.0    .     .     .     .     .     . |
         | -2.0   2.0   0.0    .     .     .     .     . |
 A   =   |  2.0  -2.0   0.0   8.0    .     .     .     . |
         |  3.0   5.0  -2.0  -6.0  12.0    .     .     . |
         | -5.0  -3.0   2.0 -10.0   6.0  16.0    .     . |
         | -2.0   2.0   0.0  -8.0   8.0   8.0   6.0    . |
         | -3.0  -5.0   6.0 -14.0   6.0  20.0  18.0  34.0 |
                                                          ┘
```

Output:

```
                                                   ┐
         |  3.0   .    .    .    .    .    .    . |
         |  5.0  3.0   .    .    .    .    .    . |
         |  1.0 -1.0  4.0   .    .    .    .    . |
 A   =   | -1.0  1.0 -1.0  8.0   .    .    .    . |
         |  1.0  0.0  0.0 -1.0  1.0   .    .    . |
         |  0.0 -1.0  1.0 -1.0  3.0  1.0   .    . |
         |  1.0 -1.0  1.0 -1.0 -1.0  1.0  2.0   . |
         | -1.0  0.0  1.0 -1.0  1.0  0.0  1.0 16.0 |
                                                   ┘
```

```
IPIV = ( -2 -2 3 4 -6 -6 7 8 )
```

```
INFO  =   0
```

Example 2

This example shows a factorization of the indefinite complex symmetric matrix ***A*** of order 4.

Matrix ***A*** is:

```
                                                                    ┐
         | ( 0.368,-0.319) (-0.021, 0.022) (0.312,-0.105) (-0.070, 0.263) |
         | (-0.021, 0.022) ( 0.259,-0.148) (0.212,-0.237) ( 0.071, 0.370) |
         | ( 0.312,-0.105) ( 0.212,-0.237) (0.273,-0.041) ( 0.384,-0.056) |
         | (-0.070, 0.263) ( 0.071, 0.370) (0.384,-0.056) (-0.230, 0.085) |
                                                                    ┘
```

Note: Because *lwork* = 0, the subroutine dynamically allocates WORK.

Call Statement and Input:

```
             UPLO  N   A  LDA  IPIV   WORK LWORK INFO
              |    |   |   |    |       |    |     |
CALL ZSYTRF( 'L' , 4 , A , 4 , IPIV , WORK,  0 , INFO )
```

```
                                                                     ┐
        | ( 0.368,-0.319)        .               .               .       |
 A   =  | (-0.021, 0.022) ( 0.259,-0.148)        .               .       |
        | ( 0.312,-0.105) ( 0.212,-0.237) ( 0.273,-0.041)        .       |
        | (-0.070, 0.263) ( 0.071, 0.370) ( 0.384,-0.056) (-0.230, 0.085)|
                                                                     ┘
```

Output:

```
                                                                      ┐
        | ( 0.368,-0.319)        .               .               .        |
 A   =  | (-0.062, 0.006) ( 0.258,-0.147)        .               .        |
        | ( 0.625, 0.257) ( 1.085,-0.335) ( 0.333, 0.315)        .        |
        | (-0.462, 0.314) (-0.444, 1.248) (-0.437,-1.386) ( 0.841, 0.431) |
                                                                      ┘
```

```
IPIV = ( 1 2 4 4 )
```

```
INFO  =   0
```

Example 3

This example shows a factorization of the indefinite complex Hermitian matrix ***A*** of order 4.

Matrix ***A*** is:

```
                                                                      ┐
         | (-0.550, 0.000) ( 0.015, 0.262) (-0.034, 0.134) ( 0.137, 0.012) |
         | ( 0.015,-0.262) (-0.609, 0.000) (-0.062,-0.150) (-0.083, 0.021) |
         | (-0.034,-0.134) (-0.062, 0.150) (-0.560, 0.000) (-0.126, 0.053) |
         | ( 0.137,-0.012) (-0.083,-0.021) (-0.126,-0.053) (-0.558, 0.000) |
                                                                      ┘
```

Notes:

1. Because *lwork* = 0, the subroutine dynamically allocates WORK.
2. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input:

```
             UPLO  N   A  LDA  IPIV   WORK LWORK INFO
              |    |   |   |    |       |    |     |
CALL ZHETRF( 'L' , 4 , A , 4 , IPIV , WORK,  0 , INFO )
```

```
                                                                    ┐
         | (-0.550,  .   )         .               .               .       |
 A   =   | ( 0.015,-0.262) (-0.609,  .   )         .               .       |
         | (-0.034,-0.134) (-0.062, 0.150) (-0.560,  .   )         .       |
         | ( 0.137,-0.012) (-0.083,-0.021) (-0.126,-0.053) (-0.558,  .  ) |
                                                                    ┘
```

Output:

```
                                                                     ┐
         | (-0.550, 0.000)         .               .               .        |
 A   =   | (-0.027, 0.476) (-0.483, 0.000)         .               .        |
         | ( 0.062, 0.244) (-0.002,-0.269) (-0.490, 0.000)         .        |
         | (-0.249, 0.022) ( 0.152,-0.091) ( 0.244,-0.002) (-0.479, 0.000) |
                                                                     ┘
```

```
IPIV = ( 1 2 4 4 )
```

```
INFO  =   0
```

Example 4

This example shows a factorization of the indefinite real symmetric matrix ***A*** of order 8.

Matrix ***A*** is the same matrix factored in the Example 1 for DBSTRF.

Call Statement and Input:

```
             UPLO  N   AP   IPIV   INFO
              |    |    |     |      |
CALL DSPTRF( 'L' , 8 , AP , IPIV , INFO )
```

```
AP = (   3.0  5.0 -2.0  2.0  3.0 -5.0 -2.0 -3.0,
              3.0  2.0 -2.0  5.0 -3.0  2.0 -5.0,
                   0.0  0.0 -2.0  2.0  0.0  6.0,
                        8.0 -6.0-10.0 -8.0-14.0,
                            12.0  6.0  8.0  6.0,
                                 16.0  8.0 20.0,
                                       6.0 18.0,
                                           34.0 )
```

Output:

```
AP = (   3.0   5.0   1.0  -1.0   1.0   0.0   1.0  -1.0,
               3.0  -1.0   1.0   0.0  -1.0  -1.0   0.0,
                     4.0  -1.0   0.0   1.0   1.0   1.0,
                           8.0  -1.0  -1.0  -1.0  -1.0,
                                 1.0   3.0  -1.0   1.0,
                                       1.0   1.0   0.0,
                                             2.0   1.0,
                                                  16.0 )
```

```
IPIV = ( -2 -2 3 4 -6 -6 7 8 )
```

```
INFO  =   0
```

Example 5

This example shows a factorization of the indefinite complex symmetric matrix ***A*** of order 4.

Matrix ***A*** is the same matrix factored in the Example 2 for ZSYTRF.

Call Statement and Input:

```
             UPLO  N   AP   IPIV   INFO
              |    |    |     |      |
CALL ZSPTRF( 'L' , 4 , AP , IPIV , INFO )
```

```
AP = ( ( 0.368, -0.319), (-0.021,  0.022), (0.312, -0.105),(-0.070,  0.263),
                         ( 0.259, -0.148), (0.212, -0.237),( 0.071,  0.370),
                                           (0.273, -0.041),( 0.384, -0.056),
                                                           (-0.230,  0.085) )
```

Output:

```
AP = ( (0.368, -0.319), (-0.062, 0.006), (0.625,  0.257), (-0.462,  0.314),
                        ( 0.258,-0.147), (1.085, -0.335), (-0.444,  1.248),
                                         (0.333,  0.315), (-0.437, -1.386),
                                                          ( 0.841,  0.431) )
```

```
IPIV = (  1 2 4 4 )
```

```
INFO  =   0
```

Example 6

This example shows a factorization of the indefinite complex Hermitian matrix ***A*** of order 4.

Matrix ***A*** is the same matrix factored in the Example 3 for ZHETRF.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input:

```
             UPLO  N   AP   IPIV   INFO
              |    |    |     |      |
CALL ZHPTRF( 'L' , 4 , AP , IPIV , INFO )
```

```
 AP = ( (-0.550,   .   ), ( 0.015,-0.262), (-0.034,-0.134), ( 0.137,-0.012),
                          (-0.609,  .    ), (-0.062, 0.150), (-0.083,-0.021),
                                            (-0.560,  .   ), (-0.126,-0.053),
                                                             (-0.558,  .   ) )
```

Output:

```
 AP = ( (-0.550, 0.000), (-0.027, 0.476), ( 0.062,  0.244), (-0.249,  0.022),
                         (-0.484, 0.000), (-0.002, -0.269), ( 0.152, -0.091),
                                          (-0.490,  0.000), ( 0.245, -0.002),
                                                            (-0.479,  0.000) )
```

```
IPIV = (  1 2 3 4 )
```

```
INFO  =   0
```

SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, ZHETRS, SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, and ZHPTRS (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system ***AX*** = ***B*** for ***X***, where ***A*** is an indefinite real or complex symmetric or complex Hermitian matrix and ***X*** and ***B*** are general matrices.

SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, or ZHETRS use the results of the factorization of matrix ***A***, produced by a preceding call to SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, or ZHETRF, respectively.

SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, or ZHPTRS use the results of the factorization of matrix ***A***, produced by a preceding call to SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, or ZHPTRF, respectively.

Table 149: Data Types.

A, ***B***	**Subroutine**
Short-precision real	SSYTRS$^{\Delta}$, SSPTRS$^{\Delta}$
Long-precision real	DSYTRS$^{\Delta}$, DSPTRS$^{\Delta}$
Short-precision complex	CSYTRS$^{\Delta}$, CHETRS$^{\Delta}$, CSPTRS$^{\Delta}$, CHPTRS$^{\Delta}$
Long-precision complex	ZSYTRS$^{\Delta}$, ZHETRS$^{\Delta}$, ZSPTRS$^{\Delta}$, ZHPTRS$^{\Delta}$
$^{\Delta}$LAPACK	

Syntax

Language	Syntax
Fortran	CALL SSYTRS | DSYTRS | CSYTRS | ZSYTRS | CHETRS | ZHETRS (*uplo*, *n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*, *info*) CALL SSPTRS | DSPTRS | CSPTRS | ZSPTRS | CHPTRS | ZHPTRS (*uplo*, *n*, *nrhs*, *ap*, *ipiv*, *b*, *ldb*, *info*)

C and C++	ssytrs | dsytrs | csytrs | zsytrs | chetrs | zhetrs (*uplo*, *n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*, *info*);
	ssptrs | dsptrs | csptrs | zsptrs | chptrs | zhptrs (*uplo*, *n*, *nrhs*, *ap*, *ipiv*, *b*, *ldb*, *info*);

LAPACKE	info = LAPACKE_ssytrs | LAPACKE_dsytrs | LAPACKE_csytrs | LAPACKE_zsytrs | LAPACKE_chetrs | LAPACKE_zhetrs (*matrix_layout*, *uplo*, *n*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*);
	info = LAPACKE_ssptrs | LAPACKE_dsptrs | LAPACKE_csptrs | LAPACKE_zsptrs | LAPACKE_chptrs | LAPACKE_zhptrs (*matrix_layout*, *uplo*, *n*, *nrhs*, *ap*, *ipiv*, *b*, *ldb*);

On Entry

matrix_layout
indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

uplo
indicates whether the upper or lower triangular part of the matrix ***A*** is referenced, where:

If *uplo* = 'U', the upper triangular part is referenced.

If *uplo* = 'L', the lower triangular part is referenced.

Specified as: a single character; *uplo* = 'U' or 'L'.

n
is the order of matrix ***A*** used in the computation.

Specified as: an integer; $n \geq 0$.

nrhs
is the number of right-hand sides; that is, the number of columns of matrix ***B***.

Specified as: an integer; $nrhs \geq 0$.

a
is the factorization of indefinite real symmetric, complex symmetric, or complex Hermitian matrix ***A*** of order *n* produced by a preceding call to SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, or ZHETRF, respectively.

If *uplo* = 'U', it is stored in upper storage mode.

If *uplo* = 'L', it is stored in lower storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 149 on page 601.

ap
is the factorization of indefinite real symmetric, complex symmetric, or complex Hermitian matrix ***A*** of order *n*, produced by a preceding call to SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, or ZHPTRF, respectively.

If *uplo* = 'U', it is stored in upper-packed storage mode.

If *uplo* = 'L', it is stored in lower-packed storage mode.

Specified as: a one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 149 on page 601.

lda
is the leading dimension of the array specified for ***A***.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

ipiv
is the array containing the pivot indices produced by a preceding call to SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, or ZHPTRF, respectively.

Specified as: a one-dimensional integer array of (at least) length *n*, containing integers.

b
is the general matrix ***B*** containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of matrix ***B***.

Specified as: an *ldb* by *nrhs* array, containing numbers of the data type indicated in Table 149 on page 601.

ldb
is the leading dimension of the array specified for ***B***.

Specified as: an integer; $ldb>0$ and $ldb \geq n$.

info
See On Return.

On Return

b

is the matrix ***X***, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of ***X***.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 149 on page 601.

info

has the following meaning:

If *info*=0, the factorization completed successfully.

Returned as:

- For SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, ZHETRS, SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, and ZHPTRS, returned as: an integer; info ≥ 0.
- For LAPACKE_ssytrs, LAPACKE_dsytrs, LAPACKE_csytrs, LAPACKE_zsytrs, LAPACKE_chetrs, LAPACKE_zhetrs, LAPACKE_ssptrs, LAPACKE_dsptrs, LAPACKE_csptrs, LAPACKE_zsptrs, LAPACKE_chptrs, and LAPACKE_zhptrs, returned as an integer function value; info ≥ 0.

Notes

1. These subroutines accept lowercase letters for the *uplo* argument.
2. In your C program, argument *info* must be passed by reference.
3. *a*, *ap*, *b* and *ipiv* must have no common elements; otherwise, results are unpredictable.
4. For a description of how real and complex symmetric matrices are stored in lower or upper storage mode, see “Lower Storage Mode” on page 83 or “Upper Storage Mode” on page 83, respectively.

   For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see “Complex Hermitian Matrix” on page 84.
5. For a description of how real and complex symmetric matrices are stored in lower- or upper-packed storage mode, see “Lower-Packed Storage Mode” on page 80 or “Upper-Packed Storage Mode” on page 82, respectively.

   For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see “Complex Hermitian Matrix” on page 84.

Function

These subroutines solve the system ***AX*** = ***B*** for ***X***, where ***A*** is a real or complex symmetric or complex Hermitian matrix and ***X*** and ***B*** are general matrices.

SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, or ZHETRS use the results of the factorization of matrix ***A***, produced by a preceding call to SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, or ZHETRF, respectively.

SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, or ZHPTRS use the results of the factorization of matrix ***A***, produced by a preceding call to SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, or ZHPTRF, respectively.

If *n* is 0 or *nrhs* is 0, no computation is performed and the subroutine returns after doing some parameter checking. See references [“9” on page 1253] and [“19” on page 1254].

Error conditions

Computational Errors

None

Note: If the factorization performed by SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, or ZHPTRF failed because matrix ***A*** is singular, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. *n* < 0
3. *nrhs* < 0
4. *lda* ≤ 0
5. *n* > *lda*
6. *ldb* ≤ 0
7. *n* > *ldb*

Examples

Example 1

This example shows how to solve the system ***AX***=***B***, for three right-hand sides, where indefinite real symmetric matrix ***A*** is the same matrix factored in the Example 1 for DSYTRF.

Call Statement and Input:

```
              UPLO   N NRHS  A  LDA  IPIV   B  LDB  INFO
               |     |   |   |   |     |    |   |     |
CALL DSYTRS( 'L' , 8 , 3 , A , 8 , IPIV , B , 8 , INFO )
```

A = (same as output A in Example 1)

IPIV = (same as output IPIV in Example 1)

```
                                   ┐
         |   1.0  -38.0  47.0 |
         |   7.0  -10.0  73.0 |
         |   6.0   52.0   2.0 |
 B   =   | -30.0 -228.0 -42.0 |
         |  32.0  183.0 105.0 |
         |  34.0  297.0   9.0 |
         |  32.0  244.0  44.0 |
         |  62.0  497.0  61.0 |
                                   ┘
```

Output:

```
                                ┐
         | 1.0   1.0   8.0 |
         | 1.0   2.0   7.0 |
         | 1.0   3.0   6.0 |
 B   =   | 1.0   4.0   5.0 |
         | 1.0   5.0   4.0 |
         | 1.0   6.0   3.0 |
         | 1.0   7.0   2.0 |
         | 1.0   8.0   1.0 |
                                ┘
```

```
INFO  =   0
```

Example 2

This example shows how to solve the system ***AX***=***B*** for three right-hand sides, where indefinite complex symmetric matrix ***A*** is the same matrix factored in the Example 2 for ZSYTRF.

Call Statement and Input:

```
              UPLO   N NRHS  A  LDA  IPIV   B  LDB  INFO
               |     |   |   |   |     |    |   |     |
CALL ZSYTRS( 'L' , 4 , 3 , A , 4 , IPIV , B , 4 , INFO )
```

A = (same as output A in Example 2)

IPIV = (same as output IPIV in Example 2)

```
                                                         ┐
        | ( 1.000, 1.000) (-1.000, 1.000) ( 2.000, 0.000) |
B   =   | ( 1.000, 1.000) (-1.000, 1.000) ( 0.000, 1.000) |
        | ( 1.000, 1.000) ( 1.000,-1.000) ( 0.000, 1.000) |
        | ( 1.000, 1.000) ( 1.000,-1.000) (-2.000, 0.000) |
                                                         ┘
```

Output:

```
                                                          ┐
        | ( 0.409, -0.663) (-0.582,-1.410) ( 2.484, 2.216) |
B   =   | (-1.664, -0.552) (-1.503,-4.837) (-3.577, 2.575) |
        | ( 2.388,  4.010) ( 1.260,-0.430) (-1.273, 0.177) |
        | ( 1.562,  0.164) ( 6.213, 1.471) (-0.980,-2.551) |
                                                          ┘
```

```
INFO  =   0
```

Example 3

This example shows how to solve the system ***AX***=***B*** for three right-hand sides, where indefinite complex Hermitian matrix ***A*** is the same matrix factored in the Example 3 for ZHETRF.

Call Statement and Input:

```
             UPLO   N NRHS  A  LDA  IPIV   B  LDB  INFO
              |     |   |   |   |     |    |   |     |
CALL ZHETRS( 'L' , 4 , 3 , A , 4 , IPIV , B , 4 , INFO )
```

A = (same as output A in Example 3)

IPIV = (same as output IPIV in Example 3)

```
                                                         ┐
        | ( 1.000, 1.000) (-1.000, 1.000) ( 2.000, 0.000) |
B   =   | ( 1.000, 1.000) (-1.000, 1.000) ( 0.000, 1.000) |
        | ( 1.000, 1.000) ( 1.000,-1.000) ( 0.000, 1.000) |
        | ( 1.000, 1.000) ( 1.000,-1.000) (-2.000, 0.000) |
                                                         ┘
```

Output:

```
                                                         ┐
        | (-1.623,-4.385) ( 2.635,-1.111) (-2.436,-0.306) |
B   =   | (-3.533,-0.212) ( 1.865,-2.830) (-0.866,-0.308) |
        | (-1.742,-1.724) (-1.576, 1.575) (-0.478,-1.172) |
        | (-1.537,-2.115) (-1.047, 1.608) ( 3.093, 0.365) |
                                                         ┘
```

```
INFO  =   0
```

Example 4

This example shows how to solve the system ***AX***=***B*** for three right-hand sides, where indefinite real symmetric matrix ***A*** is the same matrix factored in the Example 1 for DSYTRF.

Call Statement and Input:

```
              UPLO   N NRHS   AP    IPIV    B  LDB   INFO
               |     |   |     |      |     |   |      |
CALL DSPTRS( 'L' , 8 , 3 , AP , IPIV , B , 8 , INFO )
```

AP = (same as output AP in Example 4)

IPIV = (same as output IPIV in Example 4)

```
                                   ┐
         |   1.0  -38.0  47.0 |
         |   7.0  -10.0  73.0 |
         |   6.0   52.0   2.0 |
 B   =   | -30.0 -228.0 -42.0 |
         |  32.0  183.0 105.0 |
         |  34.0  297.0   9.0 |
         |  32.0  244.0  44.0 |
         |  62.0  497.0  61.0 |
                                   ┘
```

Output:

```
                                ┐
         | 1.0   1.0   8.0 |
         | 1.0   2.0   7.0 |
         | 1.0   3.0   6.0 |
 B   =   | 1.0   4.0   5.0 |
         | 1.0   5.0   4.0 |
         | 1.0   6.0   3.0 |
         | 1.0   7.0   2.0 |
         | 1.0   8.0   1.0 |
                                ┘
```

```
INFO  =   0
```

Example 5

This example shows how to solve the system ***AX***=***B*** for three right-hand sides, where indefinite complex symmetric matrix ***A*** is the same matrix factored in the Example 2 for ZSYTRF.

Call Statement and Input:

```
              UPLO   N NRHS   AP    IPIV    B  LDB   INFO
               |     |   |     |      |     |   |      |
CALL ZSPTRS( 'L' , 4 , 3 , AP , IPIV , B , 4 , INFO )
```

AP = (same as output AP in Example 5)

IPIV = (same as output IPIV in Example 5)

```
                                                             ┐
         | ( 1.000, 1.000) (-1.000, 1.000) ( 2.000, 0.000) |
 B   =   | ( 1.000, 1.000) (-1.000, 1.000) ( 0.000, 1.000) |
         | ( 1.000, 1.000) ( 1.000,-1.000) ( 0.000, 1.000) |
         | ( 1.000, 1.000) ( 1.000,-1.000) (-2.000, 0.000) |
                                                             ┘
```

Output:

```
                                                             ┐
         | ( 0.409,-0.663) (-0.582,-1.410) ( 2.484, 2.216) |
 B   =   | (-1.664,-0.552) (-1.503,-4.837) (-3.577, 2.575) |
         | ( 2.388, 4.010) ( 1.260,-0.430) (-1.273, 0.177) |
         | ( 1.562, 0.164) ( 6.213, 1.471) (-0.980,-2.551) |
                                                             ┘
```

```
INFO  =   0
```

Example 6

This example shows how to solve the system ***AX***=***B*** for three right-hand sides, where indefinite complex Hermitian matrix ***A*** is the same matrix factored in the Example 3 for ZHETRF.

Call Statement and Input:

```
              UPLO   N NRHS   AP    IPIV    B  LDB  INFO
                |    |   |     |      |     |   |     |
CALL ZHPTRS( 'L' , 4 , 3 , AP , IPIV , B , 4 , INFO )
```

AP = (same as output AP in Example 6)

`IPIV` = (same as output `IPIV` in Example 6)

```
                                                         ┐
        | ( 1.000, 1.000) (-1.000, 1.000) ( 2.000, 0.000) |
 B   =  | ( 1.000, 1.000) (-1.000, 1.000) ( 0.000, 1.000) |
        | ( 1.000, 1.000) ( 1.000,-1.000) ( 0.000, 1.000) |
        | ( 1.000, 1.000) ( 1.000,-1.000) (-2.000, 0.000) |
                                                         ┘
```

Output:

```
                                                         ┐
        | (-1.623,-4.385) ( 2.635,-1.111) (-2.436,-0.306) |
 B   =  | (-3.533,-0.212) ( 1.865,-2.830) (-0.866,-0.308) |
        | (-1.742,-1.724) (-1.576, 1.575) (-0.478,-1.172) |
        | (-1.537,-2.115) (-1.047, 1.608) ( 3.093, 0.365) |
                                                         ┘
```

```
INFO  =   0
```

DBSSV (Symmetric Indefinite Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

The DBSSV subroutine solves a system of linear equations ***AX*** = ***B*** for ***X***, where ***A*** is a real symmetric indefinite matrix, and ***X*** and ***B*** are real general matrices.

The matrix ***A***, stored in upper- or lower-packed storage mode, is factored using the Bunch-Kaufman diagonal pivoting method, where ***A*** is expressed as:

$\boldsymbol{A} = \boldsymbol{UDU}^{\mathrm{T}}$ or
$\boldsymbol{A} = \boldsymbol{LDL}^{\mathrm{T}}$

where:

U is a product of permutation and unit upper triangular matrices.

L is a product of permutation and unit lower triangular matrices.

D is a symmetric block diagonal matrix, consisting of 1 × 1 and 2 × 2 diagonal blocks.

Table 150: Data Types.

Data Types		
A*, *B	***ipvt***	**Subroutine**
Long-precision real	Integer	DBSSV

Syntax

Language	Syntax
Fortran	CALL DBSSV (*uplo*, *n*, *nrhs*, *ap*, *ipvt*, *bx*, *ldb*, *nsinfo*)
C and C++	dbssv (*uplo*, *n*, *nrhs*, *ap*, *ipvt*, *bx*, *ldb*, *nsinfo*);

On Entry

uplo

indicates whether matrix ***A*** is stored in upper- or lower-packed storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper-packed storage mode.

If *uplo* = 'L', ***A*** is stored in lower-packed storage mode.

Specified as: a single character. It must be 'U' or 'L'.

n

is the order *n* of matrix ***A*** and the number of rows of matrix **B**.

Specified as: an integer; $n \geq 0$.

nrhs

is the number of right-hand sides; i.e., the number of columns of matrix ***B***.

Specified as: an integer; $nrhs \geq 0$.

ap

is array, referred to as AP, in which matrix ***A***, to be factored, is stored in upper- or lower-packed storage mode.

Specified as: a one-dimensional array of length *nsinfo*, containing numbers of the data type indicated in Table 150 on page 608. See "Notes" on page 610.

ipvt

See On Return.

bx

is the matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of matrix ***B***.

Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 150 on page 608.

ldb

is the leading dimension of the array specified for ***B***.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

nsinfo

is the number of elements in array, AP.

If $n \leq nco$, $nsinfo = n(n + 1) / 2$

Where:

$$nco = \sqrt{2.0(ics)}$$

ics is the size in doublewords of the data cache. The data cache size can be obtained by utilizing the following C language code fragment:

```
#include <sys/systemcfg.h>
int ics;
         .
         .
         .
ics=_system_configuration.dcache_size/8;
```

Otherwise, to determine a sufficient amount of storage, use the following processor-independent formula:

$n0 = 100$
$ns = (n + n0)(n + n0 + 1) / 2 + n(n0)$
For *uplo* = 'L',
$nsinfo \geq ns$
For *uplo* = 'U',
$n1 = (n + 1) / 2$
$nt = n((n + 1) / 2)$
$nt1 = n1(n1 + 1)$
$ns1 = nt + nt1$
$nsinfo \geq \max(ns, ns1)$

To determine the minimal amount of storage see “Notes” on page 610.

Specified as: an integer; *nsinfo* > 0.

On Return

ap

is the transformed matrix ***A*** of order *n*, containing the results of the factorization.

If *nsinfo* ≥ 0 and *n* > *nco*, additional information that can be used to obtain a minimum *nsinfo* is stored in AP(1).

See “Notes” on page 610 and “Function” on page 610.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 150 on page 608.

ipvt

is an integer vector of length *n*, containing the pivot information necessary to construct the factored form of ***A***.

Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 150 on page 608.

bx

is the matrix ***X***, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of ***X***.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 150 on page 608.

nsinfo

indicates the result of the computation.

- If *nsinfo* = 0, the subroutine completed successfully.
- If *nsinfo* > 0, factorization was unsuccessful and array B was not updated. *nsinfo* is set to *i* where d_{ii} is exactly zero.
- If *nsinfo* < 0, factorization did not take place and the arrays, AP and B, remain unchanged. |*nsinfo*| is the minimal storage required for factorization to take place. Error message 2200 is issued and execution terminates, unless you have used ERRSET to make error code 2200 recoverable. See “What Can You Do about ESSL Input-Argument Errors?” on page 64.

Specified as: an integer.

Notes

1. This subroutine accepts lowercase letters for the *uplo* argument.
2. In your C program, argument *nsinfo* must be passed by reference.
3. In the input array specified for *ap*, the first $n(n+1)/2$ elements are matrix elements. The additional locations, required in the array, are used for working storage.
4. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
5. On return, if $nsinfo \geq 0$ and $n > nco$, *ap* contains additional information in AP(1) that can be used to obtain the minimal required *nsinfo*. This information can be accessed using the following code fragment:

```
REAL*8    AP(NSINFO)
INTEGER    API(2)
EQUIVALENCE(API, AP)
          .
          .
          .
NSINFOMIN = API(2)
```

6. For a description of how a symmetric matrix is stored in upper- or lower-packed storage mode in an array, see "Symmetric Matrix" on page 80.

Function

The system ***AX*** = ***B*** is solved for ***X***, where ***A*** is a real symmetric indefinite matrix, and ***X*** and ***B*** are real general matrices.

The matrix ***A***, stored in upper- or lower-packed storage mode, is factored using the Bunch-Kaufman diagonal pivoting method, where ***A*** is expressed as:

$\boldsymbol{A} = \boldsymbol{UDU}^{\mathrm{T}}$ or
$\boldsymbol{A} = \boldsymbol{LDL}^{\mathrm{T}}$

where:

U is a product of permutation and unit upper triangular matrices.

L is a product of permutation and unit lower triangular matrices.

D is a symmetric block diagonal matrix, consisting of 1 × 1 and 2 × 2 diagonal blocks.

If *n* is 0, no computation is performed and the subroutine returns after doing some parameter checking. If $n > 0$ and *nrhs* is 0, no solutions are computed and the subroutine returns after factoring the matrix.

See references ["9" on page 1253] and ["78" on page 1257].

Error conditions

Resource Errors

None.

Computational Errors

Matrix ***A*** is singular.

- The factorization completed but the block diagonal matrix ***D*** is exactly singular. *nsinfo* is set to *i*, where d_{ii} is exactly zero. This diagonal element is identified in the computational error message.
- The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2147 is set to be

unlimited in the ESSL error option table. For details, see “What Can You Do about ESSL Computational Errors?” on page 65.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. *n* < 0
3. *n* > *ldb*
4. *ldb* ≤ 0
5. *nrhs* < 0
6. *nsinfo* < (minimum value).
   - For the minimum value, see the *nsinfo* argument description.
   - Return code 1 is returned if error 2200 is recoverable.

Examples

Example 1

This example shows how to solve the system ***AX*** = ***B***, for three right-hand sides, where matrix ***A*** is a real symmetric indefinite matrix of order 8, stored in lower-packed storage mode, and ***X*** and ***B*** are real general matrices.

On input, matrix ***A*** is:

```
              |  3.0   5.0  -2.0   2.0   3.0  -5.0  -2.0  -3.0 |
              |  5.0   3.0   2.0  -2.0   5.0  -3.0   2.0  -5.0 |
              | -2.0   2.0   0.0   0.0  -2.0   2.0   0.0   6.0 |
A =           |  2.0  -2.0   0.0   8.0  -6.0 -10.0  -8.0 -14.0 |
              |  3.0   5.0  -2.0  -6.0  12.0   6.0   8.0   6.0 |
              | -5.0  -3.0   2.0 -10.0   6.0  16.0   8.0  20.0 |
              | -2.0   2.0   0.0  -8.0   8.0   8.0   6.0  18.0 |
              | -3.0  -5.0   6.0 -14.0   6.0  20.0  18.0  34.0 |
```

Note: The AP array is formatted in a triangular arrangement for readability; however, it is stored in lower-packed storage mode.

Call Statement and Input:

```
             UPLO  N   NRHS  AP  IPVT  BX  LDB  NSINFO
              |    |    |     |   |     |   |      |
CALL DBSSV ( 'L',  8,   3,   AP, IPVT, BX,  8,    36  )
```

```
AP = ( 3.0,  5.0, -2.0,  2.0,  3.0, -5.0, -2.0, -3.0,
       3.0,  2.0, -2.0,  5.0, -3.0,  2.0, -5.0,
       0.0,  0.0, -2.0,  2.0,  0.0,  6.0,
       8.0, -6.0,-10.0, -8.0,-14.0,
      12.0,  6.0,  8.0,  6.0,
      16.0,  8.0, 20.0,
       6.0, 18.0,
      34.0 )
```

```
        |   1.0  -38.0   47.0 |
        |   7.0  -10.0   73.0 |
        |   6.0   52.0    2.0 |
BX   =  | -30.0 -228.0  -42.0 |
        |  32.0  183.0  105.0 |
        |  34.0  297.0    9.0 |
        |  32.0  244.0   44.0 |
        |  62.0  497.0   61.0 |
```

Output:

```
        |   1.0    1.0    8.0 |
        |   1.0    2.0    7.0 |
        |   1.0    3.0    6.0 |
BX   =  |   1.0    4.0    5.0 |
        |   1.0    5.0    4.0 |
        |   1.0    6.0    3.0 |
        |   1.0    7.0    2.0 |
        |   1.0    8.0    1.0 |
```

NSINFO = 0

Note: AP and IPVT are stored in an internal format.

Example 2

This example shows how to solve the system ***AX*** = ***B***, for three right-hand sides, where matrix ***A*** is a real symmetric indefinite matrix of order 8, stored in upper-packed storage mode, and ***X*** and ***B*** are real general matrices.

On input, matrix ***A*** is:

```
             | 34.0  18.0  17.0   6.0 -14.0   6.0  -5.0  -3.0 |
             | 18.0   6.0   6.0   8.0  -8.0   0.0   2.0  -2.0 |
             | 17.0   6.0   9.0   9.0  -8.0   0.0   2.0  -2.0 |
             |  6.0   8.0   9.0  12.0  -6.0  -2.0   5.0   3.0 |
             |-14.0  -8.0  -8.0  -6.0   8.0   0.0  -2.0   2.0 |
             |  6.0   0.0   0.0  -2.0   0.0   0.0   2.0  -2.0 |
             | -5.0   2.0   2.0   5.0  -2.0   2.0   3.0   5.0 |
             | -3.0  -2.0  -2.0   3.0   2.0  -2.0   5.0   3.0 |
```

Note: The AP array is formatted in a triangular arrangement for readability; however, it is stored in upper-packed storage mode.

Call Statement and Input:

```
             UPLO   N   NRHS  AP  IPVT  BX  LDB  NSINFO
              |     |    |     |   |     |   |      |
CALL DBSSV ( 'U',  8,   3,   AP, IPVT, BX,  8,    36  )
```

```
AP = ( 34.0,
       18.0,  6.0,
       17.0,  6.0,  9.0,
        6.0,  8.0,  9.0, 12.0,
      -14.0, -8.0, -8.0, -6.0,  8.0,
        6.0,  0.0,  0.0, -2.0,  0.0,  0.0,
       -5.0,  2.0,  2.0,  5.0, -2.0,  2.0,  3.0,
       -3.0, -2.0, -2.0,  3.0,  2.0, -2.0,  5.0,  3.0 )
```

```
        |  59.0   52.0  479.0 |
        |  30.0   38.0  232.0 |
        |  33.0   50.0  247.0 |
BX   =  |  35.0  114.0  201.0 |
        | -28.0  -36.0 -216.0 |
        |   4.0   -4.0   40.0 |
        |  12.0   88.0   20.0 |
        |   4.0   56.0  -20.0 |
```

Output:

```
        |   1.0    1.0    8.0 |
        |   1.0    2.0    7.0 |
        |   1.0    3.0    6.0 |
BX   =  |   1.0    4.0    5.0 |
        |   1.0    5.0    4.0 |
        |   1.0    6.0    3.0 |
        |   1.0    7.0    2.0 |
        |   1.0    8.0    1.0 |
```

NSINFO = 0

Note: AP and IPVT are stored in an internal format.

DBSTRF (Symmetric Indefinite Matrix Factorization)

Purpose

DBSTRF factors a real symmetric indefinite matrix ***A***. The matrix ***A***, stored in upper- or lower-packed storage mode, is factored using the Bunch-Kaufman diagonal pivoting method, where ***A*** is expressed as:

$\boldsymbol{A} = \boldsymbol{UDU}^{T}$ or
$\boldsymbol{A} = \boldsymbol{LDL}^{T}$

where:

U is a product of permutation and unit upper triangular matrices.

L is a product of permutation and unit lower triangular matrices.

D is a symmetric block diagonal matrix, consisting of 1 × 1 and 2 × 2 diagonal blocks.

To solve a system of equations with one or more right-hand sides, follow the call to this subroutine with one or more calls to DBSTRS.

Table 151: Data Types.

Data Types		
A	***ipvt***	**Subroutine**
Long-precision real	Integer	DBSTRF

Note: The output from DBSTRF should be used only as input to DBSTRS, for performing a solve.

Syntax

Language	**Syntax**
Fortran	CALL DBSTRF (*uplo*, *n*, *ap*, *ipvt*, *nsinfo*)
C and C++	dbstrf (*uplo*, *n*, *ap*, *ipvt*, *nsinfo*);

On Entry

uplo

indicates whether matrix ***A*** is stored in upper- or lower-packed storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper-packed storage mode.

If *uplo* = 'L', ***A*** is stored in lower-packed storage mode.

Specified as: a single character. It must be 'U' or 'L'.

n

is the order *n* of matrix ***A***.

Specified as: an integer; $n \geq 0$.

ap

is array, referred to as AP, in which matrix ***A***, to be factored, is stored in upper- or lower-packed storage mode.

Specified as: a one-dimensional array of length *nsinfo*, containing numbers of the data type indicated in Table 151 on page 613. See "Notes" on page 615.

ipvt

See On Return.

nsinfo

is the number of elements in array, AP.

If $n \le nco$, $nsinfo = n(n + 1) / 2$

Where:

$nco = \sqrt{2.0(ics)}$

ics is the size in doublewords of the data cache. The data cache size can be obtained by utilizing the following C language code fragment:

```
#include <sys/systemcfg.h>
int ics;
     .
     .
     .
ics=_system_configuration.dcache_size/8;
```

ics is the size in doublewords of the data cache. The data cache size can be obtained by utilizing the following C language code fragment:

```
#include <sys/systemcfg.h>
int ics;
     .
     .
     .
ics=_system_configuration.dcache_size/8;
```

Otherwise, to determine a sufficient amount of storage, use the following processor-independent formula:

$n0 = 100$
$ns = (n + n0)\ (n + n0 + 1) / 2 + n(n0)$

For *uplo* = 'L',
$nsinfo \ge ns$

For *uplo* = 'U',
$n1 = (n + 1) / 2$
$nt = n((n + 1) / 2)$
$nt1 = n1(n1 + 1)$
$ns1 = nt + nt1$
$nsinfo \ge \max(ns, ns1)$

To determine the minimal amount of storage see "Notes" on page 615.

Specified as: an integer; $nsinfo > 0$.

On Return

ap

is the transformed matrix ***A*** of order *n*, containing the results of the factorization.

If $nsinfo \ge 0$ and $n > nco$, additional information that can be used to obtain a minimum *nsinfo* is stored in AP(1).

See "Notes" on page 615 and "Function" on page 615.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 151 on page 613.

ipvt
is an integer vector of length *n*, containing the pivot information necessary to construct the factored form of ***A***.

Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 151 on page 613.

nsinfo
indicates the result of the computation.

- If *nsinfo* = 0, the factorization completed successfully.
- If *nsinfo* > 0, factorization was unsuccessful and *nsinfo* is set to*i* where d_{ii} is exactly zero.
- If *nsinfo* < 0, factorization did not take place and the array AP remains unchanged. |*nsinfo*| is the minimal storage required for factorization to take place. Error message 2200 is issued and execution terminates, unless you have used ERRSET to make error code 2200 recoverable. See "What Can You Do about ESSL Input-Argument Errors?" on page 64.

Specified as: an integer.

Notes

1. This subroutine accepts lowercase letters for the *uplo* argument.
2. In your C program, argument *nsinfo* must be passed by reference.
3. In the input array specified for *ap*, the first *n*(*n*+1)/2 elements are matrix elements. The additional locations, required in the array, are used for working storage.
4. The array specified for *ap* should not be altered between calls to the factorization and solve subroutines; otherwise, unpredictable results may occur.
5. On return, if *nsinfo* ≥ 0 and *n* > *nco*, *ap* contains additional information in AP(1) that can be used to obtain the minimal required *nsinfo*. This information can be accessed using the following code fragment:

```
REAL*8    AP(NSINFO)
INTEGER    API(2)
EQUIVALENCE(API, AP)
          .
          .
          .
NSINFOMIN = API(2)
```

6. For a description of how a symmetric matrix is stored in upper- or lower-packed storage mode in an array, see "Symmetric Matrix" on page 80.

Function

where:

U is a product of permutation and unit upper triangular matrices.

L is a product of permutation and unit lower triangular matrices.

D is a symmetric block diagonal matrix, consisting of 1 × 1 and 2 × 2 diagonal blocks.

If *n* is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See references ["9" on page 1253] and ["78" on page 1257].

Error conditions

Resource Errors
None.

Computational Errors

Matrix ***A*** is singular.

- The factorization completed but the block diagonal matrix ***D*** is exactly singular. *nsinfo* is set to *i*, where d_{ii} is exactly zero. This diagonal element is identified in the computational error message.
- The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2147 is set to be unlimited in the ESSL error option table. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. *n* < 0
3. *nsinfo* < (minimum value).
   - For the minimum value, see the *nsinfo* argument description.
   - Return code 1 is returned if error 2200 is recoverable.

Examples

Example 1

This example shows a factorization of a symmetric indefinite matrix ***A*** of order 8, stored in lower-packed storage mode, where on input matrix ***A*** is:

```
              |  3.0   5.0  -2.0   2.0   3.0  -5.0  -2.0  -3.0 |
              |  5.0   3.0   2.0  -2.0   5.0  -3.0   2.0  -5.0 |
              | -2.0   2.0   0.0   0.0  -2.0   2.0   0.0   6.0 |
              |  2.0  -2.0   0.0   8.0  -6.0 -10.0  -8.0 -14.0 |
              |  3.0   5.0  -2.0  -6.0  12.0   6.0   8.0   6.0 |
              | -5.0  -3.0   2.0 -10.0   6.0  16.0   8.0  20.0 |
              | -2.0   2.0   0.0  -8.0   8.0   8.0   6.0  18.0 |
              | -3.0  -5.0   6.0 -14.0   6.0  20.0  18.0  34.0 |
```

Note: The AP array is formatted in a triangular arrangement for readability; however, it is stored in lower-packed storage mode.

Call Statement and Input:

```
              UPLO   N    AP  IPVT  NSINFO
               |     |    |    |       |
CALL DBSTRF ( 'L',  8,  AP, IPVT,   36  )
```

```
AP = ( 3.0,  5.0, -2.0,  2.0,  3.0, -5.0, -2.0, -3.0,
       3.0,  2.0, -2.0,  5.0, -3.0,  2.0, -5.0,
       0.0,  0.0, -2.0,  2.0,  0.0,  6.0,
       8.0, -6.0,-10.0, -8.0,-14.0,
      12.0,  6.0,  8.0,  6.0,
      16.0,  8.0, 20.0,
       6.0, 18.0,
      34.0 )
```

Output:

NSINFO = 0

Note: AP and IPVT are stored in an internal format and must be passed unchanged to the solve subroutine.

Example 2

This example shows a factorization of a symmetric indefinite matrix ***A*** of order 8, stored in upper-packed storage mode, where on input matrix ***A*** is:

```
              | 34.0  18.0  17.0   6.0 -14.0   6.0  -5.0  -3.0 |
              | 18.0   6.0   6.0   8.0  -8.0   0.0   2.0  -2.0 |
              | 17.0   6.0   9.0   9.0  -8.0   0.0   2.0  -2.0 |
              |  6.0   8.0   9.0  12.0  -6.0  -2.0   5.0   3.0 |
              |-14.0  -8.0  -8.0  -6.0   8.0   0.0  -2.0   2.0 |
              |  6.0   0.0   0.0  -2.0   0.0   0.0   2.0  -2.0 |
              | -5.0   2.0   2.0   5.0  -2.0   2.0   3.0   5.0 |
              | -3.0  -2.0  -2.0   3.0   2.0  -2.0   5.0   3.0 |
```

Note: The AP array is formatted in a triangular arrangement for readability; however, it is stored in upper-packed storage mode.

Call Statement and Input:

```
               UPLO   N    AP  IPVT  NSINFO
                |     |     |    |       |
CALL DBSTRF ( 'U',  8,  AP, IPVT,   36  )
```

```
AP = ( 34.0,
       18.0,  6.0,
       17.0,  6.0,  9.0,
        6.0,  8.0,  9.0, 12.0,
      -14.0, -8.0, -8.0, -6.0,  8.0,
        6.0,  0.0,  0.0, -2.0,  0.0,  0.0,
       -5.0,  2.0,  2.0,  5.0, -2.0,  2.0,  3.0,
       -3.0, -2.0, -2.0,  3.0,  2.0, -2.0,  5.0,  3.0 )
```

Output:

`NSINFO` = 0

Note: AP and `IPVT` are stored in an internal format and must be passed unchanged to the solve subroutine.

DBSTRS (Symmetric Indefinite Matrix Multiple Right-Hand Side Solve)

Purpose

The DBSTRS subroutine solves a system of linear equations ***AX*** = ***B*** for ***X***, where ***A*** is a real symmetric indefinite matrix, and ***X*** and ***B*** are real general matrices. This subroutine uses the results of the factorization of matrix ***A***, produced by a preceding call to DBSTRF.

Table 152: Data Types.

Data Types		
A, ***B***	***ipvt***	**Subroutine**
Long-precision real	Integer	DBSTRS

Note: The input to this solve subroutine must be the output from the factorization subroutine DBSTRF.

Syntax

Language	Syntax
Fortran	CALL DBSTRS (*uplo*, *n*, *nrhs*, *ap*, *ipvt*, *bx*, *ldb*, *info*)
C and C++	dbstrs (*uplo*, *n*, *nrhs*, *ap*, *ipvt*, *bx*, *ldb*, *info*);

On Entry

uplo
indicates whether original matrix ***A*** is stored in upper- or lower-packed storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper-packed storage mode.

If *uplo* = 'L', ***A*** is stored in lower-packed storage mode.

Specified as: a single character. It must be 'U' or 'L'.

n
is the order *n* of factored matrix ***A*** and the number of rows of matrix ***B***.

Specified as: an integer; $n \geq 0$.

nrhs
is the number of right-hand sides; i.e., the number of columns of matrix ***B***.

Specified as: an integer; $nrhs \geq 0$.

ap
is the factored matrix ***A*** produced by a preceding call to DBSTRF.

Specified as: a one-dimensional array of length *nsinfo*, containing numbers of the data type indicated in Table 152 on page 617. See "Notes" on page 619 and "DBSTRF (Symmetric Indefinite Matrix Factorization)" on page 613.

ipvt
is an integer vector of length *n*, containing the pivot information necessary to construct the factored form of ***A***, produced by a preceding call to DBSTRF.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 152 on page 617. See "Notes" on page 619.

bx
is the general matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of matrix ***B***.

Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 152 on page 617.

ldb
is the leading dimension of the array specified for ***B***.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

info
See On Return.

On Return

bx
is the matrix ***X***, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of ***X***.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 152 on page 617.

info
indicates the result of the computation.

- If *info* = 0, the subroutine completed successfully.

Returned as: an integer.

Notes

1. This subroutine accepts lowercase letters for the *uplo* argument.
2. In your C program, argument *info* must be passed by reference.
3. The array data specified for input arguments *ap* and *ipvt* for this subroutine must be the same as the corresponding output arguments for DBSTRF.
4. The scalar data specified for input arguments *uplo* and *n* must be the same as that specified for DBSTRF.
5. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
6. For a description of how a symmetric matrix is stored in upper- or lower-packed storage mode in an array, see "Symmetric Matrix" on page 80.
7. To solve ***AX*** = ***B*** for ***X***, where ***B*** and ***X*** are *n* by *nrhs* matrices, precede the call to DBSTRS with a call to DBSTRF.

Function

The system ***AX*** = ***B*** is solved for ***X***, where ***A*** is a real symmetric indefinite matrix, and ***X*** and ***B*** are real general matrices. This subroutine uses the results of the factorization of matrix ***A***, produced by a preceding call to DBSTRF.

If *n* or *nrhs* is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See references ["9" on page 1253] and ["78" on page 1257].

Error conditions

Resource Errors

None.

Computational Errors

None.

Note: If the factorization performed by DBSTRF failed because matrix ***A*** is singular, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. $n < 0$
3. $nrhs < 0$
4. $n > ldb$
5. $ldb \le 0$

Examples

Example 1

This example shows how to solve the system ***AX*** = ***B***, for three right-hand sides, where matrix ***A*** is the same matrix factored in the Example 1 for DBSTRF.

Call Statement and Input:

```
               UPLO   N    NRHS  AP   IPVT  BX  LDB  INFO
                |     |     |     |    |     |   |     |
CALL DBSTRS ( 'L',   8,    3,    AP, IPVT, BX,  8,  INFO )
```

AP = (for this subroutine must be the same
as the corresponding output argument for DBSTRF.
See Example 1 for DBSTRF.)
IPVT = (for this subroutine must be the same
as the corresponding output argument for DBSTRF.
See Example 1 for DBSTRF.)

```
         |   1.0  -38.0   47.0 |
         |   7.0  -10.0   73.0 |
         |   6.0   52.0    2.0 |
BX    =  | -30.0 -228.0  -42.0 |
         |  32.0  183.0  105.0 |
         |  34.0  297.0    9.0 |
         |  32.0  244.0   44.0 |
         |  62.0  497.0   61.0 |
```

Output:

```
         |   1.0    1.0    8.0 |
         |   1.0    2.0    7.0 |
         |   1.0    3.0    6.0 |
BX    =  |   1.0    4.0    5.0 |
         |   1.0    5.0    4.0 |
         |   1.0    6.0    3.0 |
         |   1.0    7.0    2.0 |
         |   1.0    8.0    1.0 |
```

INFO = 0

Example 2

This example shows how to solve the system ***AX*** = ***B***, for three right-hand sides, where matrix ***A*** is the same matrix factored in the Example 2 for DBSTRF.

Call Statement and Input:

```
              UPLO   N   NRHS  AP  IPVT  BX  LDB  INFO
               |     |    |     |   |     |   |     |
CALL DBSTRS ( 'U',  8,   3,   AP, IPVT, BX,  8,  INFO )
```

AP =(for this subroutine must be the same
as the corresponding output argument for DBSTRF.
See Example 2 for DBSTRF.)
IPVT =(for this subroutine must be the same
as the corresponding output argument for DBSTRF.
See Example 2 for DBSTRF.)

```
         |  59.0   52.0  479.0 |
         |  30.0   38.0  232.0 |
         |  33.0   50.0  247.0 |
BX    =  |  35.0  114.0  201.0 |
         | -28.0  -36.0 -216.0 |
         |   4.0   -4.0   40.0 |
         |  12.0   88.0   20.0 |
         |   4.0   56.0  -20.0 |
```

Output:

```
         |   1.0    1.0    8.0 |
         |   1.0    2.0    7.0 |
         |   1.0    3.0    6.0 |
BX    =  |   1.0    4.0    5.0 |
         |   1.0    5.0    4.0 |
         |   1.0    6.0    3.0 |
         |   1.0    7.0    2.0 |
         |   1.0    8.0    1.0 |
```

INFO = 0

STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)

Purpose

These subroutines find the inverse of triangular matrix ***A***:

$\boldsymbol{A} \leftarrow \boldsymbol{A}^{-1}$

Matrix ***A*** can be either upper or lower triangular, where:

- For STRTRI, DTRTRI, CTRTRI, and ZTRTRI, it is stored in upper- or lower-triangular storage mode.
- For STPTRI, DTPTRI, CTPTRI, and ZTPTRI, it is stored in upper- or lower-triangular-packed storage mode.

Table 153: Data Types.

A	**Subroutine**
Short-precision real	STRTRI^ and STPTRI^
Long-precision real	DTRTRI^ and DTPTRI^
Short-precision complex	CTRTRI^ and CTPTRI^
Long-precision complex	ZTRTRI^ and ZTPTRI^
^LAPACK	

Syntax

Table 154: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL STRTRI | DTRTRI | CTRTRI | ZTRTRI (*uplo*, *diag*, *n*, *a*, *lda*, *info*) CALL STPTRI | DTPTRI | CTPTRI | ZTPTRI (*uplo*, *diag*, *n*, *ap*, *info*)

C and C++	strtri | dtrtri | ctrtri | ztrtri (*uplo*, *diag*, *n*, *a*, *lda*, *info*);
	stptri | dtptri | ctptri | ztptri (*uplo*, *diag*, *n*, *ap*, *info*);

LAPACKE	info = LAPACKE_strtri | LAPACKE_dtrtri | LAPACKE_ctrtri | LAPACKE_ztrtri (*matrix_layout*, *uplo*, *diag*, *n*, *a*, *lda*);
	info = LAPACKE_stptri | LAPACKE_dtptri | LAPACKE_ctptri | LAPACKE_ztptri (*matrix_layout*, *uplo*, *diag*, *n*, *ap*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

uplo
indicates whether matrix ***A*** is an upper or lower triangular matrix, where:

If *uplo* = 'U', ***A*** is an upper triangular matrix.

If *uplo* = 'L', ***A*** is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

a
is the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular storage mode, respectively. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 153 on page 621.

lda
is the leading dimension of the arrays specified for *a*.

Specified as: an integer; $lda > 0$ and $lda \ge n$.

ap
is the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular-packed storage mode, respectively.

Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 153 on page 621.

n
is the order of matrix ***A***.

Specified as: an integer; $n \ge 0$.

info
See On Return.

On Return

a
is the inverse of the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular storage mode, respectively. Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 153 on page 621.

ap
is the inverse of the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular-packed storage mode, respectively. Returned as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 153 on page 621.

info
has the following meaning:

If *info* = 0, the inverse completed successfully.

If *info* > 0, *info* is set equal to the first *i* where $\boldsymbol{A}_{ii}$ is zero. Matrix ***A*** is singular and its inverse could not be computed.

Returned as:

- For STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI, returned as: an integer; info ≥ 0.

- For LAPACKE_strtri, LAPACKE_dtrtri, LAPACKE_ctrtri, LAPACKE_ztrtri, LAPACKE_stptri, LAPACKE_dtptri, LAPACKE_ctptri, and LAPACKE_ztptri, returned as an integer function value; info ≥ 0.

Notes:

1. In C programs, the argument *info* must be passed by reference.
2. These subroutines accept lowercase letters for the *uplo* and *diag* arguments.
3. If matrix ***A*** is upper triangular (*uplo* = 'U'), these subroutines refer to only the upper triangular portion of the matrix. If matrix ***A*** is lower triangular, (*uplo* = 'L'), these subroutines refer to only the lower triangular portion of the matrix. The unreferenced elements are assumed to be zero.
4. The elements of the diagonal of a unit triangular matrix are always one, so you do not need to set these values.
5. The way _TRTRI and _TPTRI subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the *info* argument to provide information about the computational error, but they also provide an error message.
6. On both input and output, matrix ***A*** conforms to LAPACK format.
7. For a description of triangular matrices and how they are stored in upper- and lower-triangular storage mode and in upper- and lower-triangular-packed storage mode, see "Triangular Matrix" on page 88.

Function

These subroutines find the inverse of triangular matrix ***A***, where ***A*** is either upper or lower triangular:

$\boldsymbol{A} \leftarrow \boldsymbol{A}^{-1}$

where:

A is the triangular matrix of order *n*.
$\boldsymbol{A}^{-1}$ is the inverse of the triangular matrix of order *n*.

If *n* is 0, no computation is performed. See references ["9" on page 1253] and ["46" on page 1255].

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
Matrix ***A*** is singular.

- One or more of the diagonal elements of matrix ***A*** are zero. The first column, *i*, of matrix ***A***, in which a zero diagonal element is found, is identified in the computational error message and returned in the argument *info*.
- The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2146 is set to be unlimited in the ESSL error option table.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. *diag* ≠ 'U' or 'N'
3. $n < 0$
4. $lda \le 0$
5. $lda < n$

Examples

Example 1

This example shows how the inverse of matrix ***A*** is computed, where ***A*** is a 5 by 5 upper triangular matrix that is not unit triangular and is stored in upper-triangular storage mode.

Matrix ***A*** is:

```
                    ┌                                 ┐
                    | 1.00  3.00  4.00   5.00   6.00 |
                    | 0.00  2.00  8.00   9.00   1.00 |
                    | 0.00  0.00  4.00   8.00   4.00 |
                    | 0.00  0.00  0.00  -2.00   6.00 |
                    | 0.00  0.00  0.00   0.00  -1.00 |
                    └                                 ┘
```

Matrix ***A***$^{-1}$ is:

```
                  ┌                                     ┐
                  | 1.00  -1.50   2.00   3.75   35.00 |
                  | 0.00   0.50  -1.00  -1.75  -14.00 |
                  | 0.00   0.00   0.25   1.00    7.00 |
                  | 0.00   0.00   0.00  -0.50   -3.00 |
                  | 0.00   0.00   0.00   0.00   -1.00 |
                  └                                     ┘
```

Call Statement and Input:

```
             UPLO  DIAG  N   A  LDA  INFO
              |     |    |   |   |    |
CALL STRTRI( 'U' , 'N' , 5 , A,  5,  INFO )
```

```
        ┌                                 ┐
        | 1.00  3.00  4.00   5.00   6.00 |
        |  .    2.00  8.00   9.00   1.00 |
A    =  |  .     .    4.00   8.00   4.00 |
        |  .     .     .    -2.00   6.00 |
        |  .     .     .      .    -1.00 |
        └                                 ┘
```

Output:

```
        ┌                                     ┐
        | 1.00  -1.50   2.00   3.75   35.00 |
        |  .     0.50  -1.00  -1.75  -14.00 |
A    =  |  .      .     0.25   1.00    7.00 |
        |  .      .      .    -0.50   -3.00 |
        |  .      .      .      .     -1.00 |
        └                                     ┘
```

```
INFO =    0
```

Example 2

This example shows how the inverse of matrix ***A*** is computed, where ***A*** is a 5 by 5 lower triangular matrix that is unit triangular and is stored in lower-triangular storage mode.

Matrix ***A*** is:

```
                       ┌                          ┐
                       | 1.0  0.0  0.0  0.0  0.0 |
                       | 3.0  1.0  0.0  0.0  0.0 |
                       | 4.0  8.0  1.0  0.0  0.0 |
                       | 5.0  9.0  8.0  1.0  0.0 |
                       | 6.0  1.0  4.0  6.0  1.0 |
                       └                          ┘
```

Matrix ***A***$^{-1}$ is:

```
                   ┌                                ┐
                   |    1.0     0.0   0.0   0.0  0.0 |
                    |   -3.0     1.0   0.0   0.0  0.0 |
                    |   20.0    -8.0   1.0   0.0  0.0 |
                    | -138.0    55.0  -8.0   1.0  0.0 |
                    |  745.0  -299.0  44.0  -6.0  1.0 |
                                                     ┘
```

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of one for the diagonal elements.

Call Statement and Input:

```
              UPLO  DIAG  N   A  LDA  INFO
               |     |    |   |   |    |
CALL STRTRI( 'L' , 'U' , 5 , A,  5,  INFO )
```

```
        ┌                       ┐
        |  .    .    .    .   . |
        | 3.0   .    .    .   . |
A    =  | 4.0  8.0   .    .   . |
        | 5.0  9.0  8.0   .   . |
        | 6.0  1.0  4.0  6.0  . |
                               ┘
```

Output:

```
        ┌                               ┐
        |    .       .     .     .   . |
        |   -3.0     .     .     .   . |
A    =  |   20.0    -8.0   .     .   . |
        | -138.0    55.0  -8.0    .   . |
        |  745.0  -299.0  44.0  -6.0  . |
                                       ┘
```

```
INFO =    0
```

Example 3

This example shows how the inverse of matrix ***A*** is computed, where ***A*** is a 5 by 5 upper triangular matrix that is not unit triangular and is stored in upper-triangular storage mode.

Matrix ***A*** is:

```
                                                                              ┐
         | (-4.00, 1.00) ( 4.00, -3.00) (-1.00,  3.00) ( 0.00,  0.00) (-1.00,  0.00) |
         | ( 0.00, 0.00) (-2.00,  0.00) (-3.00, -1.00) (-2.00, -1.00) ( 4.00,  3.00) |
         | ( 0.00, 0.00) ( 0.00,  0.00) (-5.00,  3.00) (-3.00, -3.00) (-5.00, -5.00) |
         | ( 0.00, 0.00) ( 0.00,  0.00) ( 0.00,  0.00) ( 4.00, -4.00) ( 2.00,  0.00) |
         | ( 0.00, 0.00) ( 0.00,  0.00) ( 0.00,  0.00) ( 0.00,  0.00) ( 2.00, -1.00) |
                                                                              ┘
```

Matrix A^{-1} is:

```
                                                                              ┐
        | (-0.24, -0.06) (-0.56, 0.24) ( 0.41,  0.09) (-0.22,  0.13) ( 1.32,  2.12) |
        | ( 0.00,  0.00) (-0.50, 0.00) ( 0.18,  0.21) (-0.22, -0.06) ( 0.21,  1.87) |
        | ( 0.00,  0.00) ( 0.00, 0.00) (-0.15, -0.09) ( 0.07, -0.11) ( 0.02, -0.47) |
        | ( 0.00,  0.00) ( 0.00, 0.00) ( 0.00,  0.00) ( 0.12,  0.12) (-0.05, -0.15) |
        | ( 0.00,  0.00) ( 0.00, 0.00) ( 0.00,  0.00) ( 0.00,  0.00) ( 0.40,  0.20) |
                                                                              ┘
```

Call Statement and Input:

```
             UPLO  DIAG  N   A  LDA  INFO
              |     |    |   |   |    |
CALL ZTRTRI( 'U' , 'N' , 5 , A,  5,  INFO )
```

```
                                                                                ┐
        | (-4.00, 1.00) ( 4.00, -3.00) (-1.00,  3.00) ( 0.00,  0.00) (-1.00,  0.00) |
        |       .       (-2.00,  0.00) (-3.00, -1.00) (-2.00, -1.00) ( 4.00,  3.00) |
A    =  |       .             .        (-5.00,  3.00) (-3.00, -3.00) (-5.00, -5.00) |
        |       .             .              .        ( 4.00, -4.00) ( 2.00,  0.00) |
        |       .             .              .              .        ( 2.00, -1.00) |
                                                                                ┘
```

Output:

```
                                                                                ┐
        | (-0.24, -0.06) (-0.56, 0.24) ( 0.41,  0.09) (-0.22,  0.13) ( 1.32,  2.12) |
        |       .        (-0.50, 0.00) ( 0.18,  0.21) (-0.22, -0.06) ( 0.21,  1.87) |
A    =  |       .              .       (-0.15, -0.09) ( 0.07, -0.11) ( 0.02, -0.47) |
        |       .              .              .       ( 0.12,  0.12) (-0.05, -0.15) |
        |       .              .              .              .       ( 0.40,  0.20) |
                                                                                ┘
```

```
INFO =    0
```

Example 4

This example shows how the inverse of matrix ***A*** is computed, where ***A*** is a 5 by 5 lower triangular matrix that is unit triangular and is stored in lower-triangular storage mode.

Matrix ***A*** is:

```
                                                                               ┐
        | ( 1.00,  0.00) ( 0.00,  0.00) ( 0.00,  0.00) ( 0.00, 0.00) ( 0.00, 0.00) |
        | ( 4.00, -3.00) ( 1.00,  0.00) ( 0.00,  0.00) ( 0.00, 0.00) ( 0.00, 0.00) |
        | (-1.00,  3.00) (-3.00, -1.00) ( 1.00,  0.00) ( 0.00, 0.00) ( 0.00, 0.00) |
        | ( 0.00,  0.00) (-2.00, -1.00) (-3.00, -3.00) ( 1.00, 0.00) ( 0.00, 0.00) |
        | (-1.00,  0.00) ( 4.00,  3.00) (-5.00, -5.00) ( 2.00, 0.00) ( 1.00, 0.00) |
                                                                               ┘
```

Matrix ***A***$^{-1}$ is:

```
                                                                                  ┐
        | (  1.00,   0.00) (  0.00,  0.00) ( 0.00,  0.00) ( 0.00, 0.00) ( 0.00, 0.00) |
        | ( -4.00,   3.00) (  1.00,  0.00) ( 0.00,  0.00) ( 0.00, 0.00) ( 0.00, 0.00) |
        | (-14.00,   2.00) (  3.00,  1.00) ( 1.00,  0.00) ( 0.00, 0.00) ( 0.00, 0.00) |
        | (-59.00, -34.00) (  8.00, 13.00) ( 3.00,  3.00) ( 1.00, 0.00) ( 0.00, 0.00) |
        | ( 64.00,   8.00) (-10.00, -9.00) (-1.00, -1.00) (-2.00, 0.00) ( 1.00, 0.00) |
                                                                                  ┘
```

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of one for the diagonal elements.

Call Statement and Input:

```
             UPLO  DIAG  N   A  LDA  INFO
              |     |    |   |   |    |
CALL ZTRTRI( 'L' , 'U' , 5 , A,  5,  INFO )
```

```
                                                                                ┐
        |       .              .              .              .              .     |
        | ( 4.00, -3.00)       .              .              .              .     |
        | (-1.00,  3.00) (-3.00,  1.00)       .              .              .     |
A    =  | ( 0.00,  0.00) (-2.00, -1.00) (-3.00, -3.00)       .              .     |
        | (-1.00,  0.00) ( 4.00,  3.00) (-5.00, -5.00) (2.00, 0.00)         .     |
                                                                                ┘
```

Output:

```
                                                                              ┐
        |        .                 .              .              .            .      |
        | ( -4.00,   3.00)         .              .              .            .      |
        | (-14.00,   2.00) (  3.00,  1.00)        .              .            .      |
A    =  | (-59.00, -34.00) (  8.00, 13.00) ( 3.00,  3.00)        .            .      |
        | ( 64.00,   8.00) (-10.00, -9.00) (-1.00, -1.00) (-2.00, 0.00)       .      |
                                                                              ┘
```

```
INFO =    0
```

Example 5

This example shows how the inverse of matrix ***A*** is computed, where ***A*** is the same matrix shown in Example 1 and is stored in upper-triangular-packed storage mode. The inverse matrix computed here is the same as the inverse matrix shown in Example 1 and is stored in upper-triangular-packed storage mode.

Call Statement and Input:

```
             UPLO  DIAG  N   AP  INFO
              |     |    |   |    |
CALL STPTRI( 'U' , 'N' , 5 , AP, INFO )
```

```
AP   =  (1.00, 3.00, 2.00, 4.00, 8.00, 4.00, 5.00, 9.00, 8.00,
         -2.00, 6.00, 1.00, 4.00, 6.00, -1.00)
```

Output:

```
AP   =  (1.00, -1.50, 0.50, 2.00, -1.00, 0.25, 3.75, -1.75, 1.00,
         -0.50, 35.00, -14.00, 7.00, -3.00, -1.00)
```

```
INFO =   0
```

Example 6

This example shows how the inverse of matrix ***A*** is computed, where ***A*** is the same matrix shown in Example 2 and is stored in lower-triangular-packed storage mode. The inverse matrix computed here is the same as the inverse matrix shown in Example 2 and is stored in lower-triangular-packed storage mode.

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of one for the diagonal elements.

Call Statement and Input:

```
             UPLO  DIAG  N   AP  INFO
              |     |    |   |    |
CALL STPTRI( 'L' , 'U' , N , AP, INFO )
```

```
AP   =  ( . , 3.0, 4.0, 5.0, 6.0, . , 8.0, 9.0, 1.0, . , 8.0, 4.0,
         . , 6.0, . )
```

Output:

```
AP   =  ( . , -3.0, 20.0, -138.0, 745.0, . , -8.0, 55.0, -299.0,
          . , -8.0, 44.0, . , -6.0, . )
```

```
INFO = 0
```

Example 7

This example shows how the inverse of matrix ***A*** is computed, where ***A*** is the same matrix shown in Example 3 and is stored in upper-triangular-packed storage mode. The inverse matrix computed here is the same as the inverse matrix shown in Example 3 and is stored in upper-triangular-packed storage mode.

Call Statement and Input:

```
             UPLO  DIAG  N   AP  INFO
              |     |    |   |    |
CALL ZTPTRI( 'U' , 'N' , 5 , AP, INFO )
```

```
AP   =  ( (-4.00,  1.00),
          ( 4.00, -3.00), (-2.00,  0.00),
          (-1.00,  3.00), (-3.00, -1.00), (-5.00,  3.00),
          ( 0.00,  0.00), (-2.00, -1.00), (-3.00, -3.00), (4.00, -4.00),
          (-1.00,  0.00), ( 4.00,  3.00), (-5.00, -5.00), (2.00,  0.00), (2.00, -1.00) )
```

Output:

```
AP   =  ( (-0.24, -0.06),
          (-0.56,  0.24), (-0.50,  0.00),
          ( 0.41,  0.09), ( 0.18,  0.21), (-0.15, -0.09),
          (-0.22,  0.13), (-0.22, -0.06), ( 0.07, -0.11), ( 0.12,  0.12),
          ( 1.32,  2.12), ( 0.21,  1.87), ( 0.02, -0.47), (-0.05, -0.15), (0.40, 0.20) )
```

```
INFO =    0
```

Example 8

This example shows how the inverse of matrix ***A*** is computed, where ***A*** is the same matrix shown in Example 4 and is stored in lower-triangular-packed storage mode. The inverse matrix computed here is the same as the inverse matrix shown in Example 4 and is stored in lower-triangular-packed storage mode.

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of one for the diagonal elements.

Call Statement and Input:

```
             UPLO  DIAG  N   AP  INFO
              |     |    |   |    |
CALL ZTPTRI( 'L' , 'U' , 5 , AP, INFO )
```

```
AP   =  ( ., ( 4.00, -3.00), (-1.00,  3.00), (0.00, 0.00), (-1.00, 0.00),
          ., (-3.00, -1.00), (-2.00, -1.00), (4.00, 3.00),
          ., (-3.00, -3.00), (-5.00, -5.00),
          ., ( 2.00,  0.00),
          .  )
```

Output:

```
AP   =  ( ., (-4.00, 3.00), (-14.00,  2.00), (-59.00, -34.00), (64.00, 8.00),
          ., ( 3.00, 1.00), (  8.00, 13.00), (-10.00,  -9.00),
          ., ( 3.00, 3.00), ( -1.00, -1.00),
          ., (-2.00, 0.00),
          . )
```

```
INFO =    0
```

SLANTR, DLANTR, CLANTR, ZLANTR, SLANTP, DLANTP, CLANTP, and ZLANTP (Trapezoidal or Triangular Matrix Norm)

Purpose

These subprograms compute the norm of matrix ***A*** as explained below:

SLANTR, DLANTR, CLANTR, and ZLANTR

These subprograms compute the norm of trapezoidal matrix ***A*** stored in upper- or lower-trapezoidal storage mode.

SLANTP, DLANTP, CLANTP, and ZLANTP

These subroutines compute the norm of triangular matrix ***A***, stored in upper- or lower-triangular-packed storage mode.

Table 155: Data Types.

A	***work*, Result**	**Subprogram**
Short-precision real	Short-precision real	SLANTR$^{\Delta}$, SLANTP$^{\Delta}$
Long-precision real	Long-precision real	DLANTR$^{\Delta}$, DLANTP$^{\Delta}$
Short-precision complex	Short-precision real	CLANTR$^{\Delta}$, CLANTP$^{\Delta}$
Long-precision complex	Long-precision real	ZLANTR$^{\Delta}$, ZLANTP$^{\Delta}$
$^{\Delta}$LAPACK		

Syntax

Table 156: Syntax for supported computing languages.

Language	**Syntax**
Fortran	SLANTR | DLANTR | CLANTR | ZLANTR (*norm*, *uplo*, *diag*, *m*, *n*, *a*, *lda*, *work*) SLANTP | DLANTP | CLANTP | ZLANTP (*norm*, *uplo*, *diag*, *n*, *ap*, *work*)

C and C++	slantr | dlantr | clantr | zlantr (*norm*, *uplo*, *diag*, *m*, *n*, *a*, *lda*, *work*);
	slantp | dlantp | clantp | zlantp (*norm*, *uplo*, *diag*, *n*, *ap*, *work*);

LAPACKE	LAPACKE_slantr | LAPACKE_dlantr | LAPACKE_clantr | LAPACKE_zlantr (*matrix_layout*, *norm*, *uplo*, *diag*, *m*, *n*, *a*, *lda*):
	LAPACKE_slantp | LAPACKE_dlantp | LAPACKE_clantp | LAPACKE_zlantp (*matrix_layout*, *norm*, *uplo*, *diag*, *n*, *ap*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

norm

specifies the type of computation, where:

If *norm* = 'O' or '1', the one norm of ***A*** is computed.

If *norm* = 'I', the infinity norm of ***A*** is computed.

If *norm* = 'F' or 'E', the Frobenius or Euclidean norm of ***A*** is computed.

If *norm* = 'M', the absolute value of the matrix element having the largest absolute value, i.e., max (|***A***|), is returned.

Specified as: a single character; *norm* = 'O', '1', 'I', 'F', 'E', or 'M'.

uplo
indicates the storage mode used for matrix ***A***, where:

For SLANTR, DLANTR, CLANTR, and ZLANTR
If *uplo*='U', ***A*** is stored in upper-trapezoidal storage mode.

If *uplo*='L', ***A*** is stored in lower-trapezoidal storage mode.

For SLANTP, DLANTP, CLANTP, and ZLANTP
If *uplo*='U', ***A*** is stored in upper-triangular-packed storage mode.

If *uplo*='L', ***A*** is stored in lower-triangular-packed storage mode.

Specified as: a single character. It must be 'U' or 'L'.

diag
indicates the characteristics of the diagonal of matrix ***A***, where:

For SLANTR, DLANTR, CLANTR, and ZLANTR
If *diag* = 'U', ***A*** is a unit trapezoidal matrix.

If *diag* = 'N', ***A*** is not a unit trapezoidal matrix.

For SLANTP, DLANTP, CLANTP, and ZLANTP
If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

m
is the number of rows in trapezoidal matrix ***A***.

Specified as: an integer; $m \geq 0$.

n

For SLANTR, DLANTR, CLANTR, and ZLANTR
n is the number of columns in matrix ***A***.

For SLANTP, DLANTP, CLANTP, and ZLANTP
n is the order of matrix ***A***

Specified as: an integer; $n \geq 0$.

ap
is the matrix ***A*** of order *n*, stored in upper- or lower-triangular-packed storage mode.

Specified as: a one-dimensional array of (at least) $n(n+1)/2$, containing numbers of the data type indicated in Table 155 on page 629.

a
is the trapezoidal matrix ***A***, stored in upper- or lower-trapezoidal storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 155 on page 629.

lda
is the leading dimension of matrix ***A***.

Specified as: an integer; $lda \geq m$.

work
is the work area used by this subroutine, where:

- When *norm* = 'I', '1', or 'O', *work* is (at least) of length:
  - *m* for SLANTR, DLANTR, CLANTR, and ZLANTR
  - *n* for SLANTP, DLANTP, CLANTP, and ZLANTP

- Otherwise, *work* is not referenced.

Specified as: an area of storage containing numbers of data type indicated in Table 155 on page 629.

On Return

Function value

is the result of the norm computation, returned as a number of the data type indicated in Table 155 on page 629.

If *norm* = 'O' or '1', the one norm of ***A*** is returned.

If *norm* = 'I', the infinity norm of ***A*** is returned.

If *norm* = 'F' or 'E', the Frobenius or Euclidean norm of ***A*** is returned.

If *norm* = 'M', the absolute value of the matrix element having the largest absolute value, i.e., max (|***A***|), is returned.

If *m* = 0 or *n* = 0, the function returns zero.

Notes:

1. Declare this function in your program as returning a value of the data type indicated in Table 155 on page 629.
2. This function accepts lowercase letters for the *norm*, *uplo*, and *diag* arguments.
3. For a description of triangular matrices and how they are stored in upper- and lower-triangular-packed storage mode, see "Triangular Matrix" on page 88.
4. For a description of trapezoidal matrices and how they are stored in upper- and lower-trapezoidal storage mode, see "Trapezoidal Matrix" on page 91.
5. For SLANTR, DLANTR, CLANTR, and ZLANTR, the following cases are extensions to the LAPACK standard:
   - *uplo* = 'U' and *m* > *n*
   - *uplo* = 'L' and *n* > *m*

Function

One of the following computations is performed on matrix ***A***, depending on the value specified for *norm*:

Value specified for *norm*	**Type of computation performed**
'O' or '1'	one norm
'I'	infinity norm
'F' or 'E'	Frobenius or Euclidean norm
'M'	absolute value of the matrix element having the largest absolute value, i.e., max (|***A***|)

If *m* = 0 or *n* = 0, the function returns zero.

Error conditions

Resource Errors

None.

Computational Errors

None.

Input-Argument Errors

1. *norm* ≠ 'O', '1', 'I', 'F', 'E', or 'M'
2. *uplo* ≠ 'U' or 'L'

3. *diag* ≠ 'U' or 'N'
4. *m* < 0
5. *n* < 0
6. *lda* < 1
7. *lda* < *m*

Examples

Example 1

This example computes the infinity norm of real trapezoidal matrix ***A*** stored in lower-trapezoidal storage mode.

Call Statements and Input:

```
                 NORM  UPLO  DIAG  M   N   A   LDA  WORK
                   |     |     |   |   |   |    |     |
ANORM = DLANTR(  'I',  'L', 'N', 10 , 9 , A , 10 , WORK )
```

```
                                                        ┐
        | 1.0   .    .    .    .    .    .    .    .  |
        | 1.0  2.0   .    .    .    .    .    .    .  |
        | 1.0  2.0  3.0   .    .    .    .    .    .  |
        | 1.0  2.0  3.0  4.0   .    .    .    .    .  |
A    =  | 1.0  2.0  3.0  4.0  5.0   .    .    .    .  |
        | 1.0  2.0  3.0  4.0  5.0  6.0   .    .    .  |
        | 1.0  2.0  3.0  4.0  5.0  6.0  7.0   .    .  |
        | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0   .  |
        | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 |
        | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 |
                                                     ┘
```

Output:

ANORM = 45.0

Example 2

This example computes the Frobenius norm of real trapezoidal matrix ***A*** stored upper-trapezoidal storage mode.

Call Statements and Input:

```
                 NORM  UPLO  DIAG  M    N   A   LDA  WORK
                   |     |     |   |    |   |    |     |
ANORM = DLANTR(  'F',  'U', 'U',  9 , 10 , A ,  9 , WORK )
```

```
                                                                ┐
        |    1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  |
        |     .   1.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  |
        |     .    .   1.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  |
        |     .    .    .   1.0  4.0  4.0  4.0  4.0  4.0  4.0  |
A    =  |     .    .    .    .   1.0  5.0  5.0  5.0  5.0  5.0  |
        |     .    .    .    .    .   1.0  6.0  6.0  6.0  6.0  |
        |     .    .    .    .    .    .   1.0  7.0  7.0  7.0  |
        |     .    .    .    .    .    .    .   1.0  8.0  8.0  |
        |     .    .    .    .    .    .    .    .   1.0  9.0  |

┘
```

Output:

ANORM = 28.88

Example 3

This example computes the infinity norm of complex trapezoidal matrix ***A*** stored in lower-trapezoidal storage mode.

Call Statements and Input:

```
                 NORM  UPLO  DIAG  M   N   A   LDA  WORK
                   |     |    |    |   |   |    |     |
ANORM = ZLANTR(  'I',  'L', 'N',  5 , 4 , A ,  5 , WORK )
```

```
                                                      ┐
        | (1.0, 1.0)     .          .         .       |
        | (2.0, 1.0)(2.0, 2.0)      .         .       |
A    =  | (3.0, 1.0)(3.0, 2.0)(3.0, 3.0)      .       |
        | (4.0, 1.0)(4.0, 2.0)(4.0, 3.0)(4.0, 4.0) |
        | (5.0, 1.0)(5.0, 2.0)(5.0, 3.0)(5.0, 4.0) |
                                                      ┘
```

Output:

ANORM = 22.7

Example 4

This example computes the absolute value of the matrix element having the largest absolute value of complex trapezoidal matrix ***A*** stored in upper-trapezoidal storage mode.

Call Statements and Input:

```
                 NORM  UPLO  DIAG  M   N   A   LDA  WORK
                   |     |    |    |   |   |    |     |
ANORM = ZLANTR(  'M',  'U', 'U',  4 , 5 , A ,  4 , WORK )
```

```
                                                              ┐
        | (1.0, 0.0)(1.0, 2.0)(1.0, 3.0)(1.0, 4.0)(1.0, 5.0) |
        |     .     (1.0, 0.0)(2.0, 3.0)(2.0, 4.0)(2.0, 5.0) |
 A   =  |     .         .     (1.0, 0.0)(3.0, 4.0)(3.0, 5.0) |
        |     .         .         .     (1.0, 0.0)(4.0, 5.0) |
                                                              ┘
```

Output:

ANORM = 6.40

Example 5

This example computes the infinity norm of real triangular matrix ***A***, stored in lower-triangular-packed storage mode.

Call Statements and Input:

```
                 NORM  UPLO  DIAG  N   AP   WORK
                   |     |    |    |    |     |
ANORM = DLANTP(  'I',  'L', 'N',  9 , AP , WORK )
```

```
                                                        ┐
        | 1.0   .    .    .    .    .    .    .    .  |
        | 1.0  2.0   .    .    .    .    .    .    .  |
        | 1.0  2.0  3.0   .    .    .    .    .    .  |
        | 1.0  2.0  3.0  4.0   .    .    .    .    .  |
AP   =  | 1.0  2.0  3.0  4.0  5.0   .    .    .    .  |
        | 1.0  2.0  3.0  4.0  5.0  6.0   .    .    .  |
        | 1.0  2.0  3.0  4.0  5.0  6.0  7.0   .    .  |
        | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0   .  |
        | 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 |
                                                     ┘
```

Output:

ANORM = 45.0

Example 6

This example computes the Frobenius norm of real triangular matrix ***A***, stored in upper-triangular-packed storage mode.

Call Statements and Input:

```
                NORM  UPLO  DIAG  N   AP   WORK
                  |     |    |    |    |     |
ANORM = DLANTP(  'F',  'U', 'U',  9 , AP , WORK )
```

```
                                                         ┐
        |    1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  |
        |     .   1.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  |
        |     .    .   1.0  3.0  3.0  3.0  3.0  3.0  3.0  |
        |     .    .    .   1.0  4.0  4.0  4.0  4.0  4.0  |
AP   =  |     .    .    .    .   1.0  5.0  5.0  5.0  5.0  |
        |     .    .    .    .    .   1.0  6.0  6.0  6.0  |
        |     .    .    .    .    .    .   1.0  7.0  7.0  |
        |     .    .    .    .    .    .    .   1.0  8.0  |
        |     .    .    .    .    .    .    .    .   1.0  |

┘
```

Output:

ANORM = 28.72

Example 7

This example computes the infinity norm of complex triangular matrix ***A***, stored in lower-triangular-packed storage mode.

Call Statements and Input:

```
                NORM  UPLO  DIAG  N   AP   WORK
                  |     |    |    |    |     |
ANORM = ZLANTP(  'I',  'L', 'N',  5 , AP , WORK )
```

```
                                                            ┐
        | (1.0, 1.0)    .         .         .         .      |
        | (1.0, 1.0)(2.0, 2.0)    .         .         .      |
AP   =  | (1.0, 1.0)(2.0, 2.0)(3.0, 3.0)    .         .      |
        | (1.0, 1.0)(2.0, 2.0)(3.0, 3.0)(4.0, 4.0)    .      |
        | (1.0, 1.0)(2.0, 2.0)(3.0, 3.0)(4.0, 4.0)(5.0, 5.0) |
                                                            ┘
```

Output:

ANORM = 21.2

Example 8

This example computes the absolute value of the matrix element having the largest absolute value of complex triangular matrix ***A***, stored in upper-triangular-packed storage mode.

Call Statements and Input:

```
                 NORM  UPLO  DIAG  N   AP   WORK
                   |     |    |    |    |     |
ANORM = ZLANTP(  'M',  'U', 'U',  5 , AP , WORK )
```

```
                                                      ┐
        | (1.0, 0.0)(1.0, 1.0)(1.0, 1.0)(1.0, 1.0)(1.0, 1.0) |
        |     .     (1.0, 0.0)(2.0, 2.0)(2.0, 2.0)(2.0, 2.0) |
 AP  =  |     .         .     (1.0, 0.0)(3.0, 3.0)(3.0, 3.0) |
        |     .         .         .     (1.0, 0.0)(4.0, 4.0) |
        |     .         .         .         .     (1.0, 0.0) |
                                                      ┘
```

Output:

ANORM = 5.65

Banded Linear Algebraic Equation Subroutines

This contains the banded linear algebraic equation subroutine descriptions.

SGBSV, DGBSV, CGBSV, and ZGBSV (General Band Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the general band system of linear equations ***AX***=***B*** for ***X***, where ***A*** is a general band matrix and ***B*** and ***X*** are general matrices.

The matrix ***A*** is stored in BLAS-general-band storage mode and is factored using Gaussian elimination with partial pivoting.

Table 157: Data Types.

A, B	**Subroutine**
Short-precision real	SGBSV$^{\Delta}$
Long-precision real	DGBSV$^{\Delta}$
Short-precision complex	CGBSV$^{\Delta}$
Long-precision complex	ZGBSV$^{\Delta}$
$^{\Delta}$LAPACK	

Syntax

Table 158: Syntax for supported computing languages.

Language	Syntax
Fortran	CALL SGBSV | DGBSV | CGBSV | ZGBSV (*n*, *kl*, *ku*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*, *info*)
C and C++	sgbsv | dgbsv | cgbsv | zgbsv (*n*, *kl*, *ku*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*, *info*);
LAPACKE	info = LAPACKE_sgbsv | LAPACKE_dgbsv | LAPACKE_cgbsv | LAPACKE_zgbsv (*matrix_layout*, *n*, *kl*, *ku*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

n

is the order of matrix ***A*** and the number of rows in matrix ***B***.

Specified as: an integer; $n \geq 0$.

kl

is the lower band width *kl* of the matrix ***A***.

Specified as: an integer; $kl \geq 0$.

ku

is the upper band width *ku* of the matrix ***A***.

Specified as: an integer; $ku \geq 0$.

nrhs

is the number of right-hand sides; that is, the number of columns of matrix ***B***.

Specified as: an integer; $nrhs \geq 0$.

a

is the general band matrix ***A*** of order *n*.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 157 on page 635.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and $lda \geq 2kl+ku+1$.

ipiv

See "On Return".

b

is the general matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of matrix ***B***.

Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 157 on page 635.

ldb

is the leading dimension of the array specified for ***B***.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

info
See "On Return".

On Return

a
is the transformed matrix ***A*** of order *n* containing the results of the factorization. See "Function" on page 637.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 157 on page 635.

ipiv
is the integer vector of length *n*, containing the pivot indices.

Returned as: a one-dimensional integer array of (at least) length *n*, containing integers; $1 \le ipiv_j \le n$ for all *j*.

b
If *info* = 0, *b* is the general matrix ***X***, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of matrix ***X***.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 157 on page 635.

info
has the following meaning:

If *info* = 0, the subroutine completed successfully.

If *info* > 0, *info* is set to the first *i*, where $\boldsymbol{U}_{ii}$ is zero. The solution has not been computed.

Returned as:

- For SGBSV, DGBSV, CGBSV, and ZGBSV, returned as: an integer; info ≥ 0.
- For LAPACKE_sgbsv, LAPACKE_dgbsv, LAPACKE_cgbsv, and LAPACKE_zgbsv, returned as an integer function value; info ≥ 0.

Notes:

1. In your C program, argument *info* must be passed by reference.
2. *a*, *ipiv*, and *b* must have no common elements; otherwise, results are unpredictable.
3. For a description of how a general band matrix is stored in BLAS-general-band storage mode in an array, see "General Band Matrix" on page 94.
4. The way these subroutines handle singularity differs from LAPACK. Like LAPACK, these subroutines use the info argument to provide information about the singularity of A, but they also provide an error message.

Function

These subroutines solve the general band system of linear equations ***AX*** = ***B***, where ***A*** is a general band matrix and ***B*** and ***X*** are general matrices.

If *n* is 0, no computation is performed and the subroutine returns after doing some parameter checking. If *n* > 0 and *nrhs* is 0, no solutions are computed and the subroutine returns after factoring the matrix.

See references ["9" on page 1253] and ["48" on page 1255].

Error conditions

Resource Errors
None

Computational Errors
Matrix ***A*** is singular or nearly singular.

- The first column, *i*, of ***L*** with a corresponding $\boldsymbol{U}_{ii}$ = 0 diagonal element is identified in the computational error message.
- The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2146 is set to be unlimited in the ESSL error option table.

Input-Argument Errors

1. $n < 0$
2. $kl < 0$
3. $ku < 0$
4. $nrhs < 0$
5. $lda \leq 0$
6. $2kl+ku+1 > lda$
7. $ldb \leq 0$
8. $n > ldb$

Examples

Example 1

This example shows how to solve the real general band system ***AX***=***B***, where:

Matrix ***A*** is the same used as input in Example 1 for DGBTRF.

Matrix ***B*** is the same used as input in Example 1 for DGBTRS.

Call Statement and Input:

```
             N  KL KU NRHS  A  LDA  IPIV   B  LDB  INFO)
             |   |  |   |   |   |     |    |   |     |
CALL DGBSV(  9 , 2, 3 , 3 , A , 8 , IPIV , B , 9 , INFO)
```

A = (same as input A in Example 1)
B = (same as input B in Example 1)

Output:

```
    |  .      .     .     .      .     4.000  4.000  4.000  4.000 |
    |  .      .     .     .     3.000  3.000  3.000  3.000 -2.271 |
    |  .      .     .    2.000  2.000  2.000  2.000 -4.074 -1.747 |
    |  .      .    1.000 1.000  1.000  1.000 -4.691 -4.177  1.000 |
A = |  .     2.000 2.000 2.000  2.000 -4.419 -3.174  2.000  0.927 |
    | 3.000  3.000 3.000 3.000 -3.617 -5.095  3.000 -1.546  3.037 |
    | 0.666  0.444 0.518 0.567 -0.334  0.326  0.043 -0.790   .    |
    | 0.333 -0.111 0.592 0.246 -0.829 -0.588 -0.617   .      .    |


IPIV     =  ( 3, 4, 5, 6, 5, 6, 9, 8, 9 )
```

```
    |  0.629   1.149   4.713 |
    | -0.025   1.870  -0.726 |
    |  0.523   0.615   2.946 |
    | -0.286  -1.434  -2.774 |
B = | -0.104  -0.524  -1.067 |
    | -0.118  -0.591  -0.970 |
    |  0.220  -0.896   2.027 |
    | -0.079   1.604  -0.340 |
    |  0.496   0.480   3.599 |
 INFO = 0
```

Example 2

This example shows how to solve the complex general band system ***AX***=***B***, where:

Matrix ***A*** is the same used as input in Example 2 for ZGBTRF.

Matrix ***B*** is the same used as input in Example 3 for ZGBTRS.

Call Statement and Input:

```
             N  KL  KU NRHS  A  LDA  IPIV   B  LDB  INFO)
             |   |   |   |   |   |     |    |   |     |
CALL ZGBSV(  5 , 1 , 2 , 3 , A , 5 , IPIV , B , 5 , INFO)
```

A = (same as input A in Example 2)
B = (same as input B in Example 3)

Output:

```
      |        .                 .                 .         ( 2.000, 4.000) ( 3.000,
5.000)  |
      |        .                 .         ( 2.000, 3.000) ( 3.000, 4.000) ( 4.000,
5.000)  |
A =   |        .         ( 0.200, 0.200) ( 0.300, 0.300) ( 0.400, 0.400) ( 0.500,
0.500)  |
      |( 2.000, 1.000) ( 3.000, 2.000) ( 4.000, 3.000) ( 5.000, 4.000)
(-0.339,-6.596)  |
      |( 0.060, 0.020) ( 0.534, 0.305) ( 0.443, 0.300)
(-0.412,-0.396)         .          |

└
┘
IPIV     =  ( 2, 3, 4, 5, 5 )
```

```
     |(-0.039,-0.250)( 0.237,-0.265)(-0.247,-0.723)  |
     |( 0.161,-0.014)( 0.158, 0.163)(-0.092,-0.192)  |
 B = |( 0.205, 0.105)( 0.069, 0.317)( 0.248, 0.130)  |
     |( 0.116, 0.109)(-0.015, 0.224)( 0.314, 0.146)  |
     |(-0.055, 0.038)(-0.090,-0.027)( 0.032,-0.001)  |


 INFO = 0
```

SGBTRF, DGBTRF, CGBTRF and ZGBTRF (General Band Matrix Factorization)

Purpose

These subroutines factor general band matrix ***A***, stored in BLAS-general-band storage mode, using Gaussian elimination with partial pivoting.

To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGBTRS, DGBTRS, CGBTRS, and ZGBTRS respectively.

Table 159: Data Types.	
A	**Subroutine**
Short-precision real	SGBTRF$^{\Delta}$
Long-precision real	DGBTRF$^{\Delta}$
Short-precision complex	CGBTRF$^{\Delta}$
Long-precision complex	ZGBTRF$^{\Delta}$
$^{\Delta}$LAPACK	

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SGBTRS, DGBTRS, CGBTRS, and ZGBTRS respectively.

Syntax

Table 160: Syntax for supported computing languages.	
Language	**Syntax**
Fortran	CALL SGBTRF | DGBTRF | CGBTRF | ZGBTRF (*m*, *n*, *kl*, *ku*, *a*, *lda*, *ipiv*, *info*)
C and C++	sgbtrf | dgbtrf | cgbtrf | zgbtrf (*m*, *n*, *kl*, *ku*, *a*, *lda*, *ipiv*, *info*);
LAPACKE	info = LAPACKE_sgbtrf | LAPACKE_dgbtrf | LAPACKE_cgbtrf | LAPACKE_zgbtrf (*matrix_layout*, *m*, *n*, *kl*, *ku*, *a*, *lda*, *ipiv*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

m

is the number of rows in matrix ***A.***

Specified as: an integer; $m \geq 0$.

n

is the number of columns in matrix ***A.***

Specified as: an integer; $n \geq 0$.

kl

is the lower band width *kl* of the matrix ***A***.

Specified as: an integer; $kl \geq 0$.

ku

is the upper band width *ku* of the matrix ***A***.

Specified as: an integer; $ku \geq 0$.

a

is the *m* by *n* general band matrix ***A*** to be factored.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 159 on page 640.

lda
is the leading dimension of the array specified for ***A***.

Specified as: an integer; *lda* > 0 and *lda* ≥ 2*kl*+*ku*+1.

ipiv
See "On Return".

info
See "On Return".

On Return

a
is the transformed matrix ***A*** containing the results of the factorization. See "Function" on page 641.

Returned as: an *lda* by (at least) *n* array, containing integers.

ipiv
is the integer vector of length min(*m*,*n*), containing the pivot indices.

Returned as: a one-dimensional integer array of (at least) length min(*m*,*n*), containing integers; 1 ≤ $ipiv_j$ ≤ *m* for all *j*.

info
has the following meaning:

If *info* = 0, the subroutine completed successfully.

If *info* > 0, *info* is set to the first *i*, where $\boldsymbol{U}_{ii}$ is zero. The factorization has been completed.

Returned as:

- For SGBTRF, DGBTRF, CGBTRF and ZGBTRF, returned as: an integer; info ≥ 0.
- For LAPACKE_sgbtrf, LAPACKE_dgbtrf, LAPACKE_cgbtrf and LAPACKE_zgbtrf, returned as an integer function value; info ≥ 0.

Note:

1. In your C program, argument *info* must be passed by reference.
2. *a* and *ipiv* must have no common elements; otherwise, results are unpredictable.
3. For a description of how a general band matrix is stored in BLAS-general-band storage mode in an array, see "General Band Matrix" on page 94.
4. The way these subroutines handle singularity differs from LAPACK. Like LAPACK, these subroutines use the *info* argument to provide information about the singularity of ***A***, but they also provide an error message.

Function

These subroutines factor general band matrix ***A***, stored in BLAS-general-band storage mode, using Gaussian elimination with partial pivoting to compute the ***LU*** factorization of ***A***:

A = ***PLU***

In the formula above:

P is the permutation matrix
L is a unit lower triangular band matrix
U is a upper triangular band matrix

To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGBTRS, DGBTRS, CGBTRS, and ZGBTRS respectively.

If *m* or *n* is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See references ["9" on page 1253] and ["48" on page 1255].

Error conditions

Resource Errors

None

Computational Errors

Matrix ***A*** is singular or nearly singular.

- The first column, *i*, of ***L*** with a corresponding $\boldsymbol{U}_{ii} = 0$ diagonal element is identified in the computational error message.
- The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2146 is set to be unlimited in the ESSL error option table.

Input-Argument Errors

1. $m < 0$
2. $n < 0$
3. $kl < 0$
4. $ku < 0$
5. $lda \leq 0$
6. $2kl + ku + 1 > lda$

Examples

Example 1

This example shows a factorization of the following real general band matrix of order 9. Matrix ***A*** is:

```
     |  1.0 2.0 3.0 4.0 0.0 0.0 0.0 0.0 0.0 |
     |  2.0 1.0 2.0 3.0 4.0 0.0 0.0 0.0 0.0 |
     |  3.0 2.0 1.0 2.0 3.0 4.0 0.0 0.0 0.0 |
     |  0.0 3.0 2.0 1.0 2.0 3.0 4.0 0.0 0.0 |
     |  0.0 0.0 3.0 2.0 1.0 2.0 3.0 4.0 0.0 |
     |  0.0 0.0 0.0 3.0 2.0 1.0 2.0 3.0 4.0 |
     |  0.0 0.0 0.0 0.0 3.0 2.0 1.0 2.0 3.0 |
     |  0.0 0.0 0.0 0.0 0.0 3.0 2.0 1.0 2.0 |
     |  0.0 0.0 0.0 0.0 0.0 0.0 3.0 2.0 1.0 |
```

Call Statement and Input:

```
             M   N  KL  KU   A  LDA  IPIV   INFO)
             |   |   |   |   |   |     |      |
CALL DGBTRF( 9 , 9 , 2 , 3 , A , 8 , IPIV , INFO)
```

```
     |  .   .   .   .   .   .   .   .   .  |
     |  .   .   .   .   .   .   .   .   .  |
     |  .   .   .  4.0 4.0 4.0 4.0 4.0 4.0 |
     |  .   .  3.0 3.0 3.0 3.0 3.0 3.0 3.0 |
 A = |  .  2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 |
     | 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 |
     | 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0  .  |
     | 3.0 3.0 3.0 3.0 3.0 3.0 3.0  .   .  |
```

Output:

```
     |  .      .     .     .      .     4.000  4.000  4.000  4.000 |
     |  .      .     .     .     3.000  3.000  3.000  3.000 -2.271 |
     |  .      .     .   2.000  2.000  2.000  2.000 -4.074 -1.747 |
     |  .      .   1.000 1.000  1.000  1.000 -4.691 -4.177  1.000 |
 A = |  .    2.000 2.000 2.000  2.000 -4.419 -3.174  2.000  0.927 |
     | 3.000  3.000 3.000 3.000 -3.617 -5.095  3.000 -1.546  3.037 |
     | 0.666  0.444 0.518 0.567 -0.334  0.326  0.043 -0.790   .    |
     | 0.333 -0.111 0.592 0.246 -0.829 -0.588 -0.617   .      .    |
```

```
IPIV     =  ( 3, 4, 5, 6, 5, 6, 9, 8, 9 )

INFO = 0
```

Example 2

This example shows a factorization of the following complex general band matrix of order 5. Matrix ***A*** is:

```
     |( 0.100, 0.100) ( 1.000, 2.000) ( 1.000, 3.000) ( 0.000, 0.000) ( 0.000, 0.000)  |
     |( 2.000, 1.000) ( 0.200, 0.200) ( 2.000, 3.000) ( 2.000, 4.000) ( 0.000, 0.000)  |
     |( 0.000, 0.000) ( 3.000, 2.000) ( 0.300, 0.300) ( 3.000, 4.000) ( 3.000, 5.000)  |
     |( 0.000, 0.000) ( 0.000, 0.000) ( 4.000, 3.000) ( 0.400, 0.400) ( 4.000, 5.000)  |
     |( 0.000, 0.000) ( 0.000, 0.000) ( 0.000, 0.000) ( 5.000, 4.000) ( 0.500, 0.500)  |
```

Call Statement and Input:

```
             M   N  KL  KU   A  LDA  IPIV   INFO)
             |   |   |   |   |   |     |      |
CALL ZGBTRF( 5 , 5 , 1 , 2 , A , 5 , IPIV , INFO)
```

```
      |       .               .               .               .               .         |
      |       .               .        ( 1.000, 3.000) ( 2.000, 4.000) ( 3.000, 5.000)  |
  A = |       .        ( 1.000, 2.000) ( 2.000, 3.000) ( 3.000, 4.000) ( 4.000, 5.000)  |
      |( 0.100, 0.100) ( 0.200, 0.200) ( 0.300, 0.300) ( 0.400, 0.400) ( 0.500, 0.500)  |
      |( 2.000, 1.000) ( 3.000, 2.000) ( 4.000, 3.000) ( 5.000, 4.000)        .         |
```

Output:

```
     |       .               .               .        ( 2.000, 4.000) ( 3.000, 5.000)  |
     |       .               .        ( 2.000, 3.000) ( 3.000, 4.000) ( 4.000, 5.000)  |
A =  |       .        ( 0.200, 0.200) ( 0.300, 0.300) ( 0.400, 0.400) ( 0.500, 0.500)  |
     |( 2.000, 1.000) ( 3.000, 2.000) ( 4.000, 3.000) ( 5.000, 4.000) (-0.339,-6.596)  |
     |( 0.060, 0.020) ( 0.534, 0.305) ( 0.443, 0.300) (-0.412,-0.396)        .         |


IPIV     =  ( 2, 3, 4, 5, 5 )

INFO = 0
```

SGBTRS, DGBTRS, CGBTRS, and ZGBTRS (General Band Matrix Multiple Right-Hand Side Solve)

Purpose

SGBTRS and DGTBRS solve one of the following systems of equations for multiple right-hand sides:

1. $\boldsymbol{AX}=\boldsymbol{B}$
2. $\boldsymbol{A}^T\boldsymbol{X}=\boldsymbol{B}$

CGBTRS and ZGBTRS solve one of the following systems of equations for multiple right-hand sides:

1. $\boldsymbol{AX} = \boldsymbol{B}$
2. $\boldsymbol{A}^T\boldsymbol{X}=\boldsymbol{B}$
3. $\boldsymbol{A}^H\boldsymbol{X}=\boldsymbol{B}$

In the formulas above:

- ***A*** represents the general band matrix ***A*** stored in BLAS-general-band storage mode, containing the factorization.
- ***B*** represents the general matrix ***B*** containing the right-hand sides in its columns.
- ***X*** represents the general matrix ***B*** containing the solution vectors in its columns.

These subroutines use the results of the factorization of matrix ***A*** and vector *ipiv*, produced by a preceding call to SGBTRF, DGBTRF, CGBTRF, and ZGBTRF, respectively.

Table 161: Data Types.

A*, *B	**Subroutine**
Short-precision real	SGBTRS$^{\Delta}$
Long-precision real	DGBTRS$^{\Delta}$
Short-precision complex	CGBTRS$^{\Delta}$
Long-precision complex	ZGBTRS$^{\Delta}$
$^{\Delta}$LAPACK	

Note: The input to these solve subroutines must be the output from the factorization subroutines SGBTRF, DGBTRF, CGBTRF, and ZGBTRF, respectively.

Syntax

Table 162: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SGBTRS | DGBTRS | CGBTRS | ZGBTRS (*trans*, *n*, *kl*, *ku*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*, *info*)
C and C++	sgbtrs | dgbtrs | cgbtrs | zgbtrs (*trans*, *n*, *kl*, *ku*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*, *info*);
LAPACKE	info = LAPACKE_sgbtrs | LAPACKE_dgbtrs | LAPACKE_cgbtrs | LAPACKE_zgbtrs (*matrix_layout*, *trans*, *n*, *kl*, *ku*, *nrhs*, *a*, *lda*, *ipiv*, *b*, *ldb*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

trans

indicates the form of matrix ***A*** to use in the computation, where:

If *trans* = 'N', ***A*** is used in the computation, resulting in solution 1.

If *trans* = 'T', $\boldsymbol{A}^T$ is used in the computation, resulting in solution 2.

If *trans* = 'C', $\boldsymbol{A}^H$ is used in the computation, resulting in solution 3.

Specified as: a single character; *transa* = 'N', 'T', or 'C'.

n

is the order of factored matrix ***A*** and the number of rows in matrix ***B***.

Specified as: an integer; *n* ≥ 0.

kl
is the lower band width *kl* of the matrix ***A***.

Specified as: an integer; *kl* ≥ 0.

ku
is the upper band width *ku* of the matrix ***A***.

Specified as: an integer; *ku* ≥ 0.

nrhs
is the number of right-hand sides; that is, the number of columns of matrix ***B*** used in the computation.

Specified as: an integer; *nrhs* ≥ 0.

a
is the factorization of matrix ***A***, produced by a preceding call to SGBTRF, DGBTRF, CGBTRF, or ZGBTRF, respectively.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 161 on page 644.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and *lda* ≥ 2*kl*+*ku*+1.

ipiv
is the array containing the pivot indices produced by a preceding call to SGBTRF, DGBTRF, CGBTRF, or ZGBTRF, respectively.

Specified as: a one-dimensional array of (at least) length *n*, containing integers.

b
is the general matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of matrix ***B***.

Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 161 on page 644.

ldb
is the leading dimension of the array specified for ***B***.

Specified as: an integer; *ldb* > 0 and *ldb* ≥ *n*.

info
See "On Return".

On Return

b
is the matrix ***X***, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of ***X***.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 161 on page 644.

info
has the following meaning:

If *info* = 0, the subroutine completed successfully.

SGBTRS, DGBTRS, CGBTRS, and ZGBTRS

Returned as:

- For SGBTRS, DGBTRS, CGBTRS, and ZGBTRS, returned as: an integer; info ≥ 0.
- For LAPACKE_sgbtrs, LAPACKE_dgbtrs, LAPACKE_cgbtrs, and LAPACKE_zgbtrs, returned as an integer function value; info ≥ 0.

Note:

1. These subroutines accept lowercase letters for the *trans* arguments.

2. In your C program, argument *info* must be passed by reference.
3. For SGBTRS and DGBTRS, if you specify 'C' for the *trans* argument, it is interpreted as though you specified 'T'.
4. The scalar data specified for input argument *n* must be the same for both _GTBRF and _GTBRS.
5. The array data specified for input arguments *a* and *ipiv* for these subroutines must be the same as the corresponding output arguments for SGBTRF, DGBTRF, CGBTRF, or ZGBTRF respectively.
6. *a*, *ipiv*, and *b* must have no common elements; otherwise, results are unpredictable.
7. For a description of how a general band matrix is stored in BLAS-general-band storage mode in an array, see "General Band Matrix" on page 94.

Function

One of the following systems of equations is solved for multiple right-hand sides:

1. $\boldsymbol{AX}=\boldsymbol{B}$
2. $\boldsymbol{A}^{T}\boldsymbol{X}=\boldsymbol{B}$
3. $\boldsymbol{A}^{H}\boldsymbol{X}=\boldsymbol{B}$ (only for CGBTRS and ZGBTRS)

where $\boldsymbol{A}$ is a general band matrix and $\boldsymbol{B}$ and $\boldsymbol{X}$ are general matrices. These subroutines uses the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SGBTRF, DGBTRF, CGBTRF, or ZGBTRF, respectively. For details on the factorization, see "SGBTRF, DGBTRF, CGBTRF and ZGBTRF (General Band Matrix Factorization)" on page 639.

If *n* or *nrhs* is 0, no computation is performed.

See reference ["48" on page 1255].

Error conditions

Resource Errors

None

Computational Errors

Note: If the factorization performed by SGBTRF, DGBTRF, CGBTRF, or ZGBTRF failed due to a singular matrix argument, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. *trans* ≠ 'N', 'T, or 'C'
2. $n < 0$
3. $kl < 0$
4. $ku < 0$
5. $nrhs < 0$
6. $lda \le 0$
7. $2kl+ku+1 > lda$
8. $ldb \le 0$
9. $n > ldb$

Examples

Example 1

This example shows how to solve the system $\boldsymbol{AX} = \boldsymbol{B}$, where real general band matrix $\boldsymbol{A}$ is the same matrix factored in Example 1 for DGBTRF.

Call Statement and Input:

```
            TRANS   N   KL   KU NRHS   A   LDA   IPIV    B   LDB   INFO)
              |     |    |    |    |   |    |      |     |    |      |
CALL DGBTRS( 'N' , 9 , 2 , 3 , 3 , A , 8 , IPIV , B , 9 , INFO)
```

A = (same as output A in Example 1)

IPIV = (same as output IPIV in Example 1)

```
      |  1.0   1.0   1.0 |
      |  1.0  -1.0   2.0 |
      |  1.0   1.0   3.0 |
      |  1.0  -1.0   4.0 |
 B =  |  1.0   1.0   5.0 |
      |  1.0  -1.0   6.0 |
      |  1.0   1.0   7.0 |
      |  1.0  -1.0   8.0 |
      |  1.0   1.0   9.0 |
```

Output:

```
      |  0.629   1.149   4.713 |
      | -0.025   1.870  -0.726 |
      |  0.523   0.615   2.946 |
      | -0.286  -1.434  -2.774 |
 B =  | -0.104  -0.524  -1.067 |
      | -0.118  -0.591  -0.970 |
      |  0.220  -0.896   2.027 |
      | -0.079   1.604  -0.340 |
      |  0.496   0.480   3.599 |


INFO = 0
```

Example 2

This example shows how to solve the system $\boldsymbol{A}^T\boldsymbol{X} = \boldsymbol{B}$, where real general band matrix ***A*** is the same matrix factored in Example 1 for DGBTRF.

Call Statement and Input:

```
            TRANS   N   KL   KU NRHS   A   LDA   IPIV    B   LDB   INFO)
              |     |    |    |    |   |    |      |     |    |      |
CALL DGBTRS( 'T' , 9 , 2 , 3 , 3 , A , 8 , IPIV , B , 9 , INFO)
```

A = (same as output A in Example 1)

IPIV = (same as output IPIV in Example 1)

```
      |  1.0   1.0   1.0 |
      |  1.0  -1.0   2.0 |
      |  1.0   1.0   3.0 |
      |  1.0  -1.0   4.0 |
 B =  |  1.0   1.0   5.0 |
      |  1.0  -1.0   6.0 |
      |  1.0   1.0   7.0 |
      |  1.0  -1.0   8.0 |
      |  1.0   1.0   9.0 |
```

Output:

```
      |  0.496   0.480   1.362 |
      | -0.079   1.604  -0.450 |
      |  0.220  -0.896   0.179 |
      | -0.118  -0.591  -0.211 |
 B =  | -0.104  -0.524   0.018 |
      | -0.286  -1.434  -0.095 |
      |  0.523   0.615   2.285 |
      | -0.025   1.870   0.468 |
      |  0.629   1.149   1.586 |


INFO = 0
```

Example 3

This example shows how to solve the system ***AX*** = ***B***, where complex general band matrix ***A*** is the same matrix factored in Example 2 for ZGBTRF.

Call Statement and Input:

```
             TRANS   N  KL   KU NRHS   A  LDA   IPIV    B  LDB   INFO)
               |     |   |    |   |    |   |      |     |   |      |
CALL ZGBTRS( 'N' , 5 , 1 , 2 , 3 , A , 5 , IPIV , B , 5 , INFO)
```

A = (same as output A in Example 2)

IPIV = (same as output IPIV in Example 2)

```
     |( 0.100, 1.000)(-1.000, 1.000)( 0.200, 0.400)  |
     |( 0.100, 1.000)(-1.000, 1.000)( 0.400, 0.800)  |
 B = |( 0.100, 1.000)(-1.000, 1.000)( 0.600, 1.200)  |
     |( 0.100, 1.000)(-1.000, 1.000)( 0.800, 1.600)  |
     |( 0.100, 1.000)(-1.000, 1.000)( 1.000, 2.000)  |
```

Output:

```
     |(-0.039,-0.250)( 0.237,-0.265)(-0.247,-0.723)  |
     |( 0.161,-0.014)( 0.158, 0.163)(-0.092,-0.192)  |
 B = |( 0.205, 0.105)( 0.069, 0.317)( 0.248, 0.130)  |
     |( 0.116, 0.109)(-0.015, 0.224)( 0.314, 0.146)  |
     |(-0.055, 0.038)(-0.090,-0.027)( 0.032,-0.001)  |


INFO = 0
```

Example 4

This example shows how to solve the system ***A***^{H}***X*** = ***B***, where complex general band matrix ***A*** is the same matrix factored in Example 2 for ZGBTRF.

Call Statement and Input:

```
             TRANS   N  KL   KU NRHS   A  LDA   IPIV   B  LDB   INFO)
               |     |   |    |   |    |   |      |    |   |      |
CALL ZGBTRS( 'C' , 5 , 1 , 2 , 3 , A , 5 , IPIV, B , 5 , INFO)
```

A = (same as output A in Example 2)

IPIV = (same as output IPIV in Example 2)

```
     |( 0.100, 1.000)(-1.000, 1.000)( 0.200, 0.400)  |
     |( 0.100, 1.000)(-1.000, 1.000)( 0.400, 0.800)  |
 B = |( 0.100, 1.000)(-1.000, 1.000)( 0.600, 1.200)  |
     |( 0.100, 1.000)(-1.000, 1.000)( 0.800, 1.600)  |
     |( 0.100, 1.000)(-1.000, 1.000)( 1.000, 2.000)  |
```

Output:

```
     |( 0.015,-0.126)( 0.150,-0.096)( 0.029, 0.017)  |
     |(-0.158, 0.428)(-0.607, 0.209)(-0.002, 0.200)  |
 B = |(-0.094, 0.283)(-0.392, 0.149)(-0.054, 0.230)  |
     |(-0.057,-0.111)( 0.070,-0.161)(-0.114, 0.109)  |
     |(-0.088,-0.409)( 0.367,-0.460)(-0.111,-0.050)  |


INFO = 0
```

SGBS and DGBS (General Band Matrix Solve)

Purpose

These subroutines solve the system ***Ax*** = ***b*** for ***x***, where ***A*** is a general band matrix, and ***x*** and ***b*** are vectors. They use the results of the factorization of matrix ***A***, produced by a preceding call to SGBF or DGBF, respectively.

Table 163: Data Types.

Data Types	
A, ***b***, ***x***	**Subroutine**
Short-precision real	SGBS
Long-precision real	DGBS

Note: The input to these solve subroutines must be the output from the factorization subroutines SGBF and DGBF, respectively.

Syntax

Language	Syntax
Fortran	CALL SGBS | DGBS (*agb*, *lda*, *n*, *ml*, *mu*, *ipvt*, *bx*)
C and C++	sgbs | dgbs (*agb*, *lda*, *n*, *ml*, *mu*, *ipvt*, *bx*);

On Entry

agb
: is the factorization of general band matrix ***A***, produced by a preceding call to SGBF or DGBF. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 163 on page 649, where *lda* ≥ 2*ml*+*mu*+16.

lda
: is the leading dimension of the array specified for *agb*. Specified as: an integer; *lda* > 0 and *lda* ≥ 2*ml* +*mu*+16.

n
: is the order of the matrix ***A***. Specified as: an integer; *n* > *ml* and *n* > *mu*.

ml
: is the lower band width *ml* of the matrix ***A***. Specified as: an integer; 0 ≤ *ml* < *n*.

mu
: is the upper band width *mu* of the matrix ***A***. Specified as: an integer; 0 ≤ *mu* < *n*.

ipvt

is the integer vector ***ipvt*** of length *n*, produced by a preceding call to SGBF or DGBF. It contains the pivot information necessary to construct matrix ***L*** from the information contained in the array specified for *agb*.

Specified as: a one-dimensional array of (at least) length *n*, containing integers.

bx

is the vector ***b*** of length *n*, containing the right-hand side of the system. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 163 on page 649.

On Return

bx

is the solution vector ***x*** of length *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 163 on page 649.

Notes

1. The scalar data specified for input arguments *lda*, *n*, *ml*, and *mu* for these subroutines must be the same as that specified for SGBF and DGBF, respectively.
2. The array data specified for input arguments *agb* and *ipvt* for these subroutines must be the same as the corresponding output arguments for SGBF and DGBF, respectively.
3. The entire *lda* by *n* array specified for *agb* must remain unchanged between calls to the factorization and solve subroutines.
4. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
5. This subroutine can be used for tridiagonal matrices (*ml* = *mu* = 1); however, the tridiagonal subroutines, SGTF/DGTF and SGTS/DGTS, are faster.
6. For a description of how a general band matrix is stored in general-band storage mode in an array, see "General Band Matrix" on page 94.

Function

The real system ***Ax*** = ***b*** is solved for ***x***, where ***A*** is a real general band matrix, stored in general-band storage mode, and ***x*** and ***b*** are vectors. These subroutines use the results of the factorization of matrix ***A***, produced by a preceding call to SGBF or DGBF, respectively. The transformed matrix ***A***, used by this computation, consists of the upper triangular matrix ***U*** and the multipliers necessary to construct ***L*** using ***ipvt***, as defined in "Function" on page 694. See reference ["48" on page 1255].

Error conditions

Computational Errors

Note: If the factorization performed by SGBF or DGBF failed due to a singular matrix argument, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. $lda \leq 0$
2. $ml < 0$
3. $ml \geq n$
4. $mu < 0$
5. $mu \geq n$
6. $lda < 2ml+mu+16$

Examples

Example

This example shows how to solve the system ***Ax*** = ***b***, where general band matrix ***A*** is the same matrix factored in Example for SGBF and DGBF. The input for AGB and IPVT in this example is the same as the output for that example.

Call Statement and Input:

```
           AGB  LDA   N   ML  MU  IPVT   BX
            |    |    |   |   |    |     |
CALL SGBS( AGB , 23 , 9 , 2 , 3 , IPVT , BX )
```

`IPVT = (2, -65534, -131070, -196606, -262142, -327678, -327678, -327680, -327680)`
`BX = (4.0000, 5.0000, 9.0000, 10.0000, 11.0000, 12.0000, 12.0000, 12.0000, 33.0000)`
`AGB` = (same as output AGB in Example)

Output:

```
BX       =  (1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
             0.9999, 1.0001)
```

SPBSV, DPBSV, CPBSV, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system ***AX*** = ***B*** for ***X***, where ***X*** and ***B*** are general matrices and:

- For SPBSV and DPBSV, ***A*** is a positive definite real symmetric band matrix stored in upper- or lower-band-packed storage mode.
- For CPBSV and ZPBSV, ***A*** is a positive definite complex Hermitian band matrix stored in upper- or lower-band-packed storage mode.

Table 164: Data Types.

A, B	**Subroutine**
Short-precision real	SPBSV$^{\Delta}$
Long-precision real	DPBSV$^{\Delta}$
Short-precision complex	CPBSV$^{\Delta}$
Long-precision complex	ZPBSV$^{\Delta}$
$^{\Delta}$LAPACK	

Syntax

Table 165: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SPBSV | DPBSV | CPBSV | ZPBSV (*uplo*, *n*, *k*, *nrhs*, *a*, *lda*, *b*, *ldb*, *info*)
C and C++	spbsv | dpbsv | cpbsv | zpbsv (*uplo*, *n*, *k*, *nrhs*, *a*, *lda*, *b*, *ldb*, *info*);

Table 165: Syntax for supported computing languages. (continued)	
Language	**Syntax**
LAPACKE	info = LAPACKE_spbsv | LAPACKE_dpbsv | LAPACKE_cpbsv | LAPACKE_zpbsv (*matrix_layout*, *uplo*, *n*, *k*, *nrhs*, *a*, *lda*, *b*, *ldb*);

On Entry

matrix_layout
indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

uplo
indicates whether the matrix ***A*** is stored in upper- or lower-band-backed storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper-band-packed storage mode.

If *uplo* = 'L', ***A*** is stored in lower-band-packed storage mode.

Specified as: a single character. It must be 'U' or 'L'.

n
is the order *n* of matrix ***A*** and the number of rows of matrix ***B***. Specified as: an integer; $n \geq 0$.

k
is the half band width *k* of the matrix ***A***. Specified as: an integer; $0 \leq k \leq \max(0,n-1)$.

nrhs
is the number of right-hand sides; i.e., the number of columns of matrix ***B***. Specified as: an integer; $nrhs \geq 0$.

a
is the positive definite real symmetric or complex Hermitian band matrix ***A*** of order *n*, having a half band width of *k* , where:

- If *uplo* = 'U', it is stored in upper-band-packed storage mode.
- If *uplo* = 'L', it is stored in lower-band-packed storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 164 on page 651.

lda
is the leading dimension of the array specified for ***A***. Specified as: an integer; $lda > 0$ and $lda > k$.

b
is the matrix ***B*** of right-hand side vectors. Specified as: the *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 164 on page 651.

ldb
is the leading dimension of the array specified for ***B***. Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

On Return

a
If *info* = 0, *a* is the updated matrix ***A*** containing the results of the Cholesky factorization. See "Function" on page 653.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 164 on page 651.

b

If *info* = 0, *b* is the general matrix ***X*** containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of ***X***.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 164 on page 651.

info

has the following meaning:

If *info* = 0, the subroutine completed successfully.

If *info* = *i*, the leading minor of order *i* is not positive definite. The factorization could not be completed and the solution was not computed.

Returned as:

- For SPBSV, DPBSV, CPBSV, and ZPBSV, returned as: an integer; info ≥ 0.
- For LAPACKE_spbsv, LAPACKE_dpbsv, LAPACKE_cpbsv, and LAPACKE_zpbsv, returned as an integer function value; info ≥ 0.

Notes:

1. In your C program, argument *info* must be passed by reference.
2. All subroutines accept lowercase letters for the *uplo* argument.
3. For a description of how real symmetric matrices are stored in upper- or lower-band-packed storage mode, see "Upper-Band-Packed Storage Mode" on page 100 or "Lower-Band-Packed Storage Mode" on page 100, respectively.

   For a description of how complex Hermitian matrices are stored in upper- or lower-band-packed storage mode, see "Complex Hermitian Band Matrix Storage Representation" on page 102.
4. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.
5. The matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
6. The way these subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the *info* argument to provide information about the computational error, but they also provide an error message.
7. On both input and output, matrix ***A*** conforms to LAPACK format.

Function

These subroutines solve the system ***AX*** = ***B*** for ***X***, where ***X*** and ***B*** are general matrices and:

- For SPBSV and DPBSV, ***A*** is a positive definite real symmetric band matrix stored in upper- or lower-band-packed storage mode.
- For CPBSV and ZPBSV, ***A*** is a positive definite complex Hermitian band matrix stored in upper- or lower-band-packed storage mode.

Matrix ***A*** is factored using Cholesky factorization:

- **For SPBTRF and DPBTRF**:
  - $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^{T}$ if *uplo*= 'L'.
  - $\boldsymbol{A} = \boldsymbol{U}^{T}\boldsymbol{U}$ if *uplo*= 'U'.
- **For CPBTRF and ZPBTRF**:
  - $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^{H}$ if *uplo*= 'L'.
  - $\boldsymbol{A} = \boldsymbol{U}^{H}\boldsymbol{U}$ if *uplo*= 'U'.

Where:

- ***L*** is a lower triangular band matrix
- ***U*** is a upper triangular band matrix

If $n = 0$, no computation is performed and the subroutine returns after doing some parameter checking. If $n > 0$ and $nrhs = 0$, no solutions are computed and the subroutine returns after factoring the matrix. See references [“9” on page 1253],[“46” on page 1255], and [“75” on page 1257].

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
Matrix ***A*** is not positive definite. For details, see the description of the *info* argument.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. $n < 0$
3. $k < 0$
4. $k > \max(0,n-1)$
5. $nrhs < 0$
6. $lda \le 0$
7. $k \ge lda$
8. $ldb \le 0$
9. $n > ldb$

Examples

Example 1

This example shows how to solve the system ***AX*** = ***B***, where ***A*** is a real positive definite band matrix factored in the form $\boldsymbol{LL}^T$.

Matrix ***A*** is the same used as input in Example 1 for DPBTRF.

Matrix ***B*** is the same used as input in Example 1 for DPBTRS.

Call Statement and Input:

```
             UPLO  N   K  NRHS A  LDA  B  LDB  INFO
              |    |   |   |   |   |   |   |     |
CALL DPBSV( 'L' , 9 , 3 , 3 , A , 4 , B , 9 , INFO )

     A = (same as input A in Example 1)
     B = (same as input B in Example 1)
```

Output:

```
          | 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 |
      A = | 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0  .  |
          | 1.0 1.0 1.0 1.0 1.0 1.0 1.0  .   .  |
          | 1.0 1.0 1.0 1.0 1.0 1.0  .   .   .  |
```

```
          |  1.0  1.0  1.0 |
          |  1.0 -1.0  0.0 |
          |  1.0  1.0 -1.0 |
          |  1.0 -1.0  1.0 |
      B = |  1.0  1.0  0.0 |
          |  1.0 -1.0 -1.0 |
          |  1.0  1.0  1.0 |
          |  1.0 -1.0  0.0 |
          |  1.0  1.0 -1.0 |


    INFO = 0
```

Example 2

This example shows how to solve the system ***AX*** = ***B***, where ***A*** is a real positive definite symmetric band matrix factored in the form $\boldsymbol{U}^T\boldsymbol{U}$.

Matrix ***A*** is the same used as input in Example 2 for DPBTRF.

Matrix ***B*** is the same used as input in Example 2 for DPBTRS.

Call Statement and Input:

```
            UPLO   N   K  NRHS A  LDA  B  LDB  INFO
             |     |   |   |   |   |   |   |     |
CALL DPBSV( 'U' , 9 , 2 , 3 , A , 3 , B , 9 , INFO )

     A = (same as input A in Example 2)
     B = (same as input B in Example 2)
```

Output:

```

          |  .    .   1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
      A = |  .  -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 |
          | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
```

```

          |  1.0  1.0  1.0 |
          |  1.0 -1.0  0.0 |
          |  1.0  1.0 -1.0 |
          |  1.0 -1.0  1.0 |
      B = |  1.0  1.0  0.0 |
          |  1.0 -1.0 -1.0 |
          |  1.0  1.0  1.0 |
          |  1.0 -1.0  0.0 |
          |  1.0  1.0 -1.0 |


    INFO = 0
```

Example 3

This example shows how to solve the system ***AX*** = ***B***, where ***A*** is a positive definite complex Hermitian band matrix factored in the form $\boldsymbol{LL}^H$.

Matrix ***A*** is the same used as input in Example 3 for ZPBTRF.

Matrix ***B*** is the same used as input in Example 3 for ZPBTRS.

Call Statement and Input:

```
             UPLO   N   K  NRHS A  LDA  B  LDB  INFO
              |     |   |   |   |   |   |   |     |
CALL ZPBSV ( 'L' , 6 , 3 , 3 , A , 4 , B , 6 , INFO )

     A = (same as input A in Example 3)
     B = (same as input B in Example 3)
```

Output:

```

          |  (1.0, 0.0) (1.0, 0.0) (1.0, 0.0) (1.0, 0.0) (1.0, 0.0) (1.0, 0.0)  |
      A = |  (1.0, 1.0) (1.0, 1.0) (1.0, 1.0) (1.0, 1.0) (1.0, 1.0)     .       |
          |  (1.0, 1.0) (1.0, 1.0) (1.0, 1.0) (1.0, 1.0)     .          .       |
          |  (1.0, 1.0) (1.0, 1.0) (1.0, 1.0)     .          .          .       |
```

```
          |  (  7.0,  33.0)  (  1.0, -1.0)  (  1.0,  1.0) |
          |  ( 19.0,  -1.0)  (  1.0, -1.0)  (  1.0, -1.0) |
          |  (  5.0, -13.0)  (  1.0, -1.0)  (  2.0,  1.0) |
      B = |  (-11.0,  -5.0)  (  1.0, -1.0)  (  2.0, -1.0) |
          |  ( -3.0,   9.0)  (  1.0, -1.0)  (  1.0,  2.0) |
          |  (  5.0,  -1.0)  (  1.0, -1.0)  (  1.0, -2.0) |


    INFO = 0
```

Example 4

This example shows how to solve the system ***AX*** = ***B***, where ***A*** is a complex Hermitian band matrix factored in the form $\boldsymbol{U}^H\boldsymbol{U}$.

Matrix ***A*** is the same used as input in Example 4 for ZPBTRF.

Matrix ***B*** is the same used as input in Example 4 for ZPBTRS.

Call Statement and Input:

```
            UPLO  N   K  NRHS A  LDA  B  LDB  INFO
             |    |   |   |   |   |   |   |     |
CALL ZPBSV( 'U' , 6 , 2 , 3 , A , 3 , B , 6 , INFO )

     A = (same as input A in Example 4)
     B = (same as input B in Example 4)
```

Output:

```
        |      .          .      (1.0, -1.0) (1.0, 1.0) (1.0, -1.0) (1.0, 1.0)  |
    A = |      .      (1.0, 1.0) (1.0, -1.0) (1.0, 1.0) (1.0, -1.0) (1.0, 1.0)  |
        |  (1.0, 0.0) (1.0, 0.0) (1.0,  0.0) (1.0, 0.0) (1.0,  0.0) (1.0, 0.0)  |
```

```
       |  (  5.0, 13.0)  (  1.0, -1.0)  (  1.0,  1.0) |
       |  ( -3.0,  7.0)  (  1.0, -1.0)  (  1.0, -1.0) |
       |  ( 11.0, -5.0)  (  1.0, -1.0)  (  2.0,  1.0) |
   B = |  (  3.0,  7.0)  (  1.0, -1.0)  (  2.0, -1.0) |
       |  (  1.0, -5.0)  (  1.0, -1.0)  (  1.0,  2.0) |
       |  (  1.0,  1.0)  (  1.0, -1.0)  (  1.0, -2.0)
|


INFO = 0
```

SPBTRF, DPBTRF, CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)

Purpose

These subroutines use Cholesky factorization to factor a positive definite real symmetric or complex Hermitian band matrix ***A***, stored in upper- or lower-band-packed storage mode.

To solve the system of equations, follow the call to these subroutines with one or more calls to SPBTRS, DPBTRS, CPBTRS, or ZPBTRS, respectively.

Table 166: Data Types.

A	**Subroutine**
Short-precision real	SPBTRF$^{\Delta}$

Table 166: Data Types. (continued)

A	**Subroutine**
Long-precision real	DPBTRF$^{\Delta}$
Short-precision complex	CPBTRF$^{\Delta}$
Long-precision complex	ZPBTRF$^{\Delta}$
$^{\Delta}$LAPACK	

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SPBTRS, DPBTRS, CPBTRS, or ZPBTRS, respectively.

Syntax

Table 167: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SPBTRF | DPBTRF | CPBTRF | ZPBTRF (*uplo*, *n*, *k*, *a*, *lda*, *info*)
C and C++	spbtrf | dpbtrf | cpbtrf | zpbtrf (*uplo*, *n*, *k*, *a*, *lda*, *info*);
LAPACKE	info = LAPACKE_spbtrf | LAPACKE_dpbtrf | LAPACKE_cpbtrf | LAPACKE_zpbtrf (*matrix_layout*, *uplo*, *n*, *k*, *a*, *lda*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

uplo

indicates whether matrix ***A*** is stored in upper- or lower-band-packed storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper-band-packed storage mode.

If *uplo* = 'L', ***A*** is stored in lower-band-packed storage mode.

Specified as: a single character. It must be 'U' or 'L'.

n

is the order *n* of matrix ***A***. Specified as: an integer; $n \geq 0$.

k

is the half band width *k* of the matrix ***A***. Specified as: an integer; $0 \leq k \leq \max(0,n-1)$.

a

is the positive definite real symmetric or complex Hermitian band matrix ***A*** of order *n*, having a half band width of *k* , where:

- If *uplo* = 'U', it is stored in upper-band-packed storage mode.
- If *uplo* = 'L', it is stored in lower-band-packed storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 166 on page 656.

lda

is the leading dimension of the array specified for ***A***. Specified as: an integer; *lda* > 0 and *lda* > *k*.

On Return

a

If *info* = 0, *a* is the updated matrix ***A*** containing the results of the Cholesky factorization. See "Function" on page 658.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 166 on page 656.

info

has the following meaning:

If *info* = 0, the subroutine completed successfully.

If *info* = *i*, the leading minor of order *i* is not positive definite and the factorization could not be completed.

Returned as:

- For SPBTRF, DPBTRF, CPBTRF, and ZPBTR, returned as: an integer; info ≥ 0.
- For LAPACKE_spbtrf, LAPACKE_dpbtrf, LAPACKE_cpbtrf, and LAPACKE_zpbtrf, returned as an integer function value; info ≥ 0.

Notes:

1. These subroutines accept lower case letters for the *uplo* argument.
2. In your C program, argument *info* must be passed by reference.
3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.
4. For a description of how real symmetric matrices are stored in upper- or lower-band-packed storage mode, see "Upper-Band-Packed Storage Mode" on page 100 or"Lower-Band-Packed Storage Mode" on page 100, respectively.

   For a description of how complex Hermitian matrices are stored in upper- or lower-band-packed storage mode, see "Complex Hermitian Band Matrix Storage Representation" on page 102.
5. The way these subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the *info* argument to provide information about the computational error, but they also provide an error message.
6. On both input and output, matrix ***A*** conforms to LAPACK format.

Function

These subroutines use Cholesky factorization to factor a positive definite real symmetric or complex Hermitian band matrix ***A***, stored in upper- or lower-band-packed storage mode:

- **For SPBTRF and DPBTRF**:
  - $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T$ if *uplo*= 'L'.
  - $\boldsymbol{A} = \boldsymbol{U}^T\boldsymbol{U}$ if *uplo*= 'U'.
- **For CPBTRF and ZPBTRF**:
  - $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^H$ if *uplo*= 'L'.
  - $\boldsymbol{A} = \boldsymbol{U}^H\boldsymbol{U}$ if *uplo*= 'U'.

Where:

- ***L*** is a lower triangular band matrix
- ***U*** is a upper triangular band matrix

This factorization can then be used by SPBTRS, DPBTRS, CPBTRS, or ZPBTRS, respectively, to solve the system of equations.

If n = 0, no computation is performed.

Error conditions

Resource Errors

Unable to allocate internal work area.

Computational Errors

Matrix ***A*** is not positive definite. For details, see the description of the *info* argument.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. $n < 0$
3. $k < 0$
4. $k > \max(0,n-1)$
5. $lda \leq 0$
6. $k \geq lda$

Examples

Example 1

This example shows a factorization of the following real positive definite symmetric band matrix ***A*** in the form ***A*** = ***LL***^{T}:

```
          |  1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 |
          |  1.0 2.0 2.0 2.0 1.0 0.0 0.0 0.0 0.0 |
          |  1.0 2.0 3.0 3.0 2.0 1.0 0.0 0.0 0.0 |
          |  1.0 2.0 3.0 4.0 3.0 2.0 1.0 0.0 0.0 |
          |  0.0 1.0 2.0 3.0 4.0 3.0 2.0 1.0 0.0 |
          |  0.0 0.0 1.0 2.0 3.0 4.0 3.0 2.0 1.0 |
          |  0.0 0.0 0.0 1.0 2.0 3.0 4.0 3.0 2.0 |
          |  0.0 0.0 0.0 0.0 1.0 2.0 3.0 4.0 3.0 |
          |  0.0 0.0 0.0 0.0 0.0 1.0 2.0 3.0 4.0 |
```

Call Statement and Input:

```
             UPLO  N   K    A  LDA  INFO
              |    |   |    |   |    |
CALL DPBTRF( 'L' , 9 , 3 ,  A , 4 , INFO )


       | 1.0 2.0 3.0 4.0 4.0 4.0 4.0 4.0 4.0 |
   A = | 1.0 2.0 3.0 3.0 3.0 3.0 3.0 3.0  .  |
       | 1.0 2.0 2.0 2.0 2.0 2.0 2.0  .   .  |
       | 1.0 1.0 1.0 1.0 1.0 1.0  .   .   .  |
```

Output:

```
       | 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 |
   A = | 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0  .  |
       | 1.0 1.0 1.0 1.0 1.0 1.0 1.0  .   .  |
       | 1.0 1.0 1.0 1.0 1.0 1.0  .   .   .  |


   INFO = 0
```

Example 2

This example shows a factorization of the following real positive definite symmetric band matrix ***A*** in the form ***A*** = ***U***^{T}***U***:

```
          |  1.0 -1.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0 |
          | -1.0  2.0 -2.0  1.0  0.0  0.0  0.0  0.0  0.0 |
          |  1.0 -2.0  3.0 -2.0  1.0  0.0  0.0  0.0  0.0 |
          |  0.0  1.0 -2.0  3.0 -2.0  1.0  0.0  0.0  0.0 |
          |  0.0  0.0  1.0 -2.0  3.0 -2.0  1.0  0.0  0.0 |
          |  0.0  0.0  0.0  1.0 -2.0  3.0 -2.0  1.0  0.0 |
          |  0.0  0.0  0.0  0.0  1.0 -2.0  3.0 -2.0  1.0 |
          |  0.0  0.0  0.0  0.0  0.0  1.0 -2.0  3.0 -2.0 |
          |  0.0  0.0  0.0  0.0  0.0  0.0  1.0 -2.0  3.0 |
```

Call Statement and Input:

```
             UPLO  N   K    A  LDA  INFO
              |    |   |    |   |    |
CALL DPBTRF( 'U' , 9 , 2 ,  A , 3 , INFO )


          |  .    .   1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
      A = |  .  -1.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 |
          | 1.0  2.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0 |
```

Output:

```
          |  .    .   1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
      A = |  .  -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 |
          | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |


    INFO = 0
```

Example 3

This example shows a factorization of the following positive definite complex Hermitian band matrix ***A*** in the form ***A*** = ***LL***^{H}:

```
          |  (1.0, 0.0) (1.0,-1.0) (1.0,-1.0) (1.0,-1.0) (0.0, 0.0) (0.0, 0.0)  |
          |  (1.0, 1.0) (3.0, 0.0) (3.0,-1.0) (3.0,-1.0) (1.0,-1.0) (0.0, 0.0)  |
          |  (1.0, 1.0) (3.0, 1.0) (5.0, 0.0) (5.0,-1.0) (3.0,-1.0) (1.0,-1.0)  |
          |  (1.0, 1.0) (3.0, 1.0) (5.0, 1.0) (7.0, 0.0) (5.0,-1.0) (3.0,-1.0)  |
          |  (0.0, 0.0) (1.0, 1.0) (3.0, 1.0) (5.0, 1.0) (7.0, 0.0) (5.0,-1.0)  |
          |  (0.0, 0.0) (0.0, 0.0) (1.0, 1.0) (3.0, 1.0) (5.0, 1.0) (7.0, 0.0)  |
```

Call Statement and Input:

```
             UPLO  N   K    A  LDA  INFO
              |    |   |    |   |    |
CALL ZPBTRF( 'L' , 6 , 3 ,  A , 4 , INFO )


          |  (1.0,  . ) (3.0,  . ) (5.0,  . ) (7.0,  . ) (7.0,  . ) (7.0,  . )  |
      A = |  (1.0, 1.0) (3.0, 1.0) (5.0, 1.0) (5.0, 1.0) (5.0, 1.0)     .       |
          |  (1.0, 1.0) (3.0, 1.0) (3.0, 1.0) (3.0, 1.0)     .          .       |
          |  (1.0, 1.0) (1.0, 1.0) (1.0, 1.0)     .          .          .       |
```

Output:

```
          |  (1.0, 0.0) (1.0, 0.0) (1.0, 0.0) (1.0, 0.0) (1.0, 0.0) (1.0, 0.0)  |
      A = |  (1.0, 1.0) (1.0, 1.0) (1.0, 1.0) (1.0, 1.0) (1.0, 1.0)     .       |
          |  (1.0, 1.0) (1.0, 1.0) (1.0, 1.0) (1.0, 1.0)     .          .       |
          |  (1.0, 1.0) (1.0, 1.0) (1.0, 1.0)     .          .          .       |


    INFO = 0
```

Example 4

This example shows a factorization of the following positive definite complex Hermitian band matrix ***A*** in the form ***A*** = $\boldsymbol{U}^H\boldsymbol{U}$:

```
         |  (1.0, 0.0) (1.0, 1.0) (1.0,-1.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0)  |
         |  (1.0,-1.0) (3.0, 0.0) (1.0,-3.0) (1.0, 1.0) (0.0, 0.0) (0.0, 0.0)  |
         |  (1.0, 1.0) (1.0, 3.0) (5.0, 0.0) (1.0, 3.0) (1.0,-1.0) (0.0, 0.0)  |
         |  (0.0, 0.0) (1.0,-1.0) (1.0,-3.0) (5.0, 0.0) (1.0,-3.0) (1.0, 1.0)  |
         |  (0.0, 0.0) (0.0, 0.0) (1.0, 1.0) (1.0, 3.0) (5.0, 0.0) (1.0, 3.0)  |
         |  (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (1.0,-1.0) (1.0,-3.0) (5.0, 0.0)  |
```

Call Statement and Input:

```
              UPLO  N   K    A  LDA  INFO
               |    |   |    |   |    |
CALL ZPBTRF( 'U' , 6 , 2 ,  A , 3 , INFO )
```

```
         |      .          .      (1.0,-1.0) (1.0, 1.0) (1.0,-1.0) (1.0, 1.0)  |
     A = |      .      (1.0, 1.0) (1.0,-3.0) (1.0, 3.0) (1.0,-3.0) (1.0, 3.0)  |
         |  (1.0,  . ) (3.0,  . ) (5.0,  . ) (5.0,  . ) (5.0,  . ) (5.0,  . )  |
```

Output:

```
         |      .          .      (1.0, -1.0) (1.0, 1.0) (1.0, -1.0) (1.0, 1.0)  |
     A = |      .      (1.0, 1.0) (1.0, -1.0) (1.0, 1.0) (1.0, -1.0) (1.0, 1.0)  |
         |  (1.0, 0.0) (1.0, 0.0) (1.0,  0.0) (1.0, 0.0) (1.0,  0.0) (1.0, 0.0)  |


    INFO = 0
```

SPBTRS, DPBTRS, CPBTRS, and ZPBTRS (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system ***AX*** = ***B*** for ***X***, where ***X*** and ***B*** are general matrices and:

- For SPBTRS and DPBTRS, ***A*** is a positive definite real symmetric band matrix.
- For CPBTRS and ZPBTRS, ***A*** is a positive definite complex Hermitian band matrix.

Table 168: Data Types.

A*, *B	**Subroutine**
Short-precision real	SPBTRS$^{\Delta}$
Long-precision real	DPBTRS$^{\Delta}$
Short-precision complex	CPBTRS$^{\Delta}$
Long-precision complex	ZPBTRS$^{\Delta}$
$^{\Delta}$LAPACK	

Note: The input to these solve subroutines must be the output from the factorization subroutines SPBTRF, DPBTRF, CPBTRF, and ZPBTRF, respectively.

Syntax

Table 169: Syntax for supported computing languages.

Language	Syntax
Fortran	CALL SPBTRS | DPBTRS | CPBTRS | ZPBTRS (*uplo*, *n*, *k*, *nrhs*, *a*, *lda*, *b*, *ldb*, *info*)
C and C++	spbtrs | dpbtrs | cpbtrs | zpbtrs (*uplo*, *n*, *k*, *nrhs*, *a*, *lda*, *b*, *ldb*, *info*);
LAPACKE	info = LAPACKE_spbtrs | LAPACKE_dpbtrs | LAPACKE_cpbtrs | LAPACKE_zpbtrs (*matrix_layout*, *uplo*, *n*, *k*, *nrhs*, *a*, *lda*, *b*, *ldb*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

uplo

indicates whether the factored matrix ***A*** is stored in upper- or lower-band-backed storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper-band-packed storage mode.

If *uplo* = 'L', ***A*** is stored in lower-band-packed storage mode.

Specified as: a single character. It must be 'U' or 'L'.

n

is the order *n* of matrix ***A*** and the number of rows of matrix ***B***. Specified as: an integer; $n \geq 0$.

k

is the half band width *k* of the matrix ***A***. Specified as: an integer; $0 \leq k \leq \max(0,n-1)$.

nrhs

is the number of right-hand sides; i.e., the number of columns of matrix ***B***. Specified as: an integer; $nrhs \geq 0$.

a

is the factorization of positive definite matrix ***A***, produced by a preceding call to SPBTRF, DPBTRF, CPBTRF, and ZPBTRF, respectively.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 168 on page 661.

lda

is the leading dimension of the array specified for ***A***. Specified as: an integer; $lda > 0$ and $lda > k$.

b

is the matrix ***B*** of right-hand side vectors. Specified as: the *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 168 on page 661.

ldb

is the leading dimension of the array specified for ***B***. Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

On Return

b

is the general matrix ***X***, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of ***X***.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 168 on page 661.

info

info has the following meaning:

If *info* = 0, the solve completed successfully.

S

Returned as:

- For SPBTRS, DPBTRS, CPBTRS, and ZPBTR, returned as: an integer; info ≥ 0.
- For LAPACKE_spbtrs, LAPACKE_dpbtrs, LAPACKE_cpbtrs, and LAPACKE_zpbtr, returned as an integer function value; info ≥ 0.

Notes:

1. In your C program, argument *info* must be passed by reference.
2. All subroutines accept lowercase letters for the *uplo* argument.
3. For a description of how real symmetric matrices are stored in upper- or lower-band-packed storage mode, see "Upper-Band-Packed Storage Mode" on page 100 or "Lower-Band-Packed Storage Mode" on page 100, respectively.

   For a description of how complex Hermitian matrices are stored in upper- or lower-band-packed storage mode, see "Complex Hermitian Band Matrix Storage Representation" on page 102.
4. The scalar data specified for input arguments *uplo*, *n*, *k*, and *lda* for these subroutines must be the same as the corresponding input arguments specified for SPBTRF, DPBTRF, CPBTRF, and ZPBTRF, respectively.
5. The array data specified for input argument *a* for these subroutines must be the same as the corresponding output argument for SPBTRF, DPBTRF, CPBTRF, and ZPBTRF, respectively.
6. The matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

These subroutines solve the system ***AX*** = ***B*** for ***X***, where ***X*** and ***B*** are general matrices and:

- For SPBTRS and DPBTRS, ***A*** is a positive definite real symmetric band matrix.
- For CPBTRS and ZPBTRS, ***A*** is a positive definite complex Hermitian band matrix.

These subroutines use the results of the factorization of matrix ***A***, produced by a preceding call to SPBTRF, DPBTRF, CPBTRF, and ZPBTRF, respectively. For a description of how ***A*** is factored, see "SPBTRF, DPBTRF, CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)" on page 656.

If *n* or *nrhs* is 0, no computation is performed. See references ["9" on page 1253] and ["46" on page 1255].

Error conditions

Computational Errors

None

Note: If the factorization performed by SPBTRF, DPBTRF, CPBTRF, and ZPBTRF failed because matrix ***A*** was not positive definite, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. *n* < 0

3. $k < 0$
4. $k > \max(0,n-1)$
5. $nrhs < 0$
6. $lda \leq 0$
7. $k \geq lda$
8. $ldb \leq 0$
9. $n > ldb$

Examples

Example 1

This example shows how to solve the system ***AX*** = ***B***, where matrix ***A*** is the same positive definite symmetric band matrix factored in Example 1 for DPBTRF in the form ***A*** = ***LL***^{T}.

Call Statement and Input:

```
             UPLO  N   K  NRHS A  LDA  B  LDB  INFO
              |    |   |   |   |   |   |   |    |
CALL DPBTRS( 'L' , 9 , 3 , 3 , A , 4 , B , 9 , INFO )

     A = (same output A as in Example 1)
```

```
           |  4.0  0.0  1.0 |
           |  8.0  0.0  1.0 |
           | 12.0  0.0  0.0 |
           | 16.0  0.0  1.0 |
       B = | 16.0  0.0  0.0 |
           | 16.0  0.0 -1.0 |
           | 15.0  1.0  0.0 |
           | 13.0  1.0 -2.0 |
           | 10.0  2.0 -3.0 |
```

Output:

```
           |  1.0  1.0  1.0 |
           |  1.0 -1.0  0.0 |
           |  1.0  1.0 -1.0 |
           |  1.0 -1.0  1.0 |
       B = |  1.0  1.0  0.0 |
           |  1.0 -1.0 -1.0 |
           |  1.0  1.0  1.0 |
           |  1.0 -1.0  0.0 |
           |  1.0  1.0 -1.0 |


    INFO = 0
```

Example 2

This example shows how to solve the system ***AX*** = ***B***, where matrix ***A*** is the same positive definite symmetric band matrix factored in Example 2 for DPBTRF in the form ***U***^{T}***U***.

Call Statement and Input:

```
             UPLO  N   K  NRHS A  LDA  B  LDB  INFO
              |    |   |   |   |   |   |   |    |
CALL DPBTRS( 'U' , 9 , 2 , 3 , A , 3 , B , 9 , INFO )

     A = (same as output A in Example 2)
```

```
           |  1.0  3.0  0.0 |
           |  0.0 -6.0  2.0 |
           |  1.0  9.0 -4.0 |
           |  1.0 -9.0  4.0 |
      B = |  1.0  9.0  0.0 |
          |  1.0 -9.0 -4.0 |
          |  1.0  9.0  4.0 |
          |  0.0 -8.0 -1.0 |
          |  2.0  6.0 -2.0
|
```

Output:

```
          |  1.0  1.0  1.0 |
          |  1.0 -1.0  0.0 |
          |  1.0  1.0 -1.0 |
          |  1.0 -1.0  1.0 |
      B = |  1.0  1.0  0.0 |
          |  1.0 -1.0 -1.0 |
          |  1.0  1.0  1.0 |
          |  1.0 -1.0  0.0 |
          |  1.0  1.0 -1.0 |


   INFO = 0
```

Example 3

This example shows how to solve the system ***AX*** = ***B***, where matrix ***A*** is the same positive definite complex Hermitian band matrix factored in Example 3 for ZPBTRF in the form $\boldsymbol{LL}^H$.

Call Statement and Input:

```
             UPLO   N   K  NRHS A  LDA  B  LDB  INFO
              |     |   |   |   |   |   |   |    |
CALL ZPBTRS( 'L' , 6 , 3 , 3 , A , 4 , B , 6 , INFO )

     A = (same as output A in Example 3)
```

```
          |  (  1.0,  1.0)  (  1.0,  -7.0)  (  5.0, -5.0) |
          |  (  1.0,  1.0)  (  9.0, -13.0)  ( 18.0, -4.0) |
          |  (  1.0,  1.0)  ( 17.0, -19.0)  ( 27.0,  0.0) |
      B = |  (  1.0,  1.0)  ( 25.0, -23.0)  ( 35.0,  2.0) |
          |  (  1.0,  1.0)  ( 23.0, -19.0)  ( 28.0,  5.0) |
          |  (  1.0,  1.0)  ( 19.0, -13.0)  ( 18.0, -1.0)
|
```

Output:

```
          |  (  7.0,  33.0)  (  1.0, -1.0)  (  1.0,  1.0) |
          |  ( 19.0,  -1.0)  (  1.0, -1.0)  (  1.0, -1.0) |
          |  (  5.0, -13.0)  (  1.0, -1.0)  (  2.0,  1.0) |
      B = |  (-11.0,  -5.0)  (  1.0, -1.0)  (  2.0, -1.0) |
          |  ( -3.0,   9.0)  (  1.0, -1.0)  (  1.0,  2.0) |
          |  (  5.0,  -1.0)  (  1.0, -1.0)  (  1.0, -2.0) |


   INFO = 0
```

Example 4

This example shows how to solve the system ***AX*** = ***B***, where matrix ***A*** is the same positive definite complex Hermitian band matrix factored in Example 4 for ZPBTRF in the form $\boldsymbol{U}^H\boldsymbol{U}$.

Call Statement and Input:

```
             UPLO   N   K  NRHS A  LDA  B  LDB  INFO
              |     |   |   |   |   |   |   |    |
CALL ZPBTRS( 'U' , 6 , 2 , 3 , A , 3 , B , 6 , INFO )
```

```
     A = (same as output A in Example 4)
```

```
         |  (  1.0,  1.0)  (  3.0,  -3.0)  (  6.0,   0.0) |
         |  (  1.0,  1.0)  (  3.0,  -9.0)  ( 13.0,  -7.0) |
         |  (  1.0,  1.0)  ( 15.0,  -3.0)  ( 22.0,  15.0) |
     B = |  (  1.0,  1.0)  (  3.0, -15.0)  ( 25.0, -14.0) |
         |  (  1.0,  1.0)  ( 15.0,  -1.0)  ( 18.0,  19.0) |
         |  (  1.0,  1.0)  (  3.0, -11.0)  ( 13.0, -14.0)
|
```

Output:

```
          |  (  5.0, 13.0)  (  1.0, -1.0)  (  1.0,  1.0) |
          |  ( -3.0,  7.0)  (  1.0, -1.0)  (  1.0, -1.0) |
          |  ( 11.0, -5.0)  (  1.0, -1.0)  (  2.0,  1.0) |
      B = |  (  3.0,  7.0)  (  1.0, -1.0)  (  2.0, -1.0) |
          |  (  1.0, -5.0)  (  1.0, -1.0)  (  1.0,  2.0) |
          |  (  1.0,  1.0)  (  1.0, -1.0)  (  1.0, -2.0) |


    INFO = 0
```

SGTSV, DGTSV, CGTSV, and ZGTSV (General Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the general tridiagonal system of linear equations ***AX***=***B*** for ***X***, where ***A*** is a general tridiagonal matrix and ***B*** and ***X*** are general matrices.

The matrix ***A*** is factored using Gaussian elimination with partial pivoting.

Table 170: Data Types.

dl, d, du, B	**Subroutine**
Short-precision real	SGTSV$^{\Delta}$
Long-precision real	DGTSV$^{\Delta}$
Short-precision complex	CGTSV$^{\Delta}$
Long-precision complex	ZGTSV$^{\Delta}$
$^{\Delta}$LAPACK	

Syntax

Table 171: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SGTSV | DGTSV | CGTSV | ZGTSV (*n*, *nrhs*, *dl*, *d*, *du*, *b*, *ldb*, *info*)
C and C++	sgtsv | dgtsv | cgtsv | zgtsv (*n*, *nrhs*, *dl*, *d*, *du*, *b*, *ldb*, *info*);
LAPACKE	info = LAPACKE_sgtsv | LAPACKE_dgtsv | LAPACKE_cgtsv | LAPACKE_zgtsv (*matrix_layout*, *n*, *nrhs*, *dl*, *d*, *du*, *b*, *ldb*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

n

is the order of matrix ***A*** and the number of rows in matrix ***B***.

Specified as: an integer; $n \geq 0$.

nrhs

is the number of right-hand sides; that is, the number of columns of matrix ***B*** used in the computation.

Specified as: an integer; $nrhs \geq 0$.

dl

is the array DL, containing the *n* - 1 subdiagonal elements of ***A***.

Specified as: a one-dimensional array of (at least) length *n* - 1, containing numbers of the data type indicated in Table 170 on page 666.

d

is the array D, containing the *n* diagonal elements of ***A***.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 170 on page 666.

du

is the array DU, containing the *n* - 1 superdiagonal elements of ***A***.

Specified as: a one-dimensional array of (at least) length *n* - 1, containing numbers of the data type indicated in Table 170 on page 666.

b

is the general matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of matrix ***B***.

Specified as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 170 on page 666.

ldb

is the leading dimension of the array specified for ***B***.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

info

See "On Return".

On Return

dl

The array DL is overwritten.

Returned as: a one-dimensional array of (at least) length *n* - 1, containing numbers of the data type indicated in Table 170 on page 666.

d

The array D is overwritten.

Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 170 on page 666.

du

The array DU is overwritten.

Returned as: a one-dimensional array of (at least) length *n* - 1, containing numbers of the data type indicated in Table 170 on page 666.

b

If *info* = 0, *b* is the general matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of matrix ***B***.

Returned as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 170 on page 666.

info

has the following meaning:

If *info* = 0, the subroutine completed successfully.

If *info* > 0, *info* is set to the first *i*, where $\boldsymbol{U}_{ii}$ is zero. ***B*** is overwritten; that is, the solution has not been computed.

Returned as:

- For SGTSV, DGTSV, CGTSV, and ZGTSV, returned as: an integer; info ≥ 0.
- For LAPACKE_sgtsv, LAPACKE_dgtsv, LAPACKE_cgtsv, and LAPACKE_zgtsv, returned as an integer function value; info ≥ 0.

Notes:

1. In your C program, argument *info* must be passed by reference.
2. *dl*, *d*, *du*, and B must have no common elements; otherwise, results are unpredictable.
3. For a description of how general tridiagonal matrices are stored, see "General Tridiagonal Matrix" on page 106.
4. The way these subroutines handle singularity differs from LAPACK. Like LAPACK, these subroutines use the info argument to provide information about the singularity of A, but they also provide an error message.

Function

These subroutines solve the general tridiagonal system of linear equations ***AX*** = ***B***, where ***A*** is a general tridiagonal matrix and ***B*** and ***X*** are general matrices.

If *n* is 0 or *nrhs* is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See reference ["9" on page 1253].

Error conditions

Resource Errors

None

Computational Errors

Matrix ***A*** is singular or nearly singular.

- The first column, *i*, of ***L*** with a corresponding zero diagonal element is identified in the computational error message.
- The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2168 is set to be unlimited in the ESSL error option table.

Input-Argument Errors

1. *n* < 0
2. *nrhs* < 0
3. *ldb* ≤ 0
4. *n* > *ldb*

Examples

Example 1

This example shows how to solve the real general tridiagonal system ***AX*** = ***B***, where:

Matrix ***A*** is the same used as input in Example 1 for DGTTRF.

Matrix ***B*** is the same used as input in Example 1 for DGTTRS.

Note: On output, arrays DL, D, and DU are overwritten.

Call Statement and Input:

```
             N   NRHS DL   D   DU   B   LDB  INFO
             |    |   |    |   |    |    |    |
CALL DGTSV(  9 ,  3 , DL , D , DU , B ,  9 , INFO)
```

DL = (same as input DL in Example 1)
D = (same as input D in Example 1)
DU = (same as input DU in Example 1)

B = (same as input B in Example 1)

Output:

```
         ┌                       ┐
         |  0.609   0.478   4.597 |
         |  0.098   0.130  -0.899 |
         | -0.231  -0.641  -2.723 |
         |  0.234   0.312   2.105 |
B    =   |  0.364   0.153   2.516 |
         | -0.017  -0.023  -0.958 |
         | -0.019  -0.359  -0.147 |
         |  0.267   0.357   2.505 |
         |  0.198  -0.070   1.484 |
                                ┘

INFO  =  0
```

Example 2

This example shows how to solve the complex general tridiagonal system
AX=***B***, where:

Matrix ***A*** is the same used as input in Example 2 for ZGTTRF.

Matrix ***B*** is the same used as input in Example 3 for ZGTTRS.

Note: On output, arrays DL, D, and DU are overwritten.

Call Statement and Input:

```
             N   NRHS DL   D   DU   B   LDB  INFO
             |    |   |    |   |    |    |    |
CALL ZGTSV(  4 ,  3 , DL , D , DU , B ,  4 , INFO)
```

DL = (same as input DL in Example 2)
D = (same as input D in Example 2)
DU = (same as input DU in Example 2)

B = (same as input B in Example 3.)

Output:

```
                                                          ┐
        | (-0.247, 0.0) ( 0.119, 0.0)  (0.0,  0.247)|
B    =  | ( 0.311, 0.0) ( 0.220, 0.0)  (0.0, -0.311)|
        | ( 0.357, 0.0) (-0.394, 0.0)  (0.0, -0.357)|
        | (-0.073, 0.0) ( 0.183, 0.0)  (0.0,  0.073)|
                                                          ┘
```

```
INFO  =  0
```

SGTTRF, DGTTRF, CGTTRF, and ZGTTRF (General Tridiagonal Matrix Factorization)

Purpose

These subroutines factor general tridiagonal matrix ***A*** using Gaussian elimination with partial pivoting.

To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGTTRS, DGTTRS, CGTTRS, or ZGTTRS, respectively.

Table 172: Data Types.

dl, d, du, du2	**Subroutine**
Short-precision real	SGTTRF$^{\Delta}$
Long-precision real	DGTTRF$^{\Delta}$
Short-precision complex	CGTTRF$^{\Delta}$
Long-precision complex	ZGTTRF$^{\Delta}$
$^{\Delta}$LAPACK	

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SGTTRS, DGTTRS, CGTTRS, or ZGTTRS, respectively.

Syntax

Table 173: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SGTTRF | DGTTRF | CGTTRF | ZGTTRF (*n*, *dl*, *d*, *du*, *du2*, *ipiv*, *info*)
C and C++	sgttrf | dgttrf | cgttrf | zgttrf (*n*, *dl*, *d*, *du*, *du2*, *ipiv*, *info*);
LAPACKE	info = LAPACKE_sgttrf | LAPACKE_dgttrf | cgttrf | LAPACKE_zgttrf (*matrix_layout*, *n*, *dl*, *d*, *du*, *du2*, *ipiv*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

n
the order of general tridiagonal matrix ***A*** used in the computation.

Specified as: an integer; $n \geq 0$.

dl
is the array DL, containing the n - 1 subdiagonal elements of ***A***.

Specified as: a one-dimensional array of (at least) length n - 1, containing numbers of the data type indicated in Table 172 on page 670.

d
is the array D, containing the n diagonal elements of ***A***.

Specified as: a one-dimensional array of (at least) length n, containing numbers of the data type indicated in Table 172 on page 670.

du
is the array DU, containing the n - 1 superdiagonal elements of ***A***.

Specified as: a one-dimensional array of (at least) length n - 1, containing numbers of the data type indicated in Table 172 on page 670.

du2
See "On Return".

ipiv
See "On Return".

info
See "On Return".

On Return

dl
is the array DL, containing the n - 1 multipliers that define matrix ***L*** from the factorization of ***A***.

Returned as: a one-dimensional array of (at least) length n - 1, containing numbers of the data type indicated in Table 172 on page 670.

d
is the array D, containing the n diagonal elements of matrix ***U*** from the factorization of ***A***.

Returned as: a one-dimensional array of (at least) length n, containing numbers of the data type indicated in Table 172 on page 670.

du
is the array DU, containing the n - 1 elements of the first superdiagonal of matrix ***U*** from the factorization of ***A***.

Returned as: a one-dimensional array of (at least) length n - 1, containing numbers of the data type indicated in Table 172 on page 670.

du2
is the array DU2, containing the n - 2 elements of the second superdiagonal of matrix ***U*** from the factorization of ***A***.

Returned as: a one-dimensional array of (at least) length n - 2, containing numbers of the data type indicated in Table 172 on page 670.

ipiv
Contains the pivot indices.

For $1 \leq \boldsymbol{i} \leq n$, row ***i*** of the matrix was interchanged with row $ipiv_{\boldsymbol{i}}$. $ipiv_{\boldsymbol{i}}$ will always be either ***i*** or ***i*** + 1.

If $ipiv_{\boldsymbol{i}} = \boldsymbol{i}$, no row interchange was required.

Returned as: a one-dimensional integer array of (at least) length n, containing integers; $1 \leq ipiv_{\boldsymbol{i}} \leq n$.

info
has the following meaning:

If *info* = 0, the subroutine completed successfully.

If *info* > 0, *info* is set to the first *i*, where $\boldsymbol{U}_{ii}$ is zero. The factorization has been completed.

Returned as:

- For SGTTRF, DGTTRF, CGTTRF, and ZGTTRF, returned as: an integer; info ≥ 0.
- For LAPACKE_sgttrf, LAPACKE_dgttrf, LAPACKE_cgttrf, and LAPACKE_zgttrf, returned as an integer function value; info ≥ 0.

Notes:

1. In your C program, argument *info* must be passed by reference.
2. *dl*, *d*, *du*, *du2*, and *ipiv* must have no common elements; otherwise results are unpredictable.
3. For a description of how general tridiagonal matrices are stored, see "General Tridiagonal Matrix" on page 106.
4. The way these subroutines handle singularity differs from LAPACK. Like LAPACK, these subroutines use the *info* argument to provide information about the singularity of ***A***, but they also provide an error message.

Function

These subroutines factor general tridiagonal matrix ***A*** using Gaussian elimination with partial pivoting where:

A = ***LU***

In the formula above:

- ***L*** is a product of permutation and unit lower bidiagonal matrices.
- ***U*** is upper triangular with non-zeros in only the main diagonal and first two superdiagonals.

If *n* is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See reference ["9" on page 1253].

Error conditions

Resource Errors
None

Computational Errors
Matrix ***A*** is singular or nearly singular.

- The first column, *i*, of ***L*** with a corresponding zero diagonal element is identified in the computational error message.
- The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2168 is set to be unlimited in the ESSL error option table.

Input-Argument Errors

$n < 0$

Examples

Example 1

This example shows a factorization of a real general tridiagonal matrix of order 9.

Matrix ***A*** is:

```
                    | 1.0  4.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
                    | 3.0  1.0  4.0  0.0  0.0  0.0  0.0  0.0  0.0 |
                    | 0.0  3.0  1.0  4.0  0.0  0.0  0.0  0.0  0.0 |
                    | 0.0  0.0  3.0  1.0  4.0  0.0  0.0  0.0  0.0 |
                    | 0.0  0.0  0.0  3.0  1.0  4.0  0.0  0.0  0.0 |
                    | 0.0  0.0  0.0  0.0  3.0  1.0  4.0  0.0  0.0 |
                    | 0.0  0.0  0.0  0.0  0.0  3.0  1.0  4.0  0.0 |
                    | 0.0  0.0  0.0  0.0  0.0  0.0  3.0  1.0  4.0 |
                    | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  3.0  1.0 |
```

Call Statement and Input:

```
              N   DL   D   DU   DU2   IPIV   INFO
              |    |   |    |    |      |      |
CALL DGTTRF( 9 , DL , D , DU , DU2 , IPIV , INFO )

DL       =  (3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0)

D        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)

DU       =  (4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0)
```

Output:

```
DL       =  (0.333,  0.818, 0.696,  0.908, 0.849, -0.799, 0.625, -0.332 )

D        =  (3.000,  3.666, 3.000,  3.303, 3.532,  3.000, 4.799,  3.000,  4.332 )

DU       =  (1.000, -1.333, 1.000, -2.787, 4.000,  1.000, 3.196,  1.000 )

DU2      =  (4.000,  0.000, 4.000,  0.000, 0.000,  4.000, 0.000 )


IPIV     =  ( 2, 2, 4, 4, 5, 7, 7, 9, 9 )

INFO = 0
```

Example 2

This example shows a factorization of a complex general tridiagonal matrix of order 4.

Matrix ***A*** is:

```
            | (1.0, 1.0) (4.0, 4.0) (0.0, 0.0) (0.0, 0.0) |
            | (3.0, 3.0) (1.0, 1.0) (4.0, 4.0) (0.0, 0.0) |
            | (0.0, 0.0) (3.0, 3.0) (1.0, 1.0) (4.0, 4.0) |
            | (0.0, 0.0) (0.0, 0.0) (3.0, 3.0) (1.0, 1.0) |
```

Call Statement and Input:

```
              N   DL   D   DU   DU2   IPIV   INFO
              |    |   |    |    |      |      |
CALL ZGTTRF( 4 , DL , D , DU , DU2 , IPIV , INFO )

DL       =  ( (3.0, 3.0) (3.0, 3.0) (3.0, 3.0) )

D        =  ( (1.0, 1.0) (1.0, 1.0) (1.0, 1.0) (1.0, 1.0) )

DU       =  ( (4.0, 4.0) (4.0, 4.0) (4.0, 4.0) )
```

Output:

```
DL       =  ( (0.0333, 0.0) (0.0818, 0.0) (0.0696, 0.0) )

D        =  ( (3.0, 3.0) (3.0666, 3.0666) (3.0, 3.0) (3.0303 , 3.0303) )

DU       =  ( (1.0, 1.0) (-1.0333, -1.0333) (1.0, 1.0) )

DU2      =  ( (4.0, 4.0) (0.0, 0.0) )

IPIV     =  ( 2, 2, 4, 4 )

INFO = 0
```

SGTTRS, DGTTRS, CGTTRS, and ZGTTRS (General Tridiagonal Matrix Multiple Right-Hand Side Solve)

Purpose

SGTTRS and DGTTRS solve one of the following systems of equations for multiple right-hand sides:

1. ***AX***=***B***
2. $\boldsymbol{A}^T\boldsymbol{X}=\boldsymbol{B}$

CGTTRS and ZGTTRS solve one of the following systems of equations for multiple right-hand sides:

1. ***AX*** = ***B***
2. $\boldsymbol{A}^T\boldsymbol{X}=\boldsymbol{B}$
3. $\boldsymbol{A}^H\boldsymbol{X}=\boldsymbol{B}$

In the formulas above:

- ***A*** represents the general tridiagonal matrix ***A*** containing the factorization.
- ***B*** represents the general matrix ***B*** containing the right-hand sides in its columns.
- ***X*** represents the general matrix ***B*** containing the solution vectors in its columns.

These subroutines use the results of the factorization of vectors *dl*, *d*, *du*, *du2*, and *ipiv*, produced by a preceding call to SGTTRF, DGTTRF, CGTTRF, and ZGTTRF, respectively.

Table 174: Data Types.

dl, d, du, du2, B	**Subroutine**
Short-precision real	SGTTRS$^{\Delta}$
Long-precision real	DGTTRS$^{\Delta}$
Short-precision complex	CGTTRS$^{\Delta}$
Long-precision complex	ZGTTRS$^{\Delta}$
$^{\Delta}$LAPACK	

Note: The input to these solve subroutines must be the output from the factorization subroutines SGTTRF, DGTTRF, CGTTRF, and ZGTTRF, respectively.

Syntax

Table 175: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SGTTRS | DGTTRS | CGTTRS | ZGTTRS (*trans*, *n*, *nrhs*, *dl*, *d*, *du*, *du2*, *ipiv*, *b*, *ldb*, *info*)
C and C++	sgttrs | dgttrs | cgttrs | zgttrs (*trans*, *n*, *nrhs*, *dl*, *d*, *du*, *du2*, *ipiv*, *b*, *ldb*, *info*);
LAPACKE	info = LAPACKE_sgttrs | LAPACKE_dgttrs | LAPACKE_cgttrs | LAPACKE_zgttrs (*matrix_layout*, *trans*, *n*, *nrhs*, *dl*, *d*, *du*, *du2*, *ipiv*, *b*, *ldb*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

trans

indicates the form of matrix ***A*** to use in the computation, where:

If *trans* = 'N', ***A*** is used in the computation, resulting in solution 1.

If *trans* = 'T', $\boldsymbol{A}^T$ is used in the computation, resulting in solution 2.

If *trans* = 'C', $\boldsymbol{A}^H$ is used in the computation, resulting in solution 3.

Specified as: a single character; *transa* = 'N', 'T', or 'C'.

n

is the order of factored matrix ***A*** and the number of rows in matrix ***B***.

Specified as: an integer; $n \geq 0$.

nrhs

is the number of right-hand sides; that is, the number of columns of matrix ***B*** used in the computation.

Specified as: an integer; $nrhs \geq 0$.

dl

is the array DL, containing the $n - 1$ multipliers that define matrix ***L*** from the factorization of ***A***, produced by a preceding call to SGTTRF, DGTTRF, CGTTRF, or ZGTTRF, respectively.

Specified as: a one-dimensional array of (at least) length $n - 1$, containing numbers of the data type indicated in Table 174 on page 674.

d

is the array D, containing the *n* diagonal elements of matrix ***U*** from the factorization of ***A***, produced by a preceding call to SGTTRF, DGTTRF, CGTTRF, or ZGTTRF, respectively.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 174 on page 674.

du

is the array DU, containing the $n - 1$ elements of the first superdiagonal of matrix ***U*** from the factorization of ***A***, produced by a preceding call to SGTTRF, DGTTRF, CGTTRF, or ZGTTRF, respectively.

Specified as: a one-dimensional array of (at least) length $n - 1$, containing numbers of the data type indicated in Table 174 on page 674.

du2

is the array DU2, containing the $n - 2$ elements of the second superdiagonal of matrix ***U*** from the factorization of ***A***, produced by a preceding call to SGTTRF, DGTTRF, CGTTRF, or ZGTTRF, respectively.

Specified as: a one-dimensional array of (at least) length $n - 2$, containing numbers of the data type indicated in Table 174 on page 674.

ipiv

is the array containing the pivot indices produced by a preceding call to SGTTRF, DGTTRF, CGTTRF, and ZGTTRF, respectively.

Specified as: a one-dimensional array of (at least) length *n*, containing integers; $1 \leq ipiv_i \leq n$.

b

is the general matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of matrix ***B***.

Specified as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 174 on page 674.

ldb

is the leading dimension of the array specified for ***B***.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

info

See "On Return".

On Return

b

is the matrix ***X***, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of ***X***.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 174 on page 674.

info

has the following meaning:

If *info* = 0, the subroutine completed successfully.

Returned as:

- For >SGTTRS, DGTTRS, CGTTRS, and ZGTTRS, returned as: an integer; info ≥ 0.
- For LAPACKE_sgttrs, LAPACKE_dgttrs, LAPACKE_cgttrs, and LAPACKE_zgttrs, returned as an integer function value; info ≥ 0.

Notes:

1. These subroutines accept lowercase letters for the *trans* arguments.
2. In your C program, argument *info* must be passed by reference.
3. *dl*, *d*, *du*, *du2*, *ipiv*, and B must have no common elements; otherwise results are unpredictable.
4. For SGTTRS and DGTTRS, if you specify 'C' for the *trans* argument, it is interpreted as though you specified 'T'.
5. The scalar data specified for input argument *n* must be the same for both _GTTRF and _GTTRS.
6. The array data specified for input arguments *d*, *dl*, *du*, *du2*, and *ipiv* for these subroutines must be the same as the corresponding output arguments for SGTTRF, DGTTRF, CGTTRF, and ZGTTRF, respectively.
7. For a description of how general tridiagonal matrices are stored, see "General Tridiagonal Matrix" on page 106.

Function

One of the following systems of equations is solved for multiple right-hand sides:

1. $\boldsymbol{AX}=\boldsymbol{B}$
2. $\boldsymbol{A}^T\boldsymbol{X}=\boldsymbol{B}$
3. $\boldsymbol{A}^H\boldsymbol{X}=\boldsymbol{B}$ (only for CGTTRS and ZGTTRS)

where ***A*** is a general tridiagonal matrix and ***B*** and ***X*** are general matrices. These subroutines uses the results of the factorization of matrix ***A***, produced by a preceding call to SGTTRF, DGTTRF, CGTTRF or ZGTTRF, respectively. For details on the factorization, see "SGTTRF, DGTTRF, CGTTRF, and ZGTTRF (General Tridiagonal Matrix Factorization)" on page 670.

If *n* is 0 or *nrhs* is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See reference ["9" on page 1253].

Error conditions

Resource Errors

None

Computational Errors

None

Note: If the factorization performed by SGTTRF, DGTTRF, CGTTRF or ZGTTRF failed because a pivot element is zero, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. *trans* ≠ 'N', 'T', or 'C'
2. *n* < 0
3. *nrhs* < 0
4. *ldb* ≤ 0
5. *n* > *ldb*

Examples

Example 1

This example shows how to solve the real general tridiagonal system ***AX*** = ***B***, where matrix ***A*** is the same matrix factored in Example 1 for DGTTRF.

Call Statement and Input:

```
              TRANS  N   NRHS   DL   D   DU   DU2   IPIV   B   LDB  INFO
                |    |    |     |    |   |     |     |     |    |    |
CALL DGTTRS(   'N' , 9 ,  3 ,   DL , D , DU , DU2 , IPIV , B ,  9 , INFO)
```

DL = (same as output DL in Example 1)
D = (same as output D in Example 1)
DU = (same as output DU in Example 1)
DU2 = (same as output DU2 in Example 1)

IPIV = (same as output IPIV in Example 1)

```
         | 1.0   1.0   1.0 |
         | 1.0  -1.0   2.0 |
         | 1.0   1.0   3.0 |
         | 1.0  -1.0   4.0 |
B    =   | 1.0   1.0   5.0 |
         | 1.0  -1.0   6.0 |
         | 1.0   1.0   7.0 |
         | 1.0  -1.0   8.0 |
         | 1.0   1.0   9.0 |
```

Output:

```
         |  0.609   0.478   4.597 |
         |  0.098   0.130  -0.899 |
         | -0.231  -0.641  -2.723 |
         |  0.234   0.312   2.105 |
B    =   |  0.364   0.153   2.516 |
         | -0.017  -0.023  -0.958 |
         | -0.019  -0.359  -0.147 |
         |  0.267   0.357   2.505 |
         |  0.198  -0.070   1.484 |


INFO   =  0
```

Example 2

This example shows how to solve the real general tridiagonal system $\boldsymbol{A}^T\boldsymbol{X}$ = ***B***, where matrix ***A*** is the same matrix factored in Example 1 for DGTTRF.

Call Statement and Input:

```
             TRANS    N   NRHS   DL   D   DU   DU2   IPIV   B   LDB  INFO
               |      |    |     |    |   |     |     |     |    |    |
CALL DGTTRS( 'T' ,  9 ,  3 ,   DL , D , DU , DU2 , IPIV , B ,  9 , INFO)
```

DL = (same as output DL in Example 1)
D = (same as output D in Example 1)
DU = (same as output DU in Example 1)
DU2 = (same as output DU2 in Example 1)

IPIV = (same as output IPIV in Example 1)

```
        | 1.0   1.0   1.0 |
        | 1.0  -1.0   2.0 |
        | 1.0   1.0   3.0 |
        | 1.0  -1.0   4.0 |
B    =  | 1.0   1.0   5.0 |
        | 1.0  -1.0   6.0 |
        | 1.0   1.0   7.0 |
        | 1.0  -1.0   8.0 |
        | 1.0   1.0   9.0 |
```

Output:

```
        |  0.198  -0.070   0.491 |
        |  0.267   0.356   0.170 |
        | -0.019  -0.359  -0.045 |
        | -0.017  -0.023   0.789 |
B    =  |  0.365   0.153   1.130 |
        |  0.234   0.312   0.238 |
        | -0.230  -0.641   0.414 |
        |  0.979   0.130   1.878 |
        |  0.609   0.478   1.489 |


INFO  =  0
```

Example 3

This example shows how to solve the complex general tridiagonal system ***AX***=***B,*** where matrix ***A*** is the same matrix factored in Example 2 for ZGTTRF.

Call Statement and Input:

```
            TRANS    N   NRHS   DL   D   DU   DU2   IPIV   B   LDB  INFO
              |      |    |     |    |   |     |     |     |    |     |
CALL ZGTTRS( 'N' ,   4 ,  3 ,   DL , D , DU , DU2 , IPIV , B ,  4 , INFO)
```

DL = (same as output DL in Example 2)
D = (same as output D in Example 2)
DU = (same as output DU in Example 2)
DU2 = (same as output DU2 in Example 2)

IPIV = (same as output IPIV in Example 2)

```
        | (1.0, 1.0) ( 1.0,  1.0)   (1.0, -1.0) |
B    =  | (1.0, 1.0) (-1.0, -1.0)   (1.0, -1.0) |
        | (1.0, 1.0) ( 1.0,  1.0)   (1.0, -1.0) |
        | (1.0, 1.0) (-1.0, -1.0)   (1.0, -1.0) |
```

Output:

```
        | (-0.247, 0.000)  ( 0.119, 0.000)  (0.000,  0.247) |
B     = | ( 0.311, 0.000)  ( 0.220, 0.000)  (0.000, -0.311) |
        | ( 0.357, 0.000)  (-0.394, 0.000)  (0.000, -0.357) |
        | (-0.073, 0.000)  ( 0.183, 0.000)  (0.000,  0.073) |


INFO  =  0
```

Example 4

This example shows how to solve the complex general tridiagonal system $\boldsymbol{A}^T\boldsymbol{X} = \boldsymbol{B}$, where matrix $\boldsymbol{A}$ is the same matrix factored in Example 2 for ZGTTRF.

Call Statement and Input:

```
             TRANS    N   NRHS   DL   D   DU   DU2   IPIV   B   LDB  INFO
               |      |    |     |    |    |    |      |    |    |     |
CALL ZGTTRS( 'T' ,   4 ,  3 ,   DL , D , DU , DU2 , IPIV , B ,  4 , INFO)
```

DL = (same as output DL in Example 2)
D = (same as output D in Example 2)
DU = (same as output DU in Example 2)
DU2 = (same as output DU2 in Example 2)

IPIV = (same as output IPIV in Example 2)

```
        | (1.0, 1.0) ( 1.0,  1.0)   (1.0, -1.0) |
B    =  | (1.0, 1.0) (-1.0, -1.0)   (1.0, -1.0) |
        | (1.0, 1.0) ( 1.0,  1.0)   (1.0, -1.0) |
        | (1.0, 1.0) (-1.0, -1.0)   (1.0, -1.0) |
```

Output:

```
        | (-0.073, 0.0) (-0.183, 0.0)  (0.0,  0.073)|
B    =  | ( 0.357, 0.0) ( 0.394, 0.0)  (0.0, -0.357)|
        | ( 0.311, 0.0) (-0.220, 0.0)  (0.0, -0.311)|
        | (-0.247, 0.0) (-0.119, 0.0)  (0.0,  0.247)|


INFO  =  0
```

SPTSV, DPTSV, CPTSV, and ZPTSV (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

SPTSV and DPTSV solve the tridiagonal system $\boldsymbol{AX} = \boldsymbol{B}$ for $\boldsymbol{X}$, where $\boldsymbol{X}$ and $\boldsymbol{B}$ are general matrices and $\boldsymbol{A}$ is a positive definite real symmetric matrix stored in LAPACK-symmetric-tridiagonal storage mode.

CPTSV and ZPTSV solve one of the following tridiagonal systems for $\boldsymbol{X}$, where $\boldsymbol{X}$ and $\boldsymbol{B}$ are general matrices and $\boldsymbol{A}$ is a positive definite complex Hermitian matrix stored in LAPACK-complex Hermitian-tridiagonal storage mode:

- If you specify the subdiagonal of $\boldsymbol{A}$ in *e*, then this subroutine solves $\boldsymbol{AX} = \boldsymbol{B}$.
- If you specify the superdiagonal of $\boldsymbol{A}$ in *e*, then this subroutine solves $\boldsymbol{A}^T\boldsymbol{X} = \boldsymbol{B}$.

Table 176: Data Types.

d	*e*, *B*	Subroutine
Short-precision real	Short-precision real	SPTSV$^{\Delta}$
Long-precision real	Long-precision real	DPTSV$^{\Delta}$
Short-precision real	Short-precision complex	CPTSV$^{\Delta}$
Long-precision real	Long-precision complex	ZPTSV$^{\Delta}$
$^{\Delta}$LAPACK		

Syntax

Table 177: Syntax for supported computing languages.

Language	Syntax
Fortran	CALL SPTSV | DPTSV | CPTSV | ZPTSV (*n*, *nrhs*, *d*, *e*, *b*, *ldb*, *info*)
C and C++	sptsv | dptsv | cptsv | zptsv (*n*, *nrhs*, *d*, *e*, *b*, *ldb*, *info*);
LAPACKE	info = LAPACKE_sptsv | LAPACKE_dptsv | LAPACKE_cptsv | LAPACKE_zptsv(*matrix_layout*, *n*, *nrhs*, *d*, *e*, *b*, *ldb*);

On Entry

matrix_layout
indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

n
is the order *n* of tridiagonal matrix ***A***. Specified as: an integer; $n \geq 0$.

nrhs
is the number of right-hand sides; i.e., the number of columns of matrix ***B***. Specified as: an integer; $nrhs \geq 0$.

d
is the vector ***d***, containing the main diagonal of matrix ***A*** in positions 1 through *n* in an array referred to as D. Specified as: a one-dimensional array, of (at least) length *n*, containing numbers of the data type indicated in Table 176 on page 680.

e
is the vector ***e*** containing the subdiagonal or superdiagonal of matrix ***A*** in positions 1 through *n*-1 in an array referred to as E. Specified as: a one-dimensional array, of (at least) length *n*-1, containing numbers of the data type indicated in Table 176 on page 680.

b
is the matrix ***B*** of right-hand side vectors. Specified as the *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 176 on page 680.

ldb
is the leading dimension of the array specified for ***B***. Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

On Return

d

if *info*=0, is the vector ***d***, containing the diagonal D of the factorization of matrix ***A*** in an array referred to as D. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 176 on page 680.

e

if *info*=0, is the vector ***e***, as follows:

For SPTSV and DPTSV

e contains the subdiagonal or superdiagonal elements of the unit lower bidiagonal factor ***L*** in positions 1 through *n*-1 in an array, referred to as E.

For CPTSV and ZPTSV

e contains the following:

- If, on entry, you specified the subdiagonal of matrix ***A*** in *e*, *e* contains the subdiagonal elements of the unit lower bidiagonal factor ***L*** in positions 1 through *n*-1 in an array, referred to as E.
- If, on entry, you specified the superdiagonal of matrix ***A*** in *e*, *e* contains the superdiagonal elements of the unit upper bidiagonal factor ***U*** in positions 1 through *n*-1 in an array, referred to as E.

Returned as: a one-dimensional array of (at least) length *n*-1, containing numbers of the data type indicated in Table 176 on page 680. It has the same length as E on entry.

b

If *info* = 0, *b* is the general matrix ***X***, containing the solutions to the system.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 176 on page 680.

info

has the following meaning:

If *info* = 0, the subroutine completed successfully.

If *info* = *i*, the leading minor of order *i* is not positive definite. The factorization could not be completed and the solution was not computed.

Returned as:

- For SPTSV, DPTSV, CPTSV, and ZPTSV, returned as: an integer; info ≥ 0.
- For LAPACKE_sptsv, LAPACKE_dptsv, LAPACKE_cptsv, and LAPACKE_zptsv, returned as an integer function value; info ≥ 0.

Notes:

1. In your C program, argument *info* must be passed by reference.
2. For a description of how real symmetric tridiagonal matrices are stored in LAPACK-symmetric-tridiagonal storage mode, see "LAPACK-Symmetric-Tridiagonal Storage Mode" on page 108. For a description of how complex Hermitian tridiagonal matrices are stored in LAPACK-complex Hermitian-tridiagonal storage mode, "Complex Hermitian Tridiagonal Storage Representation" on page 109.
3. The way these subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the *info* argument to provide information about the computational error, but they also provide an error message.
4. On both input and output, matrix ***A*** conforms to LAPACK format.

Function

SPTSV and DPTSV solve the tridiagonal system ***AX*** = ***B*** for ***X***, where ***X*** and ***B*** are general matrices and ***A*** is a positive definite real symmetric matrix stored in LAPACK-symmetric-tridiagonal storage mode.

The matrix ***A*** is factored using ***A*** = ***LDL***^{T}.

Note: Because ***A*** is symmetric, this may be considered to be a ***U***^{T}***DU*** factorization as well.

CPTSV and ZPTSV solve one of the following tridiagonal systems for ***X***, where ***X*** and ***B*** are general matrices and ***A*** is a positive definite complex Hermitian matrix stored in LAPACK-complex Hermitian-tridiagonal storage mode:

- If you specify the subdiagonal of ***A*** in *e*, then this subroutine solves ***AX*** = ***B*** and ***A*** = ***LDL***^{H}.
- If you specify the superdiagonal of ***A*** in *e*, then this subroutines solves ***A***^{T}***X*** = ***B*** and ***A*** = ***U***^{H}***DU***.

If *n* = 0, no computation is performed and the subroutine returns after doing some parameter checking. If *n* > 0 and *nrhs* = 0, no solutions are computed and the subroutine returns after factoring the matrix. See references [“9” on page 1253],[“46” on page 1255], and [“75” on page 1257].

Error conditions

Computational Errors

Matrix A is not positive definite. For details, see the description of the *info* argument.

Input-Argument Errors

1. *n* < 0
2. *nrhs* < 0
3. *ldb* ≤ 0
4. *n* > *ldb*

Examples

Example 1

This example shows how to solve the positive definite real symmetric tridiagonal system of linear equations ***AX*** = ***B***, where:

Matrix ***A*** is the same used as input in Example 1 for DPTTRF.

Matrix ***B*** is the same used as input in Example 1 for DPTTRS.

Call Statement and Input:

```
             N  NRHS D   E   B  LDB  INFO
             |   |   |   |   |   |     |
CALL DPTSV(  4 , 2 , D , E , B , 4 , INFO )
```

```
      D = (same as output D in Example 1)

      E = (same as output E in Example 1)

      B = (same as input B in Example 1)
```

Output:

```
D        = ( 1.0, 1.0, 2.0, 0.5 )

E        = ( 1.0, 1.0, 0.5 )


          | 1.0 -1.0 |
      B = | 1.0 -1.0 |
          | 1.0  0.0 |
          | 1.0  1.0 |


   INFO = 0
```

Example 2

This example shows how to solve the positive definite complex Hermitian tridiagonal system of linear equations ***AX*** = ***B***, where:

Matrix ***A*** is the same used as input in Example 2 for ZPTTRF.

Matrix ***B*** is the same used as input in Example 2 for ZPTTRS.

Call Statement and Input:

```
              N NRHS  D   E   B  LDB  INFO
              |   |   |   |   |   |     |
CALL ZPTSV(   4 , 3 , D , E , B , 4 , INFO )
```

```
       D = (same as output D in Example 2)

       E = (same as output E in Example 2)

       B = (same as input B in Example 2)
```

Output:

```
       D = ( 1.0 2.0 3.0 4.0 )

       E = ( ( 1.0, 1.0) ( 1.0, 1.0) ( 1.0, 1.0) )


           | (  1.0,  0.0)  (  1.0,  1.0)  (  1.0, -1.0) |
           | (  1.0,  0.0)  (  1.0,  1.0)  (  1.0, -1.0) |
       B = | (  1.0,  0.0)  (  1.0,  1.0)  (  1.0, -1.0) |
           | (  1.0,  0.0)  (  1.0,  1.0)  (  1.0, -1.0) |


    INFO = 0
```

Example 3

This example shows how to solve the positive definite complex Hermitian tridiagonal system of linear equations $\boldsymbol{A}^T\boldsymbol{X} = \boldsymbol{B}$, where:

Matrix ***A*** is the same used as input in Example 3 for ZPTTRF.

Matrix ***B*** is the same used as input in Example 3 for ZPTTRS.

Call Statement and Input:

```
              N  NRHS D   E   B  LDB  INFO
              |    |  |   |   |   |     |
CALL ZPTSV(   4 , 3 , D , E , B , 4 , INFO )
```

```
       D = (same as output D in Example 3)

       E = (same as output E in Example 3)

       B = (same as input B in Example 3)
```

Output:

```
       D        =  ( 1.0 2.0 3.0 4.0 )

       E        =  ( ( 1.0, -1.0) ( 1.0, -1.0) ( 1.0, -1.0) )



           | ( 3.00, -3.33) (  6.33, -0.33) ( -0.33, -6.33) |
           | ( 0.66,  1.66) ( -1.00,  2.33) (  2.33,  1.00) |
       B = | ( 0.83, -1.50) (  2.33, -0.66) ( -0.66, -2.33) |
           | ( 1.50,  1.00) (  0.50,  2.50) (  2.50, -0.50) |


    INFO = 0
```

SPTTRF, DPTTRF, CPTTRF, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)

Purpose

SPTTRF and DPTTRF factor a positive definite real symmetric tridiagonal matrix stored in LAPACK-symmetric-tridiagonal storage mode:

$\boldsymbol{A} = \boldsymbol{LDL}^{T}$

CPTTRF and ZPTTRF factor a positive definite complex Hermitian tridiagonal matrix stored in LAPACK-complex Hermitian-tridiagonal storage mode:

- If you specify the subdiagonal of ***A*** in vector *e*, then $\boldsymbol{A} = \boldsymbol{LDL}^{H}$
- If you specify the superdiagonal of ***A*** in vector *e*, then $\boldsymbol{A} = \boldsymbol{U}^{H}\boldsymbol{DU}$

To solve the system of equations with one or more right-hand sides, follow the call to SPTTRF, DPTTRF, CPTTRF, or ZPTTRF with a call to SPTTRS, DPTTRS, CPTTRS, or ZPTTRS, respectively.

Table 178: Data Types.

Data Types		
d	***e***	**Subroutine**
Short-precision real	Short-precision real	SPTTRF$^{\Delta}$
Long-precision real	Long-precision real	DPTTRF$^{\Delta}$
Short-precision real	Short-precision complex	CPTTRF$^{\Delta}$
Long-precision real	Long-precision complex	ZPTTRF$^{\Delta}$
$^{\Delta}$LAPACK		

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SPTTRS, DPTTRS, CPTTRS, or ZPTTRS, respectively.

Syntax

Language	Syntax
Fortran	CALL SPTTRF | DPTTRF | CPTTRF | ZPTTRF (*n*, *d*, *e*, *info*)
C and C++	spttrf | dpttrf | cpttrf | zpttrf (*n*, *d*, *e*, *info*);
LAPACKE	info = LAPACKE_spttrf | LAPACKE_dpttrf | LAPACKE_cpttrf | LAPACKE_zpttrf (*matrix_layout*, *n*, *d*, *e*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

n

is the order *n* of tridiagonal matrix ***A***. Specified as: an integer; $n \geq 0$.

d

is the vector ***d***, containing the main diagonal of matrix ***A*** in positions 1 through *n* in an array referred to as D. Specified as: a one-dimensional array, of (at least) length *n*, containing numbers of the data type indicated in Table 178 on page 684.

e

is the vector ***e*** containing the subdiagonal or superdiagonal of matrix ***A*** in positions 1 through *n*-1 in an array referred to as E. Specified as: a one-dimensional array, of (at least) length *n*-1, containing numbers of the data type indicated in Table 178 on page 684.

On Return

d

If *info* = 0, is the vector ***d***, containing the diagonal D of the factorization of matrix ***A*** in an array referred to as D. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 178 on page 684.

e

If *info* = 0, is the vector ***e***, as follows:

For SPTTRF and DPTTRF

e contains the subdiagonal elements of the unit lower bidiagonal factor ***L*** in positions 1 through *n*-1 in an array referred to as E.

For CPTTRF and ZPTTRF

e contains the following:

- If on entry you specified the subdiagonal of matrix ***A*** in *e*, *e* contains the subdiagonal elements of the unit bidiagonal factor ***L*** in positions 1 through *n*-1 in an array, referred to as E.
- If on entry you specified the superdiagonal of matrix ***A*** in *e*, *e* contains the subdiagonal elements of the unit bidiagonal factor ***U*** in positions 1 through *n*-1 in an array, referred to as E.

Returned as: a one-dimensional array of (at least) length *n*-1, containing numbers of the data type indicated in Table 178 on page 684. It has the same length as E on entry.

info

has the following meaning:

If *info* = 0, the subroutine completed successfully.

If *info* = *i*, the leading minor of order *i* is not positive definite, and the factorization could not be completed.

Returned as:

- For SPTTRF, DPTTRF, CPTTRF, and ZPTTRF, returned as: an integer; info ≥ 0.
- For LAPACKE_spttrf, LAPACKE_dpttrf, LAPACKE_cpttrf, and LAPACKE_zpttrf, returned as an integer function value; info ≥ 0.

Notes

1. In your C program, argument *info* must be passed by reference.
2. For a description of how real symmetric tridiagonal matrices are stored in LAPACK-symmetric-tridiagonal storage mode, see "LAPACK-Symmetric-Tridiagonal Storage Mode" on page 108. For a description of how complex Hermitian tridiagonal matrices are stored in LAPACK-complex Hermitian-tridiagonal storage mode, "Complex Hermitian Tridiagonal Storage Representation" on page 109.
3. The way these subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the *info* argument to provide information about the computational error, but they also provide an error message.
4. On both input and output, matrix ***A*** conforms to LAPACK format.

Function

SPTTRF and DPTTRF factor a positive definite real symmetric tridiagonal matrix stored in LAPACK-symmetric-tridiagonal storage mode:

$\boldsymbol{A} = \boldsymbol{LDL}^{\mathrm{T}}$

Note: Because $\boldsymbol{A}$ is symmetric, this may be considered to be a $\boldsymbol{U}^{\mathrm{T}}\boldsymbol{DU}$ factorization as well.

CPTTRF and ZPTTRF factor a positive definite complex Hermitian tridiagonal matrix stored in LAPACK-complex Hermitian-tridiagonal storage mode:

- If you specify the subdiagonal of $\boldsymbol{A}$ in vector *e*, then $\boldsymbol{A} = \boldsymbol{LDL}^{\mathrm{H}}$
- If you specify the superdiagonal of $\boldsymbol{A}$ in vector *e*, then $\boldsymbol{A} = \boldsymbol{U}^{\mathrm{H}}\boldsymbol{DU}$

To solve the system of equations with one or more right-hand sides, follow the call to SPTTRF, DPTTRF, CPTTRF, or ZPTTRF with a call to SPTTRS, DPTTRS, CPTTRS, or ZPTTRS, respectively.

If *n* is 0, no computation is performed and the subroutine returns after doing some parameter checking. See references [“9” on page 1253],[“46” on page 1255], and [“75” on page 1257].

Error conditions

Computational Errors

Matrix $\boldsymbol{A}$ is not positive definite. For details, see the description of the *info* argument.

Input-Argument Errors

1. $n < 0$

Examples

Example 1

This example shows a factorization of the positive definite real symmetric tridiagonal matrix $\boldsymbol{A}$**,** in the form $\boldsymbol{A} = \boldsymbol{LDL}^{\mathrm{T}}$:

```
          | 1.0  1.0  0.0  0.0 |
          | 1.0  2.0  1.0  0.0 |
          | 0.0  1.0  3.0  1.0 |
          | 0.0  0.0  1.0  1.0 |
```

Call Statement and Input:

```
             N   D   E   INFO
             |   |   |    |
CALL DPTTRF( 4 , D , E , INFO )

D        = ( 1.0, 2.0, 3.0, 1.0 )

E        = ( 1.0, 1.0, 1.0 )
```

Output:

```
D        = ( 1.0, 1.0, 2.0, 0.5 )

E        = ( 1.0, 1.0, 0.5 )

INFO = 0
```

Example 2

This example shows a factorization of the positive definite complex Hermitian tridiagonal matrix ***A***, in the form ***A*** = ***LDL***^{H}:

```
           | ( 1.0,  0.0) ( 1.0, -1.0) ( 0.0,  0.0) ( 0.0,  0.0) |
           | ( 1.0,  1.0) ( 4.0,  0.0) ( 2.0, -2.0) ( 0.0,  0.0) |
           | ( 0.0,  0.0) ( 2.0,  2.0) ( 7.0,  0.0) ( 3.0, -3.0) |
           | ( 0.0,  0.0) ( 0.0,  0.0) ( 3.0,  3.0) (10.0,  0.0) |
```

Call Statement and Input:

```
              N   D   E   INFO
              |   |   |    |
CALL ZPTTRF( 4 , D , E , INFO )

D        =  ( 1.0 4.0 7.0 10.0 )

E        =  ( ( 1.0, 1.0) ( 2.0, 2.0) ( 3.0, 3.0) )
```

Output:

```
D        =  ( 1.0 2.0 3.0 4.0 )

E        =  ( ( 1.0, 1.0) ( 1.0, 1.0) ( 1.0, 1.0) )

INFO = 0
```

Example 3

This example shows a factorization of the positive definite complex Hermitian tridiagonal matrix ***A***, in the form ***A*** = ***U***^{H}***DU***:

```
           | ( 1.0,  0.0) ( 1.0, -1.0) ( 0.0,  0.0) ( 0.0,  0.0) |
           | ( 1.0,  1.0) ( 4.0,  0.0) ( 2.0, -2.0) ( 0.0,  0.0) |
           | ( 0.0,  0.0) ( 2.0,  2.0) ( 7.0,  0.0) ( 3.0, -3.0) |
           | ( 0.0,  0.0) ( 0.0,  0.0) ( 3.0,  3.0) (10.0,  0.0) |
```

Call Statement and Input:

```
              N   D   E   INFO
              |   |   |    |
CALL ZPTTRF( 4 , D , E , INFO )

D        =  ( 1.0 4.0 7.0 10.0 )

E        =  ( ( 1.0, -1.0) ( 2.0, -2.0) ( 3.0, -3.0) )
```

Output:

```
D        =  ( 1.0 2.0 3.0 4.0 )

E        =  ( ( 1.0, -1.0) ( 1.0, -1.0) ( 1.0, -1.0) )

INFO = 0
```

SPTTRS, DPTTRS, CPTTRS, and ZPTTRS (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Multiple Right-Hand Solve)

Purpose

SPTTRS and DPTTRS solve the tridiagonal system ***AX*** = ***B*** for ***X***, where ***X*** and ***B*** are general matrices and ***A*** is a positive definite real symmetric matrix.

CPTTRS and ZPTTRS solve one of the following tridiagonal systems for ***X***, where ***X*** and ***B*** are general matrices and ***A*** is a positive definite complex Hermitian matrix.

- If, in the call to CPTTRF or ZPTTRF, you specified the subdiagonal of ***A*** in *e*:
  - If *uplo* = 'L', then this subroutine solves ***AX*** = ***B***.
  - If *uplo* = 'U', then this subroutine solves $\boldsymbol{A}^T\boldsymbol{X} = \boldsymbol{B}$.
- If, in the call to CPTTRF or ZPTTRF, you specified the superdiagonal of ***A*** in *e*:
  - If *uplo* = 'L', then this subroutine solves $\boldsymbol{A}^T\boldsymbol{X} = \boldsymbol{B}$.
  - If *uplo* = 'U', then this subroutine solves ***AX*** = ***B***.

These subroutines use the results of the factorization of matrix A, produced by a preceding call to SPTTRF, DPTTRF, CPTTRF, or ZPTTRF respectively.

Table 179: Data Types.

d	***e, B***	**Subroutine**
Short-precision real	Short-precision real	SPTTRS$^\Delta$
Long-precision real	Long-precision real	DPTTRS$^\Delta$
Short-precision real	Short-precision complex	CPTTRS$^\Delta$
Long-precision real	Long-precision complex	ZPTTRS$^\Delta$
$^\Delta$LAPACK		

Note: The input to these solve subroutines must be the output from the factorization subroutines SPTTRF, DPTTRF, CPTTRF, or ZPTTRF respectively.

Syntax

Table 180: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SPTTRS | DPTTRS (*n*, *nrhs*, *d*, *e*, *b*, *ldb*, *info*)
	CALL CPTTRS | ZPTTRS (*uplo*, *n*, *nrhs*, *d*, *e*, *b*, *ldb*, *info*)
C and C++	spttrs | dpttrs(*n*, *nrhs*, *d*, *e*, *b*, *ldb*, *info*);
	cpttrs | zpttrs (*uplo*, *n*, *nrhs*, *d*, *e*, *b*, *ldb*, *info*);

LAPACKE	info = LAPACKE_spttrs | LAPACKE_dpttrs (*matrix_layout*, *n*, *nrhs*, *d*, *e*, *b*, *ldb*);
	info = LAPACKE_cpttrs | LAPACKE_zpttrs (*matrix_layout*, *uplo*, *n*, *nrhs*, *d*, *e*, *b*, *ldb*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

uplo

indicates whether *e* is the subdiagonal of the unit bidiagonal lower triangular factor ***L*** or superdiagonal of the unit bidiagonal upper triangular factor ***U***:

If *uplo* = 'L', *e* is the subdiagonal of the unit bidiagonal lower triangular factor ***L***.

If *uplo* = 'U', *e* is the superdiagonal of the unit bidiagonal upper triangular factor ***U***.

Specified as: a single character. It must be 'L' or 'U'.

n
is the order *n* of tridiagonal matrix ***A***. Specified as: an integer; $n \geq 0$.

nrhs
is the number of right-hand sides; i.e., the number of columns of matrix ***B***. Specified as: an integer; $nrhs \geq 0$.

d
is the vector ***d***, containing part of the factorization of matrix ***A*** from SPTTRF, DPTTRF, CPTTRF, or ZPTTRF, respectively, in an array, referred to as D. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 179 on page 688.

e

For SPTTRS and DPTTRS
is the vector ***e***, containing the subdiagonal elements of the unit bidiagonal factor ***L*** in positions 1 through *n*-1 in an array, referred to as E.

For CPTTRS and ZPTTRS
is the vector ***e***, containing the subdiagonal or superdiagonal of matrix ***A*** in positions 1 through *n*-1 in an array, referred to as E.

- If *uplo* = 'L', ***e*** contains he subdiagonal elements of the unit bidiagonal factor ***L*** in positions 1 through *n*-1 in an array, referred to as E.
- If *uplo* = 'U', ***e*** contains the superdiagonal elements of the unit bidiagonal factor ***U*** in positions 1 through *n*-1 in an array, referred to as E.

Specified as: a one-dimensional array, of (at least) length *n*-1, containing numbers of the data type indicated in Table 179 on page 688.

b
is the matrix ***B*** of right-hand side vectors. Specified as the *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 179 on page 688.

ldb
is the leading dimension of the array specified for ***B***. Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

On Return

b
is the general matrix ***X***, containing the solutions to the system.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 179 on page 688.

info
info has the following meaning:

If *info* = 0, the solve completed successfully.

Returned as:

- For SPTTRS, DPTTRS, CPTTRS, and ZPTTRS, returned as: an integer; info ≥ 0.
- For LAPACKE_spttrs, LAPACKE_dpttrs, LAPACKE_cpttrs, and LAPACKE_zpttrs, returned as an integer function value; info ≥ 0.

Notes:

1. In your C program, argument *info* must be passed by reference.
2. All subroutines accept lowercase letters for the *uplo* argument.
3. For a description of how real symmetric tridiagonal matrices are stored in LAPACK-symmetric-tridiagonal storage mode, see "LAPACK-Symmetric-Tridiagonal Storage Mode" on page 108. For a description of how complex Hermitian tridiagonal matrices are stored in LAPACK-complex Hermitian-tridiagonal storage mode, "Complex Hermitian Tridiagonal Storage Representation" on page 109.

4. The scalar data specified for input argument *n* for these subroutines must be the same as the corresponding input argument specified for SPTTRF, DPTTRF, CPTTRF, or ZPTTRF, respectively.
5. The array data specified for input arguments *d* and *e* for these subroutines must be the same as the corresponding output arguments for SPTTRF, DPTTRF, CPTTRF, and ZPTTRF, respectively.

Function

SPTTRS and DPTTRS solve the tridiagonal system $\boldsymbol{AX} = \boldsymbol{B}$ for $\boldsymbol{X}$, where $\boldsymbol{X}$ and $\boldsymbol{B}$ are general matrices and $\boldsymbol{A}$ is a positive definite real symmetric matrix.

CPTTRS and ZPTTRS solve one of the following tridiagonal systems for $\boldsymbol{X}$, where $\boldsymbol{X}$ and $\boldsymbol{B}$ are general matrices and $\boldsymbol{A}$ is a positive definite complex Hermitian matrix.

- If, in the call to CPTTRF or ZPTTRF, you specified the subdiagonal of $\boldsymbol{A}$ in *e*:
  - If *uplo* = 'L', then this subroutine solves $\boldsymbol{AX} = \boldsymbol{B}$.
  - If *uplo* = 'U', then this subroutine solves $\boldsymbol{A}^T\boldsymbol{X} = \boldsymbol{B}$.
- If, in the call to CPTTRF or ZPTTRF, you specified the superdiagonal of $\boldsymbol{A}$ in *e*:
  - If *uplo* = 'L', then this subroutine solves $\boldsymbol{A}^T\boldsymbol{X} = \boldsymbol{B}$.
  - If *uplo* = 'U', then this subroutine solves $\boldsymbol{AX} = \boldsymbol{B}$.

These subroutines use the results of the factorization of matrix A, produced by a preceding call to SPTTRF, DPTTRF, CPTTRF, or ZPTTRF respectively. For a description of how $\boldsymbol{A}$ is factored, see "SPTTRF, DPTTRF, CPTTRF, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)" on page 684.

If *n* or *nrhs* is 0, no computation is performed. See references ["9" on page 1253] and ["46" on page 1255].

Error conditions

Computational Errors

None

Note: If the factorization performed by SPTTRF, DPTTRF, CPTTRF, or ZPTTRF failed because matrix $\boldsymbol{A}$ was not positive definite, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. $n < 0$
3. $nrhs < 0$
4. $ldb \le 0$
5. $n > ldb$

Examples

Example 1

This example shows how to solve the system of linear equations $\boldsymbol{AX} = \boldsymbol{B}$ where positive definite real symmetric tridiagonal matrix $\boldsymbol{A}$ is the same matrix factored in Example 1 for DPTTRF in the form $\boldsymbol{LDL}^T$.

Call Statement and Input:

```
              N  NRHS D   E   B  LDB  INFO
              |   |   |   |   |   |     |
CALL DPTTRS(  4 , 2 , D , E , B , 4 , INFO )
```

```
      D = (same as output D in Example 1)
      E = (same as output E in Example 1)


          | 2.0 -2.0 |
      B = | 4.0 -3.0 |
          | 5.0  0.0 |
          | 2.0  1.0 |
```

Output:

```
          | 1.0 -1.0 |
      B = | 1.0 -1.0 |
          | 1.0  0.0 |
          | 1.0  1.0 |


   INFO = 0
```

Example 2

This example shows how to solve the system of linear equations ***AX*** = ***B*** where positive definite complex Hermitian tridiagonal matrix ***A*** is the same matrix factored in Example 2 for ZPTTRF in the form ***LDL***^{H}.

Call Statement and Input:

```
              UPLO N  NRHS D   E   B  LDB  INFO
               |   |   |   |   |   |   |    |
CALL ZPTTRS(  'L', 4 , 3 , D , E , B , 4 , INFO )
```

```
      D = (same as output D in Example 2)

      E = (same as output E in Example 2)


          | (  2.0, -1.0)  (  3.0,  1.0)  (  1.0,  -3.0) |
      B = | (  7.0, -1.0)  (  8.0,  6.0)  (  6.0,  -8.0) |
          | ( 12.0, -1.0)  ( 13.0, 11.0)  ( 11.0, -13.0) |
          | ( 13.0,  3.0)  ( 10.0, 16.0)  ( 16.0, -10.0) |
```

Output:

```
          | (  1.0,  0.0)  (  1.0,  1.0)  (  1.0, -1.0) |
          | (  1.0,  0.0)  (  1.0,  1.0)  (  1.0, -1.0) |
      B = | (  1.0,  0.0)  (  1.0,  1.0)  (  1.0, -1.0) |
          | (  1.0,  0.0)  (  1.0,  1.0)  (  1.0, -1.0) |


   INFO = 0
```

Example 3

This example shows how to solve the system of linear equations ***A***^{T}***X*** = ***B*** where positive definite complex Hermitian tridiagonal matrix ***A*** is the same matrix factored in Example 2 for ZPTTRF in the form ***LDL***^{H}.

Call Statement and Input:

```
              UPLO N  NRHS D   E   B  LDB  INFO
               |   |   |   |   |   |   |    |
CALL ZPTTRS(  'U', 4 , 3 , D , E , B , 4 , INFO )
```

```
      D = (same as output D in Example 2)

      E = (same as output E in Example 2)
```

```
        | (  2.0, -1.0)  (  3.0,  1.0)  (  1.0,  -3.0) |
    B = | (  7.0, -1.0)  (  8.0,  6.0)  (  6.0,  -8.0) |
        | ( 12.0, -1.0)  ( 13.0, 11.0)  ( 11.0, -13.0) |
        | ( 13.0,  3.0)  ( 10.0, 16.0)  ( 16.0, -10.0) |
```

Output:

```
        | (  3.00, -3.33)  (  6.33, -0.33)  ( -0.33, -6.33) |
        | (  0.66,  1.66)  ( -1.00,  2.33)  (  2.33,  1.00) |
    B = | (  0.83, -1.50)  (  2.33, -0.66)  ( -0.66, -2.33) |
        | (  1.50,  1.00 )  ( 0.50,  2.50)  (  2.50, -0.50) |


   INFO = 0
```

Example 4

This example shows how to solve the system of linear equations ***AX*** = ***B*** where positive definite complex Hermitian tridiagonal matrix ***A*** is the same matrix factored in Example 3 for ZPTTRF in the form $\boldsymbol{U}^{H}\boldsymbol{DU}$.

Call Statement and Input:

```
              UPLO N  NRHS D   E   B  LDB  INFO
               |   |   |   |   |   |   |    |
CALL ZPTTRS(  'U', 4 , 3 , D , E , B , 4 , INFO )
```

```
    D = (same as output D in Example 3)

    E = (same as output E in Example 3)


        | (  2.0, -1.0)  (  3.0,  1.0)  (  1.0,  -3.0) |
    B = | (  7.0, -1.0)  (  8.0,  6.0)  (  6.0,  -8.0) |
        | ( 12.0, -1.0)  ( 13.0, 11.0)  ( 11.0, -13.0) |
        | ( 13.0,  3.0)  ( 10.0, 16.0)  ( 16.0, -10.0) |
```

Output:

```
        | ( 1.0, 0.0) ( 1.0, 1.0) ( 1.0, -1.0) |
        | ( 1.0, 0.0) ( 1.0, 1.0) ( 1.0, -1.0) |
    B = | ( 1.0, 0.0) ( 1.0, 1.0) ( 1.0, -1.0) |
        | ( 1.0, 0.0) ( 1.0, 1.0) ( 1.0, -1.0) |


   INFO = 0
```

Example 5

This example shows how to solve the system of linear equations $\boldsymbol{A}^{T}\boldsymbol{X}$ = ***B*** where positive definite complex Hermitian tridiagonal matrix ***A*** is the same matrix factored in Example 3 for ZPTTRF in the form $\boldsymbol{U}^{H}\boldsymbol{DU}$.

Call Statement and Input:

```
              UPLO N  NRHS D   E   B  LDB  INFO
               |   |   |   |   |   |   |    |
CALL ZPTTRS(  'U', 4 , 3 , D , E , B , 4 , INFO )
```

```
    D = (same as output D in Example 3)

    E = (same as output E in Example 3)


        | (  2.0, -1.0)  (  3.0,  1.0)  (  1.0,  -3.0) |
    B = | (  7.0, -1.0)  (  8.0,  6.0)  (  6.0,  -8.0) |
        | ( 12.0, -1.0)  ( 13.0, 11.0)  ( 11.0, -13.0) |
        | ( 13.0,  3.0)  ( 10.0, 16.0)  ( 16.0, -10.0) |
```

Output:

```
           | (  3.00, -3.33)  (  6.33, -0.33)  ( -0.33, -6.33) |
           | (  0.66,  1.66)  ( -1.00,  2.33)  (  2.33,  1.00) |
       B = | (  0.83, -1.50)  (  2.33, -0.66)  ( -0.66, -2.33) |
           | (  1.50,  1.00 )  ( 0.50,  2.50)  (  2.50, -0.50) |


    INFO = 0
```

SGBF and DGBF (General Band Matrix Factorization)

Purpose

These subroutines factor general band matrix ***A***, stored in general-band storage mode, using Gaussian elimination. To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGBS or DGBS, respectively.

Table 181: Data Types.

Data Types	
A	**Subroutine**
Short-precision real	SGBF
Long-precision real	DGBF

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SGBS and DGBS, respectively.

Syntax

Language	Syntax
Fortran	CALL SGBF | DGBF (*agb*, *lda*, *n*, *ml*, *mu*, *ipvt*)
C and C++	sgbf | dgbf (*agb*, *lda*, *n*, *ml*, *mu*, *ipvt*);

On Entry

agb
is the general band matrix ***A*** of order *n*, stored in general-band storage mode, to be factored. It has an upper band width *mu* and a lower band width *ml*. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 181 on page 693, where *lda* ≥ 2*ml*+*mu*+16.

lda
is the leading dimension of the array specified for *agb*. Specified as: an integer; *lda* > 0 and *lda* ≥ 2*ml* +*mu*+16.

n
is the order of the matrix ***A***. Specified as: an integer; *n* > *ml* and *n* > *mu*.

ml
is the lower band width *ml* of the matrix ***A***. Specified as: an integer; 0 ≤ *ml* < *n*.

mu
is the upper band width *mu* of the matrix ***A***. Specified as: an integer; 0 ≤ *mu* < *n*.

ipvt

See On Return.

On Return

agb

is the transformed matrix ***A*** of order *n*, containing the results of the factorization. See "Function" on page 694. Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 181 on page 693.

ipvt

is the integer vector ***ipvt*** of length *n*, containing the pivot information necessary to construct matrix ***L*** from the information contained in the output array *agb*. Returned as: a one-dimensional array of (at least) length *n*, containing integers.

Notes

1. ***ipvt*** is not a permutation vector in the strict sense. It is used to record column interchanges in ***L*** due to partial pivoting and to improve performance.
2. The entire *lda* by *n* array specified for *agb* must remain unchanged between calls to the factorization and solve subroutines.
3. This subroutine can be used for tridiagonal matrices (*ml* = *mu* = 1); however, the tridiagonal subroutines SGTF/DGTF and SGTS/DGTS are faster.
4. For a description of how a general band matrix is stored in general-band storage mode in an array, see "General Band Matrix" on page 94.

Function

The general band matrix ***A***, stored in general-band storage mode, is factored using Gaussian elimination with partial pivoting to compute the ***LU*** factorization of ***A***, where:

ipvt is a vector containing the pivoting information.
L is a unit lower triangular band matrix.
U is an upper triangular band matrix.

The transformed matrix ***A*** contains ***U*** in packed format, along with the multipliers necessary to construct, with the help of ***ipvt***, a matrix ***L***, such that ***A*** = ***LU***. This factorization can then be used by SGBS or DGBS, respectively, to solve the system of equations. See reference ["48" on page 1255].

Error conditions

Resource Errors

Unable to allocate internal work area.

Computational Errors

Matrix ***A*** is singular.

- One or more columns of ***L*** and the corresponding diagonal of ***U*** contain all zeros (all columns of ***L*** are checked). The last column, *i*, of ***L*** with a corresponding ***U*** = 0 diagonal element is identified in the computational error message.
- The return code is set to 1.
- *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2103 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

Input-Argument Errors

1. $lda \le 0$
2. $ml < 0$

3. *ml* ≥ *n*
4. *mu* < 0
5. *mu* ≥ *n*
6. *lda* < 2*ml*+*mu*+16

Examples

Example

This example shows a factorization of a general band matrix ***A*** of order 9, with a lower band width of 2 and an upper band width of 3. On input matrix ***A*** is:

```
             | 1.0  1.0  1.0  1.0  0.0  0.0   0.0   0.0   0.0 |
             | 1.0  1.0  1.0  1.0  1.0  0.0   0.0   0.0   0.0 |
             | 4.0  1.0  1.0  1.0  1.0  1.0   0.0   0.0   0.0 |
             | 0.0  5.0  1.0  1.0  1.0  1.0   1.0   0.0   0.0 |
             | 0.0  0.0  6.0  1.0  1.0  1.0   1.0   1.0   0.0 |
             | 0.0  0.0  0.0  7.0  1.0  1.0   1.0   1.0   1.0 |
             | 0.0  0.0  0.0  0.0  8.0  1.0   1.0   1.0   1.0 |
             | 0.0  0.0  0.0  0.0  0.0  9.0   1.0   1.0   1.0 |
             | 0.0  0.0  0.0  0.0  0.0  0.0  10.0  11.0  12.0 |
```

Matrix ***A*** is stored in general-band storage mode in the two-dimensional array AGB of size LDA by N, where LDA = 2*ml*+*mu*+16 = 23. The array AGB is declared as `AGB(1:23,1:9)`.

Note: Matrix ***A*** is the same matrix used in the examples in subroutines SGEF and DGEF (see Example 1) and SGEFCD and DGEFCD (see Example).

Call Statement and Input:

```
           AGB  LDA   N   ML  MU  IPVT )
            |    |    |   |   |    |
CALL SGBF( AGB , 23 , 9 , 2 , 3 , IPVT )
```

```
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  1.0000  1.0000  1.0000   1.0000   1.0000   1.0000 |
         | 0.0000  0.0000  1.0000  1.0000  1.0000  1.0000   1.0000   1.0000   1.0000 |
         | 0.0000  1.0000  1.0000  1.0000  1.0000  1.0000   1.0000   1.0000   1.0000 |
         | 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000   1.0000   1.0000  12.0000 |
         | 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000   1.0000  11.0000   0.0000 |
         | 4.0000  5.0000  6.0000  7.0000  8.0000  9.0000  10.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
AGB   =  | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
         | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 |
```

Output:

```
        | 0.0000  0.0000  0.0000  0.0000   0.0000   1.0000   1.0000   1.0000   1.0000 |
        | 0.0000  0.0000  0.0000  0.0000   1.0000   1.0000   1.0000   1.0000   1.0000 |
        | 0.0000  0.0000  0.0000  1.0000   1.0000   1.0000   1.0000   1.0000   1.0000 |
        | 0.0000  0.0000  1.0000  1.0000   1.0000   1.0000   1.0000   1.0000  12.0000 |
        | 0.0000  1.0000  1.0000  1.0000   1.0000   1.0000   1.0000  11.0000   0.3111 |
        | 0.2500  0.2000  0.1600  0.1400   0.1250   0.1100   0.1000   5.5380  -325.00 |
        | 0.0000  0.1500  0.0000  0.0714   0.0000  -0.0556  -0.0306   0.9385   0.0000 |
        | 0.2500  0.1500  0.1000  0.0714  -0.0714  -0.0694  -0.0194   0.0000   0.0000 |
        | 0.2500  0.0000  0.1000  0.0000   0.0536   0.0000   0.0000   0.0000   0.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 |
AGB  =  | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 |
        | 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 |
```

```
IPVT     =  (2, -65534, -131070, -196606, -262142, -327678, -327678,
             -327680, -327680)
```

SGBS and DGBS (General Band Matrix Solve)

Purpose

These subroutines solve the system ***Ax*** = ***b*** for ***x***, where ***A*** is a general band matrix, and ***x*** and ***b*** are vectors. They use the results of the factorization of matrix ***A***, produced by a preceding call to SGBF or DGBF, respectively.

Table 182: Data Types.	
Data Types	
A, ***b***, ***x***	**Subroutine**
Short-precision real	SGBS
Long-precision real	DGBS

Note: The input to these solve subroutines must be the output from the factorization subroutines SGBF and DGBF, respectively.

Syntax

Language	**Syntax**
Fortran	CALL SGBS | DGBS (*agb*, *lda*, *n*, *ml*, *mu*, *ipvt*, *bx*)
C and C++	sgbs | dgbs (*agb*, *lda*, *n*, *ml*, *mu*, *ipvt*, *bx*);

On Entry

agb

is the factorization of general band matrix ***A***, produced by a preceding call to SGBF or DGBF. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 182 on page 696, where *lda* ≥ 2*ml*+*mu*+16.

lda

is the leading dimension of the array specified for *agb*. Specified as: an integer; *lda* > 0 and *lda* ≥ 2*ml* +*mu*+16.

n

is the order of the matrix ***A***. Specified as: an integer; *n* > *ml* and *n* > *mu*.

ml

is the lower band width *ml* of the matrix ***A***. Specified as: an integer; 0 ≤ *ml* < *n*.

mu

is the upper band width *mu* of the matrix ***A***. Specified as: an integer; 0 ≤ *mu* < *n*.

ipvt

is the integer vector ***ipvt*** of length *n*, produced by a preceding call to SGBF or DGBF. It contains the pivot information necessary to construct matrix ***L*** from the information contained in the array specified for *agb*.

Specified as: a one-dimensional array of (at least) length *n*, containing integers.

bx

is the vector ***b*** of length *n*, containing the right-hand side of the system. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 182 on page 696.

On Return

bx

is the solution vector ***x*** of length *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 182 on page 696.

Notes

1. The scalar data specified for input arguments *lda*, *n*, *ml*, and *mu* for these subroutines must be the same as that specified for SGBF and DGBF, respectively.
2. The array data specified for input arguments *agb* and *ipvt* for these subroutines must be the same as the corresponding output arguments for SGBF and DGBF, respectively.
3. The entire *lda* by *n* array specified for *agb* must remain unchanged between calls to the factorization and solve subroutines.
4. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
5. This subroutine can be used for tridiagonal matrices (*ml* = *mu* = 1); however, the tridiagonal subroutines, SGTF/DGTF and SGTS/DGTS, are faster.
6. For a description of how a general band matrix is stored in general-band storage mode in an array, see "General Band Matrix" on page 94.

Function

The real system ***Ax*** = ***b*** is solved for ***x***, where ***A*** is a real general band matrix, stored in general-band storage mode, and ***x*** and ***b*** are vectors. These subroutines use the results of the factorization of matrix ***A***, produced by a preceding call to SGBF or DGBF, respectively. The transformed matrix ***A***, used by this computation, consists of the upper triangular matrix ***U*** and the multipliers necessary to construct ***L*** using ***ipvt***, as defined in "Function" on page 694. See reference ["48" on page 1255].

Error conditions

Computational Errors

Note: If the factorization performed by SGBF or DGBF failed due to a singular matrix argument, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. $lda \le 0$
2. $ml < 0$
3. $ml \ge n$
4. $mu < 0$
5. $mu \ge n$
6. $lda < 2ml+mu+16$

Examples

Example

This example shows how to solve the system ***Ax*** = ***b***, where general band matrix ***A*** is the same matrix factored in Example for SGBF and DGBF. The input for AGB and IPVT in this example is the same as the output for that example.

Call Statement and Input:

```
           AGB  LDA   N   ML  MU  IPVT   BX
            |    |    |   |   |    |     |
CALL SGBS( AGB , 23 , 9 , 2 , 3 , IPVT , BX )
```

```
IPVT = (2, -65534, -131070, -196606, -262142, -327678, -327678,
-327680, -327680)
BX = (4.0000, 5.0000, 9.0000, 10.0000, 11.0000, 12.0000,
12.0000, 12.0000, 33.0000)
```

AGB = (same as output AGB in Example)

Output:

```
BX       =  (1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
             0.9999, 1.0001)
```

SPBF, DPBF, SPBCHF, and DPBCHF (Positive Definite Symmetric Band Matrix Factorization)

Purpose

These subroutines factor positive definite symmetric band matrix ***A***, stored in lower-band-packed storage mode, using:

- Gaussian elimination for SPBF and DPBF
- Cholesky factorization for SPBCHF and DPBCHF

To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SPBS, DPBS, SPBCHS, or DPBCHS, respectively.

Table 183: Data Types.

Data Types	
A	**Subroutine**
Short-precision real	SPBF and SPBCHF
Long-precision real	DPBF and DPBCHF

Note:

1. The output from these factorization subroutines should be used only as input to the solve subroutines SPBS, DPBS, SPBCHS, and DPBCHS, respectively.
2. For optimal performance:
   - For wide band widths, use _PBCHF.
   - For narrow band widths, use either _PBF or _PBCHF.
   - For very narrow band widths:
     - Use either SPBF or SPBCHF.
     - Use DPBF.

Syntax

Language	Syntax
Fortran	CALL SPBF | DPBF | SPBCHF | DPBCHF (*apb*, *lda*, *n*, *m*)
C and C++	spbf | dpbf | spbchf | dpbchf (*apb*, *lda*, *n*, *m*);

On Entry

apb
is the positive definite symmetric band matrix ***A*** of order *n*, stored in lower-band-packed storage mode, to be factored. It has a half band width of *m*. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 183 on page 698. See "Notes" on page 699.

lda
is the leading dimension of the array specified for *apb*. Specified as: an integer; *lda* > 0 and *lda* > *m*.

n
is the order *n* of matrix ***A***. Specified as: an integer; *n* > *m*.

m
is the half band width of the matrix ***A***. Specified as: an integer; 0 ≤ *m* < *n*.

On Return

apb
is the transformed matrix ***A*** of order *n*, containing the results of the factorization. See "Function" on page 699. Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 183 on page 698. For further details, see "Notes" on page 699.

Notes

1. These subroutines can be used for tridiagonal matrices (*m* = 1); however, the tridiagonal subroutines, SPTF/DPTF and SPTS/DPTS, are faster.
2. For SPBF and DPBF when *m* > 0, location APB(2,*n*) is sometimes set to 0.
3. For a description of how a positive definite symmetric band matrix is stored in lower-band-packed storage mode in an array, see "Positive Definite Symmetric Band Matrix" on page 101.

Function

The positive definite symmetric band matrix ***A***, stored in lower-band-packed storage mode, is factored using Gaussian elimination in SPBF and DPBF and Cholesky factorization in SPBCHF and DPBCHF. The transformed matrix ***A*** contains the results of the factorization in packed format. This factorization can then be used by SPBS, DPBS, SPBCHS, and DPBCHS, respectively, to solve the system of equations.

For performance reasons, divides are done in a way that reduces the effective exponent range for which DPBF works properly, when processing narrow band widths; therefore, you may want to scale your problem.

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors

1. Matrix ***A*** is not positive definite (for SPBF and DPBF).
   - One or more elements of ***D*** contain values less than or equal to 0; all elements of ***D*** are checked. The index *i* of the last nonpositive element encountered is identified in the computational error message.
   - The return code is set to 1.
   - *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2104 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see Chapter 4, “Coding programs,” on page 125.
2. Matrix ***A*** is not positive definite (for SPBCHF and DPBCHF).
   - The leading minor of order *i* has a nonpositive determinant. The order *i* is identified in the computational error message.
   - The return code is set to 1.
   - *i* can be determined at run time by using the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2115 in the ESSL error option table; otherwise, the default value causes your program to be terminate when this error occurs. For details, see Chapter 4, “Coding programs,” on page 125.

Input-Argument Errors

1. $lda \le 0$
2. $m < 0$
3. $m \ge n$
4. $m \ge lda$

Examples

Example 1

This example shows a factorization of a real positive definite symmetric band matrix ***A*** of order 9, using Gaussian elimination, where on input, matrix ***A*** is:

```
                | 1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0 |
                | 1.0  2.0  2.0  1.0  0.0  0.0  0.0  0.0  0.0 |
                | 1.0  2.0  3.0  2.0  1.0  0.0  0.0  0.0  0.0 |
                | 0.0  1.0  2.0  3.0  2.0  1.0  0.0  0.0  0.0 |
                | 0.0  0.0  1.0  2.0  3.0  2.0  1.0  0.0  0.0 |
                | 0.0  0.0  0.0  1.0  2.0  3.0  2.0  1.0  0.0 |
                | 0.0  0.0  0.0  0.0  1.0  2.0  3.0  2.0  1.0 |
                | 0.0  0.0  0.0  0.0  0.0  1.0  2.0  3.0  2.0 |
                | 0.0  0.0  0.0  0.0  0.0  0.0  1.0  2.0  3.0 |
```

and on output, matrix ***A*** is:

```
                | 1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0 |
                | 1.0  1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0 |
                | 1.0  1.0  1.0  1.0  1.0  0.0  0.0  0.0  0.0 |
                | 0.0  1.0  1.0  1.0  1.0  1.0  0.0  0.0  0.0 |
                | 0.0  0.0  1.0  1.0  1.0  1.0  1.0  0.0  0.0 |
                | 0.0  0.0  0.0  1.0  1.0  1.0  1.0  1.0  0.0 |
                | 0.0  0.0  0.0  0.0  1.0  1.0  1.0  1.0  1.0 |
                | 0.0  0.0  0.0  0.0  0.0  1.0  1.0  1.0  1.0 |
                | 0.0  0.0  0.0  0.0  0.0  0.0  1.0  1.0  1.0 |
```

where array location APB(2,9) is set to 0.0.

Call Statement and Input:

```
            APB  LDA  N   M
             |    |   |   |
CALL SPBF( APB , 3 , 9 , 2 )


        | 1.0  2.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0 |
APB  =  | 1.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0   .  |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0   .    .  |
```

Output:

```
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
APB  =  | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  0.0 |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0   .    .  |
```

Example 2

This example shows a Cholesky factorization of the same matrix used in Example 1.

Call Statement and Input:

```
              APB  LDA  N   M
               |    |   |   |
CALL SPBCHF( APB , 3 , 9 , 2 )
```

APB = (same as input APB in Example 1)

Output:

```
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
APB  =  | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0   .  |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0   .    .  |
```

SPBS, DPBS, SPBCHS, and DPBCHS (Positive Definite Symmetric Band Matrix Solve)

Purpose

These subroutines solve the system $\boldsymbol{Ax} = \boldsymbol{b}$ for $\boldsymbol{x}$, where $\boldsymbol{A}$ is a positive definite symmetric band matrix, and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors. They use the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SPBF, DPBF, SPBCHF, and DPBCHF, respectively, where:

- Gaussian elimination was used by SPBF and DPBF.
- Cholesky factorization was used by SPBCHF and DPBCHF.

Table 184: Data Types.

Data Types	
A*, *b*, *x	**Subroutine**
Short-precision real	SPBS and SPBCHS

Table 184: Data Types. (continued)	
Data Types	
Long-precision real	DPBS and DPBCHS

Note:

1. The input to these solve subroutines must be the output from the factorization subroutines SPBF, DPBF, SPBCHF, and DPBCHF, respectively.
2. For performance tradeoffs, see "SPBF, DPBF, SPBCHF, and DPBCHF (Positive Definite Symmetric Band Matrix Factorization)" on page 698.

Syntax

Language	Syntax
Fortran	CALL SPBS | DPBS | SPBCHS | DPBCHS (*apb*, *lda*, *n*, *m*, *bx*)
C and C++	spbs | dpbs | spbchs | dpbchs (*apb*, *lda*, *n*, *m*, *bx*);

On Entry

apb
is the factorization of matrix ***A***, produced by a preceding call to SPBF or DPBF. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 184 on page 701. See "Notes" on page 702.

lda
is the leading dimension of the array specified for *apb*. Specified as: an integer; *lda* > 0 and *lda* > *m*.

n
is the order *n* of matrix ***A***. Specified as: an integer; *n* > *m*.

m
is the half band width of the matrix ***A***. Specified as: an integer; 0 ≤ *m* < *n*.

bx
is the vector ***b*** of length *n*, containing the right-hand side of the system. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 184 on page 701.

On Return

bx
is the solution vector ***x*** of length *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 184 on page 701.

Notes

1. The scalar data specified for input arguments *lda*, *n*, and *m* for these subroutines must be the same as that specified for SPBF, DPBF, SPBCHF, and DPBCHF, respectively.
2. The array data specified for input argument *apb* for these subroutines must be the same as the corresponding output argument for SPBF, DPBF, SPBCHF, and DPBCHF, respectively.
3. These subroutines can be used for tridiagonal matrices (*m* = 1); however, the tridiagonal subroutines, SPTF/DPTF and SPTS/DPTS, are faster.
4. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
5. For a description of how a positive definite symmetric band matrix is stored in lower-band-packed storage mode in an array, see "Positive Definite Symmetric Band Matrix" on page 101.

Function

The system ***Ax*** = ***b*** is solved for ***x***, where ***A*** is a positive definite symmetric band matrix, stored in lower-band-packed storage mode, and ***x*** and ***b*** are vectors. These subroutines use the results of the factorization of matrix ***A***, produced by a preceding call to SPBF, DPBF, SPBCHF, or DPBCHF, respectively.

Error conditions

Computational Errors

None

Note: If the factorization subroutine resulted in a nonpositive definite matrix, error 2104 for SPBF and DPBF or error 2115 for SPBCHF and DPBCHF, results of these subroutines may be unpredictable.

Input-Argument Errors

1. *lda* ≤ 0
2. *m* < 0
3. *m* ≥ *n*
4. *m* ≥ *lda*

Examples

Example 1

This example shows how to solve the system ***Ax*** = ***b***, where matrix ***A*** is the same matrix factored in the Example 1 for SPBF and DPBF, using Gaussian elimination.

Call Statement and Input:

```
           APB  LDA  N   M   BX
            |    |   |   |   |
CALL SPBS( APB , 3 , 9 , 2 , BX )
```

APB = (same as output APB in Example 1)
BX = (3.0, 6.0, 9.0, 9.0, 9.0, 9.0, 9.0, 8.0, 6.0)

Output:

```
BX       =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

This example shows how to solve the system ***Ax*** = ***b***, where matrix ***A*** is the same matrix factored in the Example 2 for SPBCHF and DPBCHF, using Cholesky factorization.

Call Statement and Input:

```
             APB  LDA  N   M   BX
              |    |   |   |   |
CALL SPBCHS( APB , 3 , 9 , 2 , BX )
```

APB = (same as output APB in Example 2)
BX = (3.0, 6.0, 9.0, 9.0, 9.0, 9.0, 9.0, 8.0, 6.0)

Output:

```
BX       =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

SGTF and DGTF (General Tridiagonal Matrix Factorization)

Purpose

These subroutines compute the standard Gaussian factorization with partial pivoting for tridiagonal matrix ***A***, stored in tridiagonal storage mode. To solve a tridiagonal system with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGTS or DGTS, respectively.

Table 185: Data Types.

Data Types	
c, d, e, f	**Subroutine**
Short-precision real	SGTF
Long-precision real	DGTF

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SGTS and DGTS, respectively.

Syntax

Language	**Syntax**
Fortran	CALL SGTF | DGTF (*n*, *c*, *d*, *e*, *f*, *ipvt*)
C and C++	sgtf | dgtf (*n*, *c*, *d*, *e*, *f*, *ipvt*);

On Entry

n
: is the order *n* of tridiagonal matrix ***A***. Specified as: an integer; $n \geq 0$.

c
: is the vector ***c***, containing the lower subdiagonal of matrix ***A*** in positions 2 through *n* in an array, referred to as C. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 185 on page 704.

d
: is the vector ***d***, containing the main diagonal of matrix ***A***, in positions 1 through *n* in an array, referred to as D. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 185 on page 704.

e
: is the vector ***e***, containing the upper subdiagonal of matrix ***A***, in positions 1 through *n*-1 in an array, referred to as E. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 185 on page 704.

f
: See On Return.

ipvt
: See On Return.

On Return

c
: is the vector ***c***, containing part of the factorization of matrix ***A*** in positions 1 through *n* in an array, referred to as C. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 185 on page 704.

d
: is the vector ***d***, containing part of the factorization of matrix ***A*** in an array, referred to as D. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 185 on page 704.

e
: is the vector ***e***, containing part of the factorization of the matrix ***A*** in positions 1 through *n* in an array, referred to as E. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 185 on page 704.

f
: is the vector ***f***, containing part of the factorization of matrix ***A*** in the first *n* positions in an array, referred to as F. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 185 on page 704.

ipvt
: is the integer vector ***ipvt*** of length *n*, containing the pivot information. Returned as: a one-dimensional array of (at least) length *n*, containing integers.

Notes

1. For a description of how tridiagonal matrices are stored, see "General Tridiagonal Matrix" on page 106.
2. ***ipvt*** is not a permutation vector in the strict sense. It is used to record column interchanges in the tridiagonal matrix due to partial pivoting.
3. The factorization matrix ***A*** is stored in nonstandard format.

Function

The standard Gaussian elimination with partial pivoting of tridiagonal matrix ***A*** is computed. The factorization is returned by overwriting input arrays C, D, and E, and by writing into output array F, along with pivot information in vector ***ipvt***. This factorization can then be used by SGTS or DGTS, respectively, to solve tridiagonal systems of linear equations. See references ["53" on page 1256], ["65" on page 1256], ["66" on page 1256], and ["111" on page 1259]. If *n* is 0, no computation is performed.

Error conditions

Computational Errors

Matrix ***A*** is singular or nearly singular.

- A pivot element has a value that cannot be reciprocated or is equal to 0. The index *i* of the element is identified in the computational error message.
- The return code is set to 1.
- *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2105 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

Input-Argument Errors

n < 0

Examples

Example

This example shows how to factor the following tridiagonal matrix ***A*** of order 4:

```
                    | 2.0  2.0  0.0  0.0 |
                    | 1.0  3.0  2.0  0.0 |
                    | 0.0  1.0  3.0  2.0 |
                           | 0.0  0.0  1.0  3.0 |
```

Call Statement and Input:

```
           N   C   D   E   F   IPVT
           |   |   |   |   |    |
CALL DGTF( 4 , C , D , E , F , IPVT )

C        =  ( . , 1.0, 1.0, 1.0)
D        =  (2.0, 3.0, 3.0, 3.0)
E        =  (2.0, 2.0, 2.0, . )
```

Output:

```
C        =  ( . , -0.5, -0.5, -0.5)
D        =  (-0.5, -0.5, -0.5, -0.5)
E        =  (2.0, 2.0, 2.0, . )
IPVT     =  (X'00', X'00', X'00', X'00')
```

Notes :

1. F is stored in an internal format and is passed unchanged to the solve subroutine.
2. A "." means you do not have to store a value in that position in the array. However, these storage positions are required and may be overwritten during the computation.

SGTS and DGTS (General Tridiagonal Matrix Solve)

Purpose

These subroutines solve a tridiagonal system of linear equations using the factorization of tridiagonal matrix *A*, stored in tridiagonal storage mode, produced by SGTF or DGTF, respectively.

Table 186: Data Types.

Data Types	
c, ***d***, ***e***, ***f***, ***b***, ***x***	**Subroutine**
Short-precision real	SGTS
Long-precision real	DGTS

Note: The input to these solve subroutines must be the output from the factorization subroutines SGTF and DGTF, respectively.

Syntax

Language	**Syntax**
Fortran	CALL SGTS | DGTS (*n*, *c*, *d*, *e*, *f*, *ipvt*, *bx*)
C and C++	sgts | dgts (*n*, *c*, *d*, *e*, *f*, *ipvt*, *bx*);

On Entry

n

is the order *n* of tridiagonal matrix ***A***. Specified as: an integer; $n \geq 0$.

c

is the vector ***c***, containing part of the factorization of matrix ***A*** from SGTF or DGTF, respectively, in an array, referred to as C. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 186 on page 706.

d
: is the vector ***d***, containing part of the factorization of matrix ***A*** from SGTF or DGTF, respectively, in an array, referred to as D. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 186 on page 706.

e
: is the vector ***e***, containing part of the factorization of matrix ***A*** from SGTF or DGTF, respectively, in an array, referred to as E. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 186 on page 706.

f
: is the vector ***f***, containing part of the factorization of matrix ***A*** from SGTF or DGTF, respectively, in an array, referred to as F. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 186 on page 706.

ipvt
: is the integer vector ***ipvt*** of length *n*, containing the pivot information, produced by a preceding call to SGTF and DGTF, respectively. Specified as: a one-dimensional array of (at least) length *n*, containing integers.

bx
: is the vector ***b*** of length *n*, containing the right-hand side of the system in the first *n* positions in an array, referred to as BX. Specified as: a one-dimensional array of (at least) length *n*+1, containing numbers of the data type indicated in Table 186 on page 706. For details on specifying the length, see "Notes" on page 707.

On Return

bx
: is the solution vector ***x*** (at least) of length *n*, containing the solution of the tridiagonal system in the first *n* positions in an array, referred to as BX. Returned as: a one-dimensional array, of (at least) length (*n*+1), containing numbers of the data type indicated in Table 186 on page 706. For details about the length, see "Notes" on page 707.

Notes

1. For a description of how tridiagonal matrices are stored, see "General Tridiagonal Matrix" on page 106.
2. Array BX can have a length of *n* if memory location BX(*n*+1) is addressable—that is, not in read-protected storage. If it is in read-protected storage, array BX must have a length of *n*+1. In both cases, the vector ***b*** (on input) and vector ***x*** (on output) reside in positions 1 through *n* in array BX. Array location BX(*n*+1) is not altered by these subroutines.

Function

Given the factorization produced by SGTF or DGTF, respectively, these subroutines use the standard forward elimination and back substitution to solve the tridiagonal system ***Ax*** = ***b***, where ***A*** is a general tridiagonal matrix. See references ["53" on page 1256], ["65" on page 1256], ["66" on page 1256], and ["111" on page 1259].

Error conditions

Computational Errors
: None

Input-Argument Errors
: $n < 0$

Examples

Example

This example solves the tridiagonal system ***Ax*** = ***b,*** where matrix ***A*** is the same matrix factored in Example for SGTF and DGTF, and where:

```
b = (4.0, 6.0, 6.0, 4.0)
x = (1.0, 1.0, 1.0, 1.0)
```

Call Statement and Input:

```
           N   C   D   E   F   IPVT   BX
           |   |   |   |   |    |     |
CALL DGTS( 4 , C , D , E , F , IPVT , BX )
```

C = (same as output C in Example)
D = (same as output D in Example)
E = (same as output E in Example)
F = (same as output F in Example)
IPVT = (same as output IPVT in Example)
BX = (4.0, 6.0, 6.0, 4.0, .)

Output:

```
BX       =  (1.0, 1.0, 1.0, 1.0, . )
```

SGTNP, DGTNP, CGTNP, and ZGTNP (General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting)

Purpose

These subroutines solve the tridiagonal system ***Ax*** = ***b*** using Gaussian elimination, where tridiagonal matrix ***A*** is stored in tridiagonal storage mode.

Table 187: Data Types.

Data Types	
c, ***d***, ***e***, ***b***, ***x***	**Subroutine**
Short-precision real	SGTNP
Long-precision real	DGTNP
Short-precision complex	CGTNP
Long-precision complex	ZGTNP

Note: In general, these subroutines provide better performance than the _GTNPF and _GTNPS subroutines; however, in the following instances, you get better performance by using _GTNPF and _GTNPS:

- For small *n*
- When performing a single factorization followed by multiple solves

Syntax

Language	**Syntax**
Fortran	CALL SGTNP | DGTNP | CGTNP | ZGTNP (*n*, *c*, *d*, *e*, *bx*)

Language	Syntax
C and C++	sgtnp | dgtnp | cgtnp | zgtnp (*n*, *c*, *d*, *e*, *bx*);

On Entry

n

is the order *n* of tridiagonal matrix ***A***. Specified as: an integer; $n \geq 0$.

c

is the vector ***c***, containing the lower subdiagonal of matrix ***A*** in positions 2 through *n* in an array, referred to as C. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 187 on page 708. On output, C is overwritten; that is, the original input is not preserved.

d

is the vector ***d***, containing the main diagonal of matrix ***A*** in positions 1 through *n* in an array, referred to as D. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 187 on page 708. On output, D is overwritten; that is, the original input is not preserved.

e

is the vector ***e***, containing the upper subdiagonal of matrix ***A*** in positions 1 through *n*-1 in an array, referred to as E. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 187 on page 708. On output, E is overwritten; that is, the original input is not preserved.

bx

is the vector ***b***, containing the right-hand side of the system in the first *n* positions in an array, referred to as BX. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 187 on page 708.

On Return

bx

is the solution vector ***x*** of length *n*, containing the solution of the tridiagonal system in the first *n* positions in an array, referred to as BX. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 187 on page 708.

Notes

For a description of how tridiagonal matrices are stored, see “General Tridiagonal Matrix” on page 106.

Function

The solution of the tridiagonal system ***Ax*** = ***b*** is computed by Gaussian elimination.

No pivoting is done. Therefore, these subroutines should not be used when pivoting is necessary to maintain the numerical accuracy of the solution. Overflow may occur if small main diagonal elements are generated. Underflow or accuracy loss may occur if large main diagonal elements are generated.

For performance reasons, complex divides are done without scaling. Computing the inverse in this way restricts the range of numbers for which the ZGTNP subroutine works properly.

For performance reasons, divides are done in a way that reduces the effective exponent range for which DGTNP and ZGTNP work properly; therefore, you may want to scale your problem, such that the diagonal elements are close to 1.0 for DGTNP and (1.0, 0.0) for ZGTNP.

Error conditions

Computational Errors

None

Input-Argument Errors

$n < 0$

Examples

Example 1

This example shows a factorization of the real tridiagonal matrix ***A,*** of order 4:

```
                         | 7.0  4.0  0.0   0.0 |
                         | 1.0  8.0  5.0   0.0 |
                         | 0.0  2.0  9.0   6.0 |
                         | 0.0  0.0  3.0  10.0 |
```

It then finds the solution of the tridiagonal system ***Ax*** = ***b***, where ***b*** is:

```
                         (11.0, 14.0, 17.0, 13.0)
```

and ***x*** is:

```
                         (1.0, 1.0, 1.0, 1.0)
```

On output, arrays C, D, and E are overwritten.

Call Statement and Input:

```
            N   C   D   E   BX
            |   |   |   |   |
CALL DGTNP( 4 , C , D , E , BX )

C        =  ( . , 1.0, 2.0, 3.0)
D        =  (7.0, 8.0, 9.0, 10.0)
E        =  (4.0, 5.0, 6.0, . )
BX       =  (11.0, 14.0, 17.0, 13.0)
```

Output:

```
BX       =  (1.0, 1.0, 1.0, 1.0)
```

Example 2

This example shows a factorization of the complex tridiagonal matrix ***A,*** of order 4:

```
         | (7.0,  7.0)  (4.0,  4.0)  (0.0,  0.0)   (0.0,  0.0)   |
         | (1.0,  1.0)  (8.0,  8.0)  (5.0,  5.0)   (0.0,  0.0)   |
         | (0.0,  0.0)  (2.0,  2.0)  (9.0,  9.0)   (6.0,  6.0)   |
         | (0.0,  0.0)  (0.0,  0.0)  (3.0,  3.0)  (10.0, 10.0)   |
```

It then finds the solution of the tridiagonal system ***Ax*** = ***b***, where ***b*** is:

```
          ((-11.0,19.0), (-14.0,50.0), (-17.0,93.0), (-13.0,85.0))
```

and ***x*** is:

```
          ((1.0,-1.0), (2.0,-2.0), (3.0,-3.0), (4.0,-4.0))
```

On output, arrays C, D, and E are overwritten.

Call Statement and Input:

```
            N   C   D   E   BX
            |   |   |   |   |
CALL ZGTNP( 4 , C , D , E , BX )

C        =  ( . , (1.0, 1.0), (2.0, 2.0), (3.0, 3.0))
```

```
D        =  ((7.0, 7.0), (8.0, 8.0), (9.0, 9.0), (10.0, 10.0))
E        =  ((4.0, 4.0), (5.0, 5.0), (6.0, 6.0), . )
BX       =  ((-11.0, 19.0), (-14.0, 50.0), (-17.0, 93.0), (-13.0, 85.0))
```

Output:

```
BX       =  ((0.0, 1.0), (1.0, 2.0), (2.0, 3.0), (3.0, 4.0))
```

SGTNPF, DGTNPF, CGTNPF, and ZGTNPF (General Tridiagonal Matrix Factorization with No Pivoting)

Purpose

These subroutines factor tridiagonal matrix ***A***, stored in tridiagonal storage mode, using Gaussian elimination. To solve a tridiagonal system of linear equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGTNPS, DGTNPS, CGTNPS, or ZGTNPS, respectively.

Table 188: Data Types.

Data Types	
c, ***d***, ***e***	**Subroutine**
Short-precision real	SGTNPF
Long-precision real	DGTNPF
Short-precision complex	CGTNPF
Long-precision complex	ZGTNPF

Note:

1. The output from these factorization subroutines should be used only as input to the solve subroutines SGTNPS, DGTNPS, CGTNPS, and ZGTNPS, respectively.
2. In general, the _GTNP subroutines provide better performance than the _GTNPF and _GTNPS subroutines; however, in the following instances, you get better performance by using _GTNPF and _GTNPS:
   - For small *n*
   - When performing a single factorization followed by multiple solves

Syntax

Language	**Syntax**
Fortran	CALL SGTNPF | DGTNPF | CGTNPF | ZGTNPF (*n*, *c*, *d*, *e*, *iopt*)
C and C++	sgtnpf | dgtnpf | cgtnpf | zgtnpf (*n*, *c*, *d*, *e*, *iopt*);

On Entry

n

is the order *n* of tridiagonal matrix ***A***. Specified as: an integer; $n \geq 0$.

c

is the vector ***c***, containing the lower subdiagonal of matrix ***A*** in positions 2 through *n* in an array, referred to as C. Specified as: a one-dimensional array, of (at least) length n, containing numbers of the data type indicated in Table 188 on page 711.

d
is the vector ***d***, containing the main diagonal of matrix ***A*** in positions 1 through *n* in an array, referred to as D. Specified as: a one-dimensional array, of (at least) length *n*, containing numbers of the data type indicated in Table 188 on page 711.

e
is the vector ***e***, containing the upper subdiagonal of matrix ***A*** in positions 1 through *n*-1 in an array, referred to as E. Specified as: a one-dimensional array, of (at least) length *n*, containing numbers of the data type indicated in Table 188 on page 711.

iopt
indicates the type of computation to be performed, where:

If *iopt* = 0 or 1, Gaussian elimination is used to factor the matrix.

Specified as: an integer; *iopt* = 0 or 1.

On Return

c
is the vector ***c***, containing part of the factorization of matrix ***A*** in positions 1 through *n* in an array, referred to as C. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 188 on page 711.

d
is the vector ***d***, containing part of the factorization of matrix ***A*** in an array, referred to as D. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 188 on page 711.

e
is the vector ***e***, containing part of the factorization of matrix ***A*** in positions 1 through *n* in an array, referred to as E. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 188 on page 711. It has the same length as E on entry.

Notes

For a description of how tridiagonal matrices are stored, see "General Tridiagonal Matrix" on page 106.

Function

The factorization of a diagonally-dominant tridiagonal matrix ***A*** is computed using Gaussian elimination, This factorization can then be used by SGTNPS, DGTNPS, CGTNPS, or ZGTNPS respectively, to solve the tridiagonal systems of linear equations. See reference ["93" on page 1258].

No pivoting is done by these subroutines. Therefore, these subroutines should not be used when pivoting is necessary to maintain the numerical accuracy of the solution. Overflow may occur if small main diagonal elements are generated. Underflow or accuracy loss may occur if large main diagonal elements are generated.

For performance reasons, complex divides are done without scaling. Computing the inverse in this way restricts the range of numbers for which ZGTNPF works properly.

For performance reasons, divides are done in a way that reduces the effective exponent range for which DGTNPF and ZGTNPF work properly; therefore, you may want to scale your problem, such that the diagonal elements are close to 1.0 for DGTNPF and (1.0, 0.0) for ZGTNPF.

Error conditions

Computational Errors
None

Input-Argument Errors

1. $n < 0$
2. *iopt* ≠ 0 or 1

Examples

Example 1

This example shows a factorization of the tridiagonal matrix ***A,*** of order 4:

```
                         | 1.0  1.0  0.0  0.0 |
                         | 1.0  2.0  1.0  0.0 |
                         | 0.0  1.0  3.0  1.0 |
                         | 0.0  0.0  1.0  1.0 |
```

Call Statement and Input:

```
             N   C   D   E   IOPT
             |   |   |   |    |
CALL DGTNPF( 4 , C , D , E ,  0   )

C        =  ( . , 1.0, 1.0, 1.0)
D        =  (1.0, 2.0, 3.0, 1.0)
E        =  (1.0, 1.0, 1.0, .  )
```

Output:

```
C        =  ( . , -1.0, -1.0, 1.0)
D        =  (-1.0, -1.0, -1.0, -1.0)
E        =  (1.0, 1.0, -1.0, . )
```

Example 2

This example shows a factorization of the tridiagonal matrix ***A,*** of order 4:

```
             | (7.0, 7.0) (4.0, 4.0) (0.0, 0.0)   (0.0, 0.0) |
             | (1.0, 1.0) (8.0, 8.0) (5.0, 5.0)   (0.0, 0.0) |
             | (0.0, 0.0) (2.0, 2.0) (9.0, 9.0)   (6.0, 6.0) |
             | (0.0, 0.0) (0.0, 0.0) (3.0, 3.0) (10.0, 10.0) |
```

Call Statement and Input:

```
             N   C   D   E   IOPT
             |   |   |   |    |
CALL ZGTNPF( 4 , C , D , E ,  0   )

C        =  ( . , (1.0, 1.0), (2.0, 2.0), (3.0, 3.0))
D        =  ((7.0, 7.0), (8.0, 8.0), (9.0, 9.0), (10.0, 10.0))
E        =  ((4.0, 4.0), (5.0, 5.0), (6.0, 6.0), . )
```

Output:

```
C        =  ( . , (-0.142, 0.0), (-0.269, 0.0), (3.0, 3.0))
D        =  ((-0.0714, 0.0714), (-0.0673, 0.0673), (-0.0854, 0.0854),
             (-0.05, 0.05))
E        =  ((4.0, 4.0), (5.0, 5.0), (-0.6, 0.0), . )
```

Notes :

1. A "**.**" means you do not have to store a value in that position in the array. However, these storage positions are required and may be overwritten during the computation.

SGTNPS, DGTNPS, CGTNPS, and ZGTNPS (General Tridiagonal Matrix Solve with No Pivoting)

Purpose

These subroutines solve a tridiagonal system of equations using the factorization of matrix ***A***, stored in tridiagonal storage mode, produced by SGTNPF, DGTNPF, CGTNPF, or ZGTNPF, respectively.

Table 189: Data Types.

Data Types	
c, ***d***, ***e***, ***b***, ***x***	**Subroutine**
Short-precision real	SGTNPS
Long-precision real	DGTNPS
Short-precision complex	CGTNPS
Long-precision complex	ZGTNPS

Note: The input to these solve subroutines must be the output from the factorization subroutines SGTNPF, DGTNPF, CGTNPF, and ZGTNPF, respectively.

Syntax

Language	**Syntax**
Fortran	CALL SGTNPS | DGTNPS | CGTNPS | ZGTNPS (*n*, *c*, *d*, *e*, *bx*)
C and C++	sgtnps | dgtnps | cgtnps | zgtnps (*n*, *c*, *d*, *e*, *bx*);

On Entry

n

is the order *n* of tridiagonal matrix ***A***. Specified as: an integer; $n \geq 0$.

c

is the vector ***c***, containing part of the factorization of matrix ***A*** from SGTNPF, DGTNPF, CGTNPF, and ZGTNPF, respectively, in an array, referred to as C. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 189 on page 714.

d

is the vector ***d***, containing part of the factorization of matrix ***A*** from SGTNPF, DGTNPF, CGTNPF, and ZGTNPF, respectively, in an array, referred to as D. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 189 on page 714.

e

is the vector ***e***, containing part of the factorization of matrix ***A*** from SGTNPF, DGTNPF, CGTNPF, and ZGTNPF, respectively, in an array, referred to as E. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 189 on page 714.

bx

is the vector ***b***, containing the right-hand side of the system in the first *n* positions in an array, referred to as BX. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 189 on page 714.

On Return

bx

is the solution vector ***x*** of length *n*, containing the solution of the tridiagonal system in the first *n* positions in an array, referred to as BX. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 189 on page 714.

Notes

For a description of how tridiagonal matrices are stored, see "General Tridiagonal Matrix" on page 106.

Function

The solution of tridiagonal system ***Ax*** = ***b*** is computed using the factorization produced by SGTNPF, DGTNPF, CGTNPF, or ZGTNPF, respectively. The factorization is based on Gaussian elimination. See reference ["93" on page 1258].

Error conditions

Computational Errors
None

Input-Argument Errors
$n < 0$

Examples

Example 1

This example finds the solution of tridiagonal system ***Ax*** = ***b***, where matrix ***A*** is the same matrix factored in Example 1 for SGTNPF and DGTNPF. ***b*** is:

```
                    (2.0, 4.0, 5.0, 2.0)
```

and ***x*** is:

```
                    (1.0, 1.0, 1.0, 1.0)
```

Call Statement and Input:

```
             N   C   D   E   BX
             |   |   |   |   |
CALL DGTNPS( 4 , C , D , E , BX )
```

C = (same as output C in Example 1)
D = (same as output D in Example 1)
E = (same as output E in Example 1)
BX = (2.0, 4.0, 5.0, 2.0)

Output:

```
BX       =  (1.0, 1.0, 1.0, 1.0)
```

Example 2

This example finds the solution of tridiagonal system ***Ax*** = ***b***, where matrix ***A*** is the same matrix factored in Example 2 for CGTNPF and ZGTNPF. ***b*** is:

```
          ((-11.0,19.0), (-14.0,50.0), (-17.0,93.0), (-13.0,85.0))
```

and ***x*** is:

```
          ((0.0,1.0), (1.0,2.0), (2.0,3.0), (3.0,4.0))
```

Call Statement and Input:

```
              N   C   D   E   BX
              |   |   |   |   |
CALL ZGTNPS( 4 , C , D , E , BX )
```

C = (same as output C in Example 2)
D = (same as output D in Example 2)
E = (same as output E in Example 2)
BX = ((-11.0, 19.0), (-14.0, 50.0), (-17.0, 93.0), (-13.0, 8))

Output:

```
BX       =  ((0.0, 1.0), (1.0, 2.0), (2.0, 3.0), (3.0, 4.0))
```

SPTF and DPTF (Positive Definite Symmetric Tridiagonal Matrix Factorization)

Purpose

These subroutines factor symmetric tridiagonal matrix *A*, stored in symmetric-tridiagonal storage mode, using Gaussian elimination. To solve a tridiagonal system of linear equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SPTS or DPTS, respectively.

Table 190: Data Types.

Data Types	
c*, *d	**Subroutine**
Short-precision real	SPTF
Long-precision real	DPTF

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SPTS and DPTS, respectively.

Syntax

Language	**Syntax**
Fortran	CALL SPTF | DPTF (*n*, *c*, *d*, *iopt*)
C and C++	sptf | dptf (*n*, *c*, *d*, *iopt*);

On Entry

n
is the order *n* of tridiagonal matrix *A*. Specified as: an integer; *n* ≥ 0.

c
is the vector ***c***, containing the off-diagonal of matrix ***A*** in positions 2 through *n* in an array, referred to as C. Specified as: a one-dimensional array, of (at least) length *n*, containing numbers of the data type indicated in Table 190 on page 716.

d
is the vector ***d***, containing the main diagonal of matrix ***A*** in positions 1 through *n* in an array referred to as D. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 190 on page 716.

iopt
indicates the type of computation to be performed, where:

If *iopt* = 0 or 1, Gaussian elimination is used to factor the matrix.

Specified as: an integer; *iopt* = 0 or 1.

On Return

c

is the vector ***c***, containing part of the factorization of matrix ***A*** in an array, referred to as C. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 190 on page 716.

d

is the vector ***d***, containing part of the factorization of matrix ***A*** in positions 1 through *n* in an array, referred to as D. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 190 on page 716. It has the same length as D on entry.

Notes

For a description of how positive definite symmetric tridiagonal matrices are stored, see "Positive Definite Symmetric Tridiagonal Matrix" on page 109.

Function

The factorization of positive definite symmetric tridiagonal matrix ***A*** is computed using Gaussian elimination. This factorization can then be used by SPTS or DPTS, respectively, to solve the tridiagonal systems of linear equations. See reference ["93" on page 1258].

No pivoting is done. Therefore, these subroutines should not be used when pivoting is necessary to maintain the numerical accuracy of the solution. Overflow may occur if small pivots are generated.

For performance reasons, divides are done in a way that reduces the effective exponent range for which DPTF works properly; therefore, you may want to scale your problem, such that the diagonal elements are close to 1.0 for DPTF.

Error conditions

Computational Errors

None

Note: There is no test for positive definiteness in these subroutines.

Input-Argument Errors

1. $n < 0$
2. *iopt* ≠ 0 or 1

Examples

Example

This example shows a factorization of the tridiagonal matrix ***A***, of order 4:

```
                        ┌                    ┐
                        | 1.0  1.0  0.0  0.0 |
                        | 1.0  2.0  1.0  0.0 |
                        | 0.0  1.0  3.0  1.0 |
                        | 0.0  0.0  1.0  1.0 |
                                             ┘
```

Call Statement and Input:

```
           N   C   D  IOPT
           |   |   |   |
CALL DPTF( 4 , C , D , 0  )

C        =  ( . , 1.0, 1.0, 1.0)
D        =  (1.0, 2.0, 3.0, 1.0)
```

Output:

```
C        =  ( . , -1.0, -1.0, -1.0)
D        =  (-1.0, -1.0, -1.0, -1.0)
```

Note

A "." means you do not have to store a value in that position in the array. However, these storage positions are required and may be overwritten during the computation.

SPTS and DPTS (Positive Definite Symmetric Tridiagonal Matrix Solve)

Purpose

These subroutines solve a positive definite symmetric tridiagonal system of equations using the factorization of matrix ***A***, stored in symmetric-tridiagonal storage mode, produced by SPTF and DPTF, respectively.

Table 191: Data Types.

Data Types	
c, ***d***, ***b***, ***x***	**Subroutine**
Short-precision real	SPTS
Long-precision real	DPTS

Note: The input to these solve subroutines must be the output from the factorization subroutines SPTF and DPTF, respectively.

Syntax

Language	Syntax
Fortran	CALL SPTS | DPTS (*n*, *c*, *d*, *bx*)
C and C++	spts | dpts (*n*, *c*, *d*, *bx*);

On Entry

n

is the order *n* of tridiagonal matrix ***A***. Specified as: an integer; $n \geq 0$.

c

is the vector ***c***, containing part of the factorization of matrix ***A*** from SPTF or DPTF, respectively, in an array, referred to as C. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 191 on page 718.

d

is the vector ***d***, containing part of the factorization of matrix ***A*** from SPTF or DPTF, respectively, in an array, referred to as D. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 191 on page 718.

bx

is the vector ***b***, containing the right-hand side of the system in the first *n* positions in an array, referred to as BX. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 191 on page 718.

On Return

bx

is the solution vector ***x*** of length *n*, containing the solution of the tridiagonal system in the first *n* positions in an array, referred to as BX. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 191 on page 718.

Notes

For a description of how tridiagonal matrices are stored, see “Positive Definite or Negative Definite Symmetric Matrix” on page 84.

Function

The solution of positive definite symmetric tridiagonal system ***Ax*** = ***b*** is computed using the factorization produced by SPTF or DPTF, respectively. The factorization is based on Gaussian elimination. See reference [“93” on page 1258].

Error conditions

Computational Errors

None

Input-Argument Errors

n < 0

Examples

Example

This example finds the solution of tridiagonal system ***Ax*** = ***b***, where matrix ***A*** is the same matrix factored in Example for SPTF and DPTF. ***b*** is:

```
                   (2.0, 4.0, 5.0, 2.0)
```

and ***x*** is:

```
                   (1.0, 1.0, 1.0, 1.0)
```

Call Statement and Input:

```
            N   C   D   BX
            |   |   |   |
CALL DPTS( 4 , C , D , BX )

C        =  ( . , -1.0, -1.0, -1.0)
D        =  (-1.0, -1.0, -1.0, -1.0)
BX       =  (2.0, 4.0, 5.0, 2.0)
```

Output:

```
BX       =  (1.0, 1.0, 1.0, 1.0)
```

Sparse Linear Algebraic Equation Subroutines

This contains the sparse linear algebraic equation subroutine descriptions.

DGSF (General Sparse Matrix Factorization Using Storage by Indices, Rows, or Columns)

Purpose

This subroutine factors sparse matrix ***A*** by Gaussian elimination, using a modified Markowitz count with threshold pivoting. The sparse matrix can be stored by indices, rows, or columns. To solve the system of equations, follow the call to this subroutine with a call to DGSS.

Syntax

Language	Syntax
Fortran	CALL DGSF (*iopt*, *n*, *nz*, *a*, *ia*, *ja*, *lna*, *iparm*, *rparm*, *oparm*, *aux*, *naux*)
C and C++	dgsf (*iopt*, *n*, *nz*, *a*, *ia*, *ja*, *lna*, *iparm*, *rparm*, *oparm*, *aux*, *naux*);

On Entry

iopt

indicates the storage technique used for sparse matrix ***A***, where:

If *iopt* = 0, it is stored by indices.

If *iopt* = 1, it is stored by rows.

If *iopt* = 2, it is stored by columns.

Specified as: an integer; *iopt* = 0, 1, or 2.

n

is the order *n* of sparse matrix ***A***. Specified as: an integer; *n* ≥ 0.

nz

is the number of elements in sparse matrix ***A***, stored in an array, referred to as A.

Specified as: an integer; *nz* > 0.

a

is the sparse matrix ***A***, to be factored, stored in an array, referred to as A.

Specified as: an array of length *lna*, containing long-precision real numbers.

ia

is the array, referred to as IA, where:

If *iopt* = 0, it contains the row numbers that correspond to the elements in array A.

If *iopt* = 1, it contains the row pointers.

If *iopt* = 2, it contains the row numbers that correspond to the elements in array A.

Specified as: an array of length *lna*, containing integers; IA(*i*) ≥ 1. See "Sparse Matrix" on page 110 for more information on storage techniques.

ja

is the array, referred to as JA, where:

If *iopt* = 0, it contains the column numbers that correspond to the elements in array A.

If *iopt* = 1, it contains the column numbers that correspond to the elements in array A.

If *iopt* = 2, it contains the column pointers.

Specified as: an array of length *lna*, containing integers; JA(*i*) ≥ 1. See "Sparse Matrix" on page 110 for more information on storage techniques.

lna

is the length of the arrays specified for *a*, *ia*, and *ja*.

Specified as: an integer; *lna* > 2*nz*. If you do not specify a sufficient amount, it results in an error. See "Error conditions" on page 723.

The size of *lna* depends on the structure of the input matrix. The requirement that *lna* > 2*nz* does not guarantee a successful run of the program. If the input matrix is expected to have many fill-ins, *lna* should be set larger. Larger *lna* may result in a performance improvement.

For details on how *lna* relates to storage compressions, see "Performance and Accuracy Considerations" on page 477.

iparm
is an array of parameters, `IPARM`(*i*), where:

- `IPARM(1)` determines whether the default values for *iparm* and *rparm* are used by this subroutine.

  If `IPARM(1)` = 0, the following default values are used:

  `IPARM(2)` = 10
  `IPARM(3)` = 1
  `IPARM(4)` = 0
  `RPARM(1)` = 10^{-12}
  `RPARM(2)` = 0.1

  If `IPARM(1)` = 1, the default values are not used.
- `IPARM(2)` determines the number of minimal Markowitz counts that are examined to determine a pivot. (See reference ["123" on page 1259].)
- `IPARM(3)` has the following meaning, where:

  If `IPARM(3)` = 0, this subroutine checks the values in arrays `IA` and `JA`.

  If `IPARM(3)` = 1, this subroutine assumes that the input values are correct in arrays `IA` and `JA`.
- `IPARM(4)` has the following meaning, where:

  If `IPARM(4)` = 0, this computation is not performed.

  If `IPARM(4)` = 1, this subroutine computes:

  The absolute value of the smallest pivot element
  The absolute value of the largest element in ***U***.

  These values are stored in `OPARM(2)` and `OPARM(3)`, respectively.
- `IPARM(5)` is reserved.

Specified as: an array of (at least) length 5, containing integers, where the *iparm* values must be:

`IPARM(1)` = 0 or 1
`IPARM(2)` ≥ 1
`IPARM(3)` = 0 or 1
`IPARM(4)` = 0 or 1

rparm
is an array of parameters, `RPARM`(*i*), where:

- `RPARM(1)` contains the lower bound of the absolute value of all elements in the matrix. If a pivot element is less than this number, the matrix is reported as singular. Any computed element whose absolute value is less than this number is set to 0.
- `RPARM(2)` is the threshold pivot tolerance used to control the choice of pivots.
- `RPARM(3)` is reserved.
- `RPARM(4)` is reserved.
- `RPARM(5)` is reserved.

Specified as: a one-dimensional array of (at least) length 5, containing long-precision real numbers, where the *rparm* values must be:

`RPARM(1)` ≥ 0.0
0.0 ≤ `RPARM(2)` ≤ 1.0

For additional information about *rparm*, see "Performance and Accuracy Considerations" on page 477.

oparm
See On Return.

aux
is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing long-precision real numbers.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: an integer.

For 32-bit integer arguments
naux ≥ 10*n*+100.

For 64-bit integer arguments
naux ≥ 18*n*+100.

On Return

a
is the transformed array, referred to as A, containing the factored matrix ***A***, required as input to DGSS. Returned as: a one-dimensional array of length *lna*, containing long-precision real numbers.

ia
is the transformed array, referred to as `IA`, required as input to DGSS. Returned as: a one-dimensional array of length *lna*, containing integers.

ja
is the transformed array, referred to as `JA`, required as input to DGSS. Returned as: a one-dimensional array of length *lna*, containing integers.

oparm
is an array of parameters, `OPARM`(*i*), where:

- `OPARM(1)` is the amount of fill-ins for the sparse processing portion of the algorithm.
- `OPARM(2)` contains the absolute value of the smallest pivot element of the matrix. This value is computed and set only if `IPARM(4)` = 1.
- `OPARM(3)` contains the absolute value of the largest element encountered in ***U*** after the factorization. This value is computed and set only if `IPARM(4)` = 1.
- `OPARM(4)` is reserved.
- `OPARM(5)` is reserved.

Returned as: a one-dimensional array of length 5, containing long-precision real numbers.

aux
is the storage work area used by this subroutine. It contains the information required as input for DGSS.

Specified as: an area of storage, containing long-precision real numbers.

Notes

1. For a description of the three storage techniques used by this subroutine for sparse matrices, see "Sparse Matrix" on page 110.
2. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

The matrix ***A*** is factored by Gaussian elimination, using a modified Markowitz count with threshold pivoting to compute the sparse LU factorization of ***A***:

LU = ***PAQ***

where:

A is a general sparse matrix of order *n*, stored by indices, columns, or rows in arrays `A`, `IA`, and `JA`.

L is a unit lower triangular matrix.

U is an upper triangular matrix.

P is a permutation matrix.

Q is a permutation matrix.

To solve the system of equations, follow the call to this subroutine with a call to DGSS. If *n* is 0, no computation is performed. See references [“17” on page 1254], [“58” on page 1256], and [“114” on page 1259].

Error conditions

Computational Errors

1. If this subroutine has to perform storage compressions, an attention message is issued. When this occurs, the performance of this subroutine is affected. The performance can be improved by increasing the value specified for *lna*.
2. The following errors with their corresponding return codes can occur in this subroutine. Where a value of *i* is indicated, it can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for that particular error code in the ESSL error option table; otherwise, the default value causes your program to terminate when the error occurs. For details, see “What Can You Do about ESSL Computational Errors?” on page 65.
   - For error 2117, return code 2 indicates that the pivot element in a column, *i*, is smaller than the value specified in `RPARM(1)`.
   - For error 2118, return code 3 indicates that pivot element in a row, *i*, is smaller than the value specified in `RPARM(1)`.
   - For error 2120, return code 4 indicates that a row, *i*, is found empty on factorization. The matrix is singular.
   - For error 2121, return code 5 indicates that a column is found empty on factorization. The matrix is singular.
   - For error 2119, return code 6 indicates that the storage space indicated by *lna* is insufficient.
   - For error 2122, return code 7 indicates that no pivot element was found in the active submatrix.

Input-Argument Errors

1. $iopt \neq 0, 1,$ or 2
2. $n < 0$
3. $nz \leq 0$
4. $lna \leq 2nz$
5. `IPARM(1)` ≠ 0 or 1
6. `IPARM(2)` ≤ 0

7. IPARM(3) ≠ 0 or 1
8. IPARM(4) ≠ 0 or 1
9. RPARM(1) < 0.0
10. RPARM(2) < 0.0 or RPARM(2) > 1.0
11. *iopt* = 1 and *ia*(*i*) ≥ *ia* (*i*+1), *i* = 1, *n*
12. *iopt* = 2 and *ja*(*i*) ≥ *ja*(*i*+1), *i* = 1, *n*
13. *iopt* = 0 or 1 and *ja*(*i*) < 1 or *ja*(*i*) > *n*, *i* = 1, *nz*
14. *iopt* = 0 or 1 and *ia*(*i*) < 1 or *ia*(*i*) > *n*, *i* = 1, *nz*
15. There are duplicate indices in a row or column of the input matrix.
16. The matrix is singular if a row or column of the input matrix is empty.
17. *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example

This example factors 5 by 5 sparse matrix ***A***, which is stored by indices in arrays A, IA, and JA. The three storage techniques are shown in this example, and the output is the same regardless of the storage technique used. The matrix is factored using Gaussian elimination with threshold pivoting. Matrix ***A*** is:

```
                        | 2.0  0.0  4.0  0.0  0.0 |
                        | 1.0  1.0  0.0  0.0  3.0 |
                        | 0.0  0.0  3.0  4.0  0.0 |
                        | 2.0  2.0  0.0  1.0  5.0 |
                        | 0.0  0.0  1.0  1.0  0.0 |
```

Note: In this example, only nonzero elements are used as input to the matrix.

Call Statement and Input (Storage-By-Indices):

```
          IOPT  N  NZ  A  IA  JA  LNA  IPARM  RPARM  OPARM  AUX  NAUX
           |    |   |  |   |   |   |     |      |      |     |    |
CALL DGSF( 0  , 5, 13, A, IA, JA, 27 , IPARM, RPARM, OPARM, AUX, 150 )
```

```
A        =  (2.0, 1.0, 1.0, 3.0, 4.0, 1.0, 5.0, 2.0, 2.0, 1.0, 1.0,
             4.0, 3.0, . , . , . , . , . , . , . , . , . , . , . , . ,
            . , . )
IA       =  (1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5, 1, 2, . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . , . , . , . , . )
JA       =  (1, 1, 2, 3, 4, 4, 5, 1, 2, 3, 4, 3, 5, . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . , . , . , . , . )
IPARM    =  (1, 3, 1, 1)
RPARM    =  (1.D-12, 0.1D0)
```

Call Statement and Input (Storage-By-Rows):

```
          IOPT  N  NZ  A  IA  JA  LNA  IPARM  RPARM  OPARM  AUX  NAUX
           |    |   |  |   |   |   |     |      |      |     |    |
CALL DGSF( 1  , 5, 13, A, IA, JA, 27 , IPARM, RPARM, OPARM, AUX, 150 )
```

```
A        =  (2.0, 4.0, 1.0, 1.0, 3.0, 3.0, 4.0, 2.0, 2.0, 1.0, 5.0,
             1.0, 1.0, . , . , . , . , . , . , . , . , . , . , . , . ,
            . , . )
IA       =  (1, 3, 6, 8, 12, 14, . , . , . , . , . , . , . , . , . , . ,
             . , . , . , . , . , . )
JA       =  (1, 3, 1, 2, 5, 3, 4, 1, 2, 4, 5, 3, 4, . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . , . , . , . , . )
IPARM    =  (1, 3, 1, 1)
RPARM    =  (1.D-12, 0.1D0)
```

Call Statement and Input (Storage-By-Columns):

```
          IOPT  N  NZ  A  IA  JA  LNA  IPARM  RPARM  OPARM  AUX  NAUX
           |    |   |  |   |   |   |     |      |      |     |    |
CALL DGSF( 2  , 5, 13, A, IA, JA, 27 , IPARM, RPARM, OPARM, AUX, 150 )
```

```
A        =  (2.0, 1.0, 2.0, 1.0, 2.0, 4.0, 3.0, 1.0, 4.0, 1.0, 1.0,
             3.0, 5.0, . , . , . , . , . , . , . , . , . , . , . , . ,
             . , . )
IA       =  (1, 2, 4, 2, 4, 1, 3, 5, 3, 4, 5, 2, 4, . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . , . )
JA       =  (1, 4, 6, 9, 12, 14, . , . , . , . , . , . , . , . , . , . ,
             . , . , . , . , . , . )
IPARM    =  (1, 3, 0, 1)
RPARM    =  (1.D-12, 0.1D0)
```

Output:

```
A        =  (0.5, . , 0.3, 1.0, . , 1.0, . , 3.0, . , . , . , 1.0,
             1.0, . , . , . , . , . , . , . , -1.7, -0.5, -1.0, -1.0,
             4.0, -3.0, -4.0)
IA       =  (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, . , . , . , . , . ,
             . , . , . , 2, 1, 1, 3, 3, 5, 5)
JA       =  (1, 0, 5, 2, 0, 4, 0, 2, 0, 0, 0, 3, 4, . , . , . , . , . ,
             . , . , . , 4, 2, 4, 4, 1, 3, 1)
OPARM    =  (1.000000, 0.333333, 3.000000)
```

Note: On input, a "." means that you do not have to store a value in that position in the array. However, the storage position is required and may be overwritten during the computation. On output, a "." means that the value in that position in the array is not significant.

DGSS (General Sparse Matrix or Its Transpose Solve Using Storage by Indices, Rows, or Columns)

Purpose

This subroutine solves either of the following systems:

$\boldsymbol{Ax} = \boldsymbol{b}$
$\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$

where $\boldsymbol{A}$ is a sparse matrix, $\boldsymbol{A}^T$ is the transpose of sparse matrix $\boldsymbol{A}$, and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors. DGSS uses the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to DGSF.

Note: The input to this solve subroutine must be the output from the factorization subroutine, DGSF.

Syntax

Language	Syntax
Fortran	CALL DGSS (*jopt*, *n*, *a*, *ia*, *ja*, *lna*, *bx*, *aux*, *naux*)
C and C++	dgss (*jopt*, *n*, *a*, *ia*, *ja*, *lna*, *bx*, *aux*, *naux*);

On Entry

jopt

indicates the type of computation to be performed, where:

If *jopt* = 0, $\boldsymbol{Ax} = \boldsymbol{b}$ is solved, where the right-hand side is not sparse.

If *jopt* = 1, $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$ is solved, where the right-hand side is not sparse.

If *jopt* = 10, $\boldsymbol{Ax} = \boldsymbol{b}$ is solved, where the right-hand side is sparse.

If *jopt* = 11, $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$ is solved, where the right-hand side is sparse.

Specified as: an integer; *jopt* = 0, 1, 10, or 11.

n
is the order *n* of sparse matrix ***A***. Specified as: an integer; $n \geq 0$.

a
is the factorization of sparse matrix ***A***, stored in array A, produced by a preceding call to DGSF.

Specified as: an array of length *lna*, containing long-precision real numbers.

ia
is the array, referred to as IA, produced by a preceding call to DGSF.

Specified as: an array of length *lna*, containing integers.

ja
is the array, referred to as JA, produced by a preceding call to DGSF.

Specified as: an array of length *lna*, containing integers.

lna
is the length of the arrays A, IA, and JA. In DGSS, *lna* must be identical to the value specified in DGSF; otherwise, results are unpredictable.

Specified as: an integer; $lna > 0$.

bx
is the vector ***b*** of length *n*, containing the right-hand side of the system.

Specified as: a one-dimensional array of (at least) length *n*, containing long-precision real numbers.

aux
is the storage work area passed to this subroutine by a preceding call to DGSF. Its size is specified by *naux*.

Specified as: an area of storage, containing long-precision real numbers.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: an integer.

For 32-bit integer arguments
$naux \geq 10n+100$.

For 64-bit integer arguments
$naux \geq 18n+100$.

On Return

ia
is the transformed array, referred to as IA, which can be used as input in subsequent calls to this subroutine. This may result in a performance increase.

Specified as: an array of length *lna*, containing integers.

bx
is the solution vector ***x*** of length *n*, containing the results of the computation.

Specified as: a one-dimensional array, containing long-precision real numbers.

Notes

1. The input arguments *n*, *lna*, and *naux*, must be the same as those specified for DGSF. Whereas, the input arguments *a, ia, ja*, and *aux* must be those produced on output by DGSF. Otherwise, results are unpredictable.
2. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

The system ***Ax*** = ***b*** is solved for ***x***, where ***A*** is a sparse matrix and ***x*** and ***b*** are vectors. Depending on the value specified for the *jopt* argument, DGSS can also solve the system $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{A}^T$ is the transpose of sparse matrix ***A***.

If the value specified for the *jopt* argument is 0 or 10, the following equation is solved:

Ax = ***b***

If the value specified for the *jopt* argument is 1 or 11, the following equation is solved:

$\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$

DGSS uses the results of the factorization of matrix ***A***, produced by a preceding call to DGSF. The transformed matrix ***A*** consists of the upper triangular matrix ***U*** and the lower triangular matrix ***L***.

See references [“17” on page 1254], [“58” on page 1256], and [“114” on page 1259].

Error conditions

Computational Errors
None

Input-Argument Errors

1. *jopt* ≠ 0, 1, 10, or 11
2. *n* < 0
3. *lna* ≤ 0
4. *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows how to solve the system ***Ax*** = ***b***, where matrix ***A*** is a 5 by 5 sparse matrix. The right-hand side is not sparse.

Note: The input for this subroutine is the same as the output from DGSF, except for BX.

Matrix ***A*** is:

```
                 | 2.0  0.0  4.0  0.0  0.0 |
                 | 1.0  1.0  0.0  0.0  3.0 |
                 | 0.0  0.0  3.0  4.0  0.0 |
                 | 2.0  2.0  0.0  1.0  5.0 |
                 | 0.0  0.0  1.0  1.0  0.0 |
```

Call Statement and Input:

```
          JOPT  N   A   IA   JA   LNA   BX   AUX   NAUX
           |    |   |    |    |    |     |    |     |
CALL DGSS( 0  , 5 , A , IA , JA , 27  , BX , AUX , 150 )
```

```
A        =  (0.5, . , 0.3, 1.0, . , 1.0, . , 3.0, . , . , . , 1.0,
             1.0, . , . , . , . , . , . , . , -1.7, -0.5, -1.0, -1.0,
             4.0, -3.0, -4.0)
IA       =  (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, . , . , . , . ,
             . , . , . , 2, 1, 1, 3, 3, 5, 5)
JA       =  (1, 0, 5, 2, 0, 4, 0, 2, 0, 0, 0, 3, 4, . , . , . , . ,
             . , . , . , 4, 2, 4, 4, 1, 3, 1)
BX       =  (1.0, 1.0, 1.0, 1.0, 1.0)
```

Output:

```
IA       =  (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, . , . , . , . ,
             . , . , . , 2, 1, 1, 3, 3, 5, 5)
BX       =  (-5.500000, 9.500000, 3.000000, -2.000000, -1.000000)
```

Note: On input, a "." means that you do not have to store a value in that position in the array. However, the storage position is required and may be overwritten during the computation. On output, a "." means that the value in that position in the array is not significant.

Example 2

This example shows how to solve the system $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$, using the same matrix $\boldsymbol{A}$ used in Example 1. The input is also the same as in Example 1, except for the *jopt* argument. The right-hand side is not sparse.

Call Statement and Input:

```
          JOPT  N   A   IA   JA   LNA   BX   AUX   NAUX
           |    |   |    |    |    |     |    |     |
CALL DGSS( 1  , 5 , A , IA , JA , 27  , BX , AUX , 150 )

BX       =  (1.0, 1.0, 1.0, 1.0, 1.0)
```

Output:

```
IA       =  (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, . , . , . , . ,
             . , . , . , 2, 1, 1, 3, 3, 5, 5)
BX       =  (0.000000, -3.000000, -2.000000, 2.000000, 7.000000)
```

Note: On input, a "." means that you do not have to store a value in that position in the array. However, the storage position is required and may be overwritten during the computation. On output, a "." means that the value in that position in the array is not significant.

Example 3

This example shows how to solve the system $\boldsymbol{Ax} = \boldsymbol{b}$, using the same matrix $\boldsymbol{A}$ as in Examples 1 and 2. The input is also the same as in Examples 1 and 2, except for the *jopt* and *bx* arguments. The right-hand side is sparse.

Call Statement and Input:

```
          JOPT  N   A   IA   JA   LNA   BX   AUX   NAUX
           |    |   |    |    |    |     |    |     |
CALL DGSS( 10 , 5 , A , IA , JA , 27  , BX , AUX , 150 )

BX       =  (0.0, 0.0, 0.0, 1.0, 0.0)
```

Output:

```
IA       =  (1, 4, 2, 5, 3, 5, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
             0, 2, 1, 1, 3, 3, 5, 5)
BX       =  (0.000000, 3.000000, 0.000000, 0.000000, -1.000000)
```

Note: On input, a "." means that you do not have to store a value in that position in the array. However, the storage position is required and may be overwritten during the computation. On output, a "." means that the value in that position in the array is not significant.

Example 4

This example shows how to solve the system $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$, using the same matrix $\boldsymbol{A}$ as in Examples 1, 2, and 3. The input is also the same as in Examples 1, 2, and 3, except for the *jopt* argument. The right-hand side is sparse.

Call Statement and Input:

```
          JOPT  N   A   IA   JA   LNA   BX   AUX   NAUX
           |    |   |    |    |    |     |    |     |
CALL DGSS( 11 , 5 , A , IA , JA , 27  , BX , AUX , 150 )
```

```
BX       =  (0.0, 0.0, 0.0, 1.0, 0.0)
```

Output:

```
IA       =  (1, 4, 2, 5, 3, 5, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
             0, 2, 1, 1, 3, 3, 5, 5)
BX       =   (0.000000, 0.000000, 1.000000, 0.000000, -3.000000 )
```

Note: On input, a "." means that you do not have to store a value in that position in the array. However, the storage position is required and may be overwritten during the computation. On output, a "." means that the value in that position in the array is not significant.

DGKFS (General Sparse Matrix or Its Transpose Factorization, Determinant, and Solve Using Skyline Storage Mode)

Purpose

This subroutine can perform either or both of the following functions for general sparse matrix ***A***, stored in skyline storage mode, and for vectors ***x*** and ***b***:

- Factor ***A*** and, optionally, compute the determinant of ***A***.
- Solve the system ***Ax*** = ***b*** or $\boldsymbol{A}^{T}\boldsymbol{x}$ = ***b*** using the results of the factorization of matrix ***A***, produced on this call or a preceding call to this subroutine.

You also have the choice of using profile-in or diagonal-out skyline storage mode for ***A*** on input or output.

Note: The input to the solve performed by this subroutine must be the output from the factorization performed by this subroutine.

Syntax

Language	Syntax
Fortran	CALL DGKFS (*n*, *au*, *nu*, *idu*, *al*, *nl*, *idl*, *iparm*, *rparm*, *aux*, *naux*, *bx*, *ldbx*, *mbx*)
C and C++	dgkfs (*n*, *au*, *nu*, *idu*, *al*, *nl*, *idl*, *iparm*, *rparm*, *aux*, *naux*, *bx*, *ldbx*, *mbx*);

On Entry

n

is the order of general sparse matrix ***A***. Specified as: an integer; *n* ≥ 0.

au

is the array, referred to as AU, containing one of three forms of the upper triangular part of general sparse matrix ***A***, depending on the type of computation performed, where:

- If you are doing a **factor and solve** or a **factor only**, and if `IPARM(3)` = 0, then AU contains the unfactored upper triangle of general sparse matrix ***A***.
- If you are doing a **factor only**, and if `IPARM(3)` > 0, then AU contains the partially factored upper triangle of general sparse matrix ***A***. The first `IPARM(3)` columns in the upper triangle of ***A*** are already factored. The remaining columns are factored in this computation.
- If you are doing a **solve only**, then AU contains the factored upper triangle of general sparse matrix ***A***, produced by a preceding call to this subroutine.

In each case:

If `IPARM(4)` = 0, diagonal-out skyline storage mode is used for ***A***.

If `IPARM(4)` = 1, profile-in skyline storage mode is used for ***A***.

Specified as: a one-dimensional array of (at least) length *nu*, containing long-precision real numbers.

nu

is the length of array AU.

Specified as: an integer; *nu* ≥ 0 and *nu* ≥ (IDU(*n*+1)-1).

idu
is the array, referred to as IDU, containing the relative positions of the diagonal elements of matrix ***A*** (in one of its three forms) in array AU.

Specified as: a one-dimensional array of (at least) length *n*+1, containing integers.

al
is the array, referred to as AL, containing one of three forms of the lower triangular part of general sparse matrix ***A***, depending on the type of computation performed, where:

- If you are doing a **factor and solve** or a **factor only**, and if IPARM(3) = 0, then AL contains the unfactored lower triangle of general sparse matrix ***A***.
- If you are doing a **factor only**, and if IPARM(3) > 0, then AL contains the partially factored lower triangle of general sparse matrix ***A***. The first IPARM(3) rows in the lower triangle of ***A*** are already factored. The remaining rows are factored in this computation.
- If you are doing a **solve only**, then AL contains the factored lower triangle of general sparse matrix ***A***, produced by a preceding call to this subroutine.

Note: In all these cases, entries in AL for diagonal elements of ***A*** are not assumed to have meaningful values.

In each case:

If IPARM(4) = 0, diagonal-out skyline storage mode is used for ***A***.

If IPARM(4) = 1, profile-in skyline storage mode is used for ***A***.

Specified as: a one-dimensional array of (at least) length *nl*, containing long-precision real numbers.

nl
is the length of array AL.

Specified as: an integer; *nl* ≥ 0 and *nl* ≥ (IDL(*n*+1)-1).

idl
is the array, referred to as IDL, containing the relative positions of the diagonal elements of matrix ***A*** (in one of its three forms) in array AL.

Specified as: a one-dimensional array of (at least) length *n*+1, containing integers.

iparm
is an array of parameters, IPARM(*i*), where:

- IPARM(1) indicates whether certain default values for *iparm* and *rparm* are used by this subroutine, where:

  If IPARM(1) = 0, the following default values are used. For restrictions, see "Notes" on page 735.

  IPARM(2) = 0
  IPARM(3) = 0
  IPARM(4) = 0
  IPARM(5) = 0
  IPARM(10) = 0
  IPARM(11) = -1
  IPARM(12) = -1
  IPARM(13) = -1
  IPARM(14) = -1
  IPARM(15) = 0
  RPARM(10) = 10^{-12}

  If IPARM(1) = 1, the default values are not used.

- IPARM(2) indicates the type of computation performed by this subroutine. The following table gives the IPARM(2) values for each variation:

Type of Computation	$Ax = b$	$Ax = b$ and Determinant(A)	$A^Tx = b$	$A^Tx = b$ and Determinant(A)
Factor and Solve	0	10	100	110
Factor Only	1	11	N/A	N/A
Solve Only	2	N/A	102	N/A

- `IPARM(3)` indicates whether a full or partial factorization is performed on matrix ***A***, where:

  If `IPARM(3)` = 0, and:

  If you are doing a **factor and solve** or a **factor only**, then a full factorization is performed for matrix ***A*** on rows and columns 1 through *n*.

  If you are doing a **solve only**, this argument has no effect on the computation, but must be set to 0.

  If `IPARM(3)` > 0, and you are doing a **factor only**, then a partial factorization is performed on matrix ***A***. Rows 1 through `IPARM(3)` of columns 1 through `IPARM(3)` in matrix ***A*** must be in factored form from a preceding call to this subroutine. The factorization is performed on rows `IPARM(3)+1` through *n* and columns `IPARM(3)+1` through *n*. For an illustration, see "Notes" on page 735.
- `IPARM(4)` indicates the input storage mode used for matrix ***A***. This determines the arrangement of data in arrays `AU`, `IDU`, `AL`, and `IDL` on input, where:

  If `IPARM(4)` = 0, diagonal-out skyline storage mode is used.

  If `IPARM(4)` = 1, profile-in skyline storage mode is used.
- `IPARM(5)` indicates the output storage mode used for matrix ***A***. This determines the arrangement of data in arrays `AU`, `IDU`, `AL`, and `IDL` on output, where:

  If `IPARM(5)` = 0, diagonal-out skyline storage mode is used.

  If `IPARM(5)` = 1, profile-in skyline storage mode is used.
- `IPARM(6)` through `IPARM(9)` are reserved.
- `IPARM(10)` has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, then `IPARM(10)` indicates whether certain default values for *iparm* and *rparm* are used by this subroutine, where:

  If `IPARM(10)` = 0, the following default values are used. For restrictions, see "Notes" on page 735.

  `IPARM(11)` = -1
  `IPARM(12)` = -1
  `IPARM(13)` = -1
  `IPARM(14)` = -1
  `IPARM(15)` = 0
  `RPARM(10)` = 10^{-12}

  If `IPARM(10)` = 1, the default values are not used.

  If you are doing a **solve only**, this argument is not used.
- `IPARM(11)` through `IPARM(15)` have the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, then `IPARM(11)` through `IPARM(15)` control the type of processing to apply to pivot elements occurring in regions 1 through 5, respectively. The pivot elements are u_{kk} for $k = 1, n$ when doing a full factorization, and they are k = `IPARM(3)+1`, n when doing a partial factorization. The region in which a pivot element falls depends on the sign and magnitude of the pivot element. The regions are determined by `RPARM(10)`. For a description of the regions and associated pivot values, see "Notes" on page 735. For each region i for $i = 1,5$, where the pivot occurs in region i, the processing applied to the pivot element is determined by `IPARM(10+`*i*`)`, where:

If IPARM(10+*i*) = -1, the pivot element is trapped and computational error 2126 is generated. See "Error conditions" on page 737.

If IPARM(10+*i*) = 0, for *i* = 1, 2, 4, and 5, processing continues normally.

Note: A value of 0 is not permitted for region 3, because if processing continues, a divide-by-zero exception occurs.

If IPARM(10+*i*) = 1, the pivot element is replaced with the value in RPARM(10+*i*), and processing continues normally.

If you are doing a **solve only**, these arguments are not used.

- IPARM(16) through IPARM(25), see On Return.

Specified as: a one-dimensional array of (at least) length 25, containing integers, where:

IPARM(1) = 0 or 1
IPARM(2) = 0, 1, 2, 10, 11, 100, 102, or 110
If IPARM(2) = 0, 2, 10, 100, 102, or 110, then IPARM(3) = 0
If IPARM(2) = 1 or 11, then 0 ≤ IPARM(3) ≤ *n*
IPARM(4), IPARM(5) = 0 or 1
If IPARM(2) = 0, 1, 10, 11, 100, or 110, then:
IPARM(10) = 0 or 1
IPARM(11), IPARM(12) = -1, 0, or 1
IPARM(13) = -1 or 1
IPARM(14), IPARM(15) = -1, 0, or 1

rparm
is an array of parameters, RPARM(*i*), where:

- RPARM(1) through RPARM(9) are reserved.
- RPARM(10) has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, RPARM(10) is the tolerance value for small pivots. This sets the bounds for the pivot regions, where pivots are processed according to the options you specify for the five regions in IPARM(11) through IPARM(15), respectively. The suggested value is 10^{-15} ≤ IPARM(10) ≤ 1.

  If you are doing a **solve only**, this argument is not used.
- RPARM(11) through RPARM(15) have the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, RPARM(11) through RPARM(15) are the fix-up values to use for the pivots in regions 1 through 5, respectively. For each RPARM(10+*i*) for *i* = 1,5, where the pivot occurs in region *i*:

  If IPARM(10+*i*) = 1, the pivot is replaced with RPARM(10+*i*), where |RPARM(10+*i*)| should be a sufficiently large nonzero value to avoid overflow when calculating the reciprocal of the pivot. The suggested value is 10^{-15} ≤ |RPARM(10+*i*)| ≤ 1.

  If IPARM(10+*i*) ≠ 1, RPARM(10+*i*) is not used.

  If you are doing a **solve only**, these arguments are not used.
- RPARM(16) through RPARM(25), see On Return.

Specified as: a one-dimensional array of (at least) length 25, containing long-precision real numbers, where if IPARM(2) = 0, 1, 10, 11, 100, or 110, then:

RPARM(10) ≥ 0.0
RPARM(11) through RPARM(15) ≠ 0.0

aux
has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing long-precision real numbers.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, DGKFS dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise,

If you are doing a **factor only**:

For 32-bit integer arguments
Use $naux \geq 5n$.

For 64-bit integer arguments
Use $naux \geq 7n$.

If you are doing a **factor and solve** or a **solve only**:

For 32-bit integer arguments
Use $naux \geq 5n + 4mbx$.

For 64-bit integer arguments
Use $naux \geq 7n + 4mbx$.

bx

has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *bx* is the array, containing the *mbx* right-hand side vectors $\boldsymbol{b}$ of the system $\boldsymbol{Ax} = \boldsymbol{b}$ or $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$. Each vector $\boldsymbol{b}$ is length *n* and is stored in the corresponding column of the array.

If you are doing a **factor only**, this argument is not used in the computation.

Specified as: an *ldbx* by (at least) *mbx* array, containing long-precision real numbers.

ldbx

has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *ldbx* is the leading dimension of the array specified for *bx*.

If you are doing a **factor only**, this argument is not used in the computation.

Specified as: an integer; $ldbx \geq n$ and:

If $mbx \neq 0$, then $ldbx > 0$.

If $mbx = 0$, then $ldbx \geq 0$.

mbx

has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *mbx* is the number of right-hand side vectors, $\boldsymbol{b}$, in the array specified for *bx*.

If you are doing a **factor only**, this argument is not used in the computation.

Specified as: an integer; $mbx \geq 0$.

On Return

au

is the array, referred to as AU, containing the upper triangular part of the $\boldsymbol{LU}$ factored form of general sparse matrix $\boldsymbol{A}$, where:

If `IPARM(5)` = 0, diagonal-out skyline storage mode is used for $\boldsymbol{A}$.

If `IPARM(5)` = 1, profile-in skyline storage mode is used for ***A***.

(If *mbx* = 0 and you are doing a solve only, then *au* is unchanged on output.) Returned as: a one-dimensional array of (at least) length *nu*, containing long-precision real numbers.

idu
is the array, referred to as `IDU`, containing the relative positions of the diagonal elements of the factored output matrix ***A*** in array AU. (If *mbx* = 0 and you are doing a solve only, then *idu* is unchanged on output.) Returned as: a one-dimensional array of (at least) length *n*+1, containing integers.

al
is the array, referred to as `AL`, containing the lower triangular part of the ***LU*** factored form of general sparse matrix ***A***, where:

If `IPARM(5)` = 0, diagonal-out skyline storage mode is used for ***A***.

If `IPARM(5)` = 1, profile-in skyline storage mode is used for ***A***.

Note: You should assume that entries in AL for diagonal elements of ***A*** do not have meaningful values.

(If *mbx* = 0 and you are doing a solve only, then *al* is unchanged on output.) Returned as: a one-dimensional array of (at least) length *nl*, containing long-precision real numbers.

idl
is the array, referred to as `IDL`, containing the relative positions of the diagonal elements of the factored output matrix ***A*** in array AL. (If *mbx* = 0 and you are doing a solve only, then *idl* is unchanged on output.) Returned as: a one-dimensional array of (at least) length *n*+1, containing integers.

iparm
is an array of parameters, `IPARM`(*i*), where:

- `IPARM(1)` through `IPARM(15)` are unchanged.
- `IPARM(16)` has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, and:

  If `IPARM(16)` = -1, your factorization did not complete successfully, resulting in computational error 2126.

  If `IPARM(16)` > 0, it is the row number *k*, in which the maximum absolute value of the ratio a_{kk}/u_{kk} occurred, where:

  If `IPARM(3)` = 0, *k* can be any of the rows, 1 through *n*, in the full factorization.

  If `IPARM(3)` > 0, *k* can be any of the rows, `IPARM(3)`+1 through *n*, in the partial factorization.

  If you are doing a **solve only**, this argument is not used in the computation and is unchanged.
- `IPARM(17)` through `IPARM(20)` are reserved.
- `IPARM(21)` through `IPARM(25)` have the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, `IPARM(21)` through `IPARM(25)` have the following meanings for each region *i* for *i* = 1,5, respectively:

  If `IPARM`(20+*i*) = -1, your factorization did not complete successfully, resulting in computational error 2126.

  If `IPARM`(20+*i*) ≥ 0, it is the number of pivots in region *i* for the columns that were factored in matrix ***A***, where:

  If `IPARM(3)` = 0, columns 1 through *n* were factored in the full factorization.

  If `IPARM(3)` > 0, columns `IPARM(3)`+1 through *n* were factored in the partial factorization.

  If you are doing a **solve only**, these arguments are not used in the computation and are unchanged.

Returned as: a one-dimensional array of (at least) length 25, containing integers.

rparm

is an array of parameters, `RPARM`(*i*), where:

- `RPARM(1)` through `RPARM(15)` are unchanged.
- `RPARM(16)` has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, and:

  If `RPARM(16)` = 0.0, your factorization did not complete successfully, resulting in computational error 2126.

  If |`RPARM(16)`| > 0.0, it is the ratio for row *k*, a_{kk}/u_{kk}, having the maximum absolute value. Row *k* is indicated in `IPARM(16)`, and:

  If `IPARM(3)` = 0, the ratio corresponds to one of the rows, 1 through *n*, in the full factorization.

  If `IPARM(3)` > 0, the ratio corresponds to one of the rows, `IPARM(3)`+1 through *n*, in the partial factorization.

  If you are doing a **solve only**, this argument is not used in the computation and is unchanged.
- `RPARM(17)` and `RPARM(18)` have the following meaning, where:

  If you are **computing the determinant** of matrix ***A***, then `RPARM(17)` is the mantissa, *detbas*, and `RPARM(18)` is the power of 10, *detpwr*, used to express the value of the determinant: $detbas(10^{detpwr})$, where $1 \le detbas < 10$. Also:

  If `IPARM(3)` = 0, the determinant is computed for columns 1 through *n* in the full factorization.

  If `IPARM(3)` > 0, the determinant is computed for columns `IPARM(3)`+1 through *n* in the partial factorization.

  If you are **not computing the determinant** of matrix ***A***, these arguments are not used in the computation and are unchanged.
- `RPARM(19)` through `RPARM(25)` are reserved.

Returned as: a one-dimensional array of (at least) length 25, containing long-precision real numbers.

bx

has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *bx* is the array, containing the *mbx* solution vectors ***x*** of the system $\boldsymbol{Ax} = \boldsymbol{b}$ or $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$. Each vector ***x*** is length *n* and is stored in the corresponding column of the array. (If *mbx* = 0, then *bx* is unchanged on output.)

If you are doing a **factor only**, this argument is not used in the computation and is unchanged.

Returned as: an *ldbx* by (at least) *mbx* array, containing long-precision real numbers.

Notes

1. If you set either `IPARM(1)` = 0 or `IPARM(10)` = 0, indicating you want to use the default values for `IPARM(11)` through `IPARM(15)` and `RPARM(10)`, then:
   - Matrix ***A*** must be positive definite.
   - No pivots are fixed, using `RPARM(11)` through `RPARM(15)` values.
   - No small pivots are tolerated; that is, the value should be |*pivot*| > `RPARM(10)`.
2. Many of the input and output parameters for *iparm* and *rparm* are defined for the five pivot regions handled by this subroutine. The limits of the regions are based on `RPARM(10)`, as shown in Figure 13 on page 736. The pivot values in each region are:

   Region 1: *pivot* < -`RPARM(10)`
   Region 2: -`RPARM(10)` ≤ *pivot* < 0
   Region 3: *pivot* = 0

Region 4: 0 < *pivot* ≤ `RPARM(10)`
Region 5: *pivot* > `RPARM(10)`

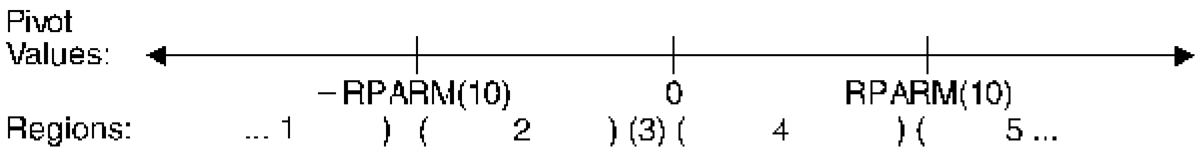


Figure 13: Five Pivot Regions

3. The `IPARM(4)` and `IPARM(5)` arguments allow you to specify the same or different skyline storage modes for your input and output arrays for matrix ***A***. This allows you to change storage modes as needed. However, if you are concerned with performance, you should use diagonal-out skyline storage mode for both input and output, if possible, because there is less overhead.

   For a description of how sparse matrices are stored in skyline storage mode, see "Profile-In Skyline Storage Mode" on page 119 and "Diagonal-Out Skyline Storage Mode" on page 117.

4. Following is an illustration of the portion of matrix ***A*** factored in the partial factorization when `IPARM(3)` > 0. In this case, the subroutine assumes that rows and columns 1 through `IPARM(3)` are already factored and that rows and columns `IPARM(3)`+1 through *n* are to be factored in this computation.

$$
\begin{array}{cc}
 & \left|\leftarrow \text{factored} \rightarrow\right|\leftarrow \text{to be factored} \rightarrow\Big| \\
\begin{array}{c} \overline{\uparrow} \\ \text{factored} \\ \downarrow \\ \uparrow \\ \text{to be} \\ \text{factored} \\ \underline{\downarrow} \end{array} &
\begin{bmatrix}
a_{11} & \cdots & a_{1j} & a_{1,j+1} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots & & & \vdots \\
a_{j1} & \cdots & a_{jj} & & & \vdots \\
a_{j+1,1} & & & & & \vdots \\
\vdots & & & & \ddots & \vdots \\
a_{n1} & \cdots & \cdot & \cdot & \cdots & a_{nn}
\end{bmatrix}
\end{array}
\qquad \text{where } j = \texttt{IPARM(3)}
$$

   You use the partial factorization function when, for design or storage reasons, you must factor the matrix ***A*** in stages. When doing a partial factorization, you must use the same skyline storage mode for all parts of the matrix as it is progressively factored.

5. Your various arrays must have no common elements; otherwise, results are unpredictable.
6. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

This subroutine can factor, compute the determinant of, and solve general sparse matrix ***A***, stored in skyline storage mode. For all computations, input matrix ***A*** can be stored in either diagonal-out or profile-in skyline storage mode. Output matrix ***A*** can also be stored in either of these modes and can be different from the mode used for input.

Matrix ***A*** is factored into the following form using specified pivot processing:

A = ***LU***

where:

U is an upper triangular matrix.
L is a lower triangular matrix.

The transformed matrix ***A***, factored into its ***LU*** form, is stored in packed format in arrays AU and AL. The inverse of the diagonal of matrix ***U*** is stored in the corresponding elements of array AU. The off-diagonal elements of the upper triangular matrix ***U*** are stored in the corresponding off-diagonal elements of array AU. The off-diagonal elements of the lower triangular matrix ***L*** are stored in the corresponding off-diagonal elements of array AL. (The diagonal elements stored in array AL do not have meaningful values.)

The partial factorization of matrix ***A***, which you can do when you specify the factor-only option, assumes that the first `IPARM(3)` rows and columns are already factored in the input matrix. It factors the remaining *n*-`IPARM(3)` rows and columns in matrix ***A***. (See "Notes" on page 735 for an illustration.) It updates only the elements in arrays AU and AL corresponding to the part of matrix ***A*** that is factored.

The determinant can be computed with any of the factorization computations. With a full factorization, you get the determinant for the whole matrix. With a partial factorization, you get the determinant for only that part of the matrix factored in this computation.

The system $\boldsymbol{Ax} = \boldsymbol{b}$ or $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$, having multiple right-hand sides, is solved for ***x***, using the transformed matrix ***A*** produced by this call or a subsequent call to this subroutine.

See references ["12" on page 1253], ["20" on page 1254], ["33" on page 1254], ["58" on page 1256], and ["85" on page 1257]. If *n* is 0, no computation is performed. If *mbx* is 0, no solve is performed.

Error conditions

Resource Errors

- Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.
- Unable to allocate internal work area.

Computational Errors

1. If a pivot occurs in region *i* for *i* = 1,5 and IPARM(10+*i*) = 1, the pivot value is replaced with RPARM(10+*i*), an attention message is issued, and processing continues.
2. Unacceptable pivot values occurred in the factorization of matrix ***A***.
   - One or more diagonal elements of ***U*** contains unacceptable pivots and no valid fixup is applicable. The row number *i* of the first unacceptable pivot element is identified in the computational error message.
   - The return code is set to 2.
   - *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2126 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

Input-Argument Errors

1. *n* < 0
2. *nu* < 0
3. `IDU`(*n*+1) > *nu*+1
4. IDU(*i*+1) ≤ IDU(*i*) for *i* = 1, *n*
5. IDU(*i*+1) > IDU(*i*)+*i* and `IPARM(4)` = 0 for *i* = 1, *n*
6. IDU(*i*) > IDU(*i*-1)+*i* and `IPARM(4)` = 1 for *i* = 2, *n*
7. *nl* < 0

8. `IDL`(*n*+1) > *nl*+1
9. `IDL`(*i*+1) ≤ `IDL`(*i*) for *i* = 1, *n*
10. `IDL`(*i*+1) > `IDL`(*i*)+*i* and `IPARM(4)` = 0 for *i* = 1, *n*
11. `IDL`(*i*) > `IDL`(*i*-1)+*i* and `IPARM(4)` = 1 for *i* = 2, *n*
12. `IPARM(1)` ≠ 0 or 1
13. `IPARM(2)` ≠ 0, 1, 2, 10, 11, 100, 102, or 110
14. `IPARM(3)` < 0
15. `IPARM(3)` > *n*
16. `IPARM(3)` > 0 and `IPARM(2)` ≠ 1 or 11
17. `IPARM(4)`, `IPARM(5)` ≠ 0 or 1
18. `IPARM(2)` = 0, 1, 10, 11, 100, or 110 and:

    `IPARM(10)` ≠ 0 or 1
    `IPARM(11)`, `IPARM(12)` ≠ -1, 0, or 1
    `IPARM(13)` ≠ -1 or 1
    `IPARM(14)`, `IPARM(15)` ≠ -1, 0, or 1
    `RPARM(10)` < 0.0
    `RPARM`(10+*i*) = 0.0 and `IPARM`(10+*i*) = 1 for *i* = 1,5
19. `IPARM(2)` = 0, 2, 10, 100, 102, or 110 and:

    ldbx ≤ 0 and *mbx* ≠ 0 and *n* ≠ 0
    ldbx < 0 and *mbx* = 0
    ldbx < *n* and *mbx* ≠ 0
    mbx < 0
20. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows how to factor a 9 by 9 general sparse matrix ***A*** and solve the system ***Ax*** = ***b*** with three right-hand sides. The default values are used for `IPARM` and `RPARM`. Input matrix ***A***, shown here, is stored in diagonal-out skyline storage mode. Matrix ***A*** is:

```
        | 2.0  2.0  2.0  0.0  0.0  0.0   0.0  0.0   0.0 |
        | 2.0  4.0  4.0  2.0  2.0  0.0   0.0  0.0   2.0 |
        | 2.0  4.0  6.0  4.0  4.0  0.0   2.0  0.0   4.0 |
        | 2.0  4.0  6.0  6.0  6.0  2.0   4.0  0.0   6.0 |
        | 0.0  0.0  0.0  2.0  4.0  4.0   4.0  2.0   4.0 |
        | 0.0  2.0  4.0  6.0  8.0  6.0   8.0  4.0  10.0 |
        | 0.0  0.0  0.0  2.0  4.0  6.0   8.0  6.0   8.0 |
        | 0.0  0.0  0.0  2.0  4.0  6.0   8.0  8.0  10.0 |
        | 2.0  4.0  6.0  6.0  8.0  6.0  10.0  8.0  16.0 |
```

Output matrix ***A***, shown here, is in ***LU*** factored form with U^{-1} on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix ***B*** is:

```
        | 0.5  2.0  2.0  0.0  0.0  0.0  0.0  0.0  0.0 |
        | 1.0  0.5  2.0  2.0  2.0  0.0  0.0  0.0  2.0 |
        | 1.0  1.0  0.5  2.0  2.0  0.0  2.0  0.0  2.0 |
        | 1.0  1.0  1.0  0.5  2.0  2.0  2.0  0.0  2.0 |
        | 0.0  0.0  0.0  1.0  0.5  2.0  2.0  2.0  2.0 |
        | 0.0  1.0  1.0  1.0  1.0  0.5  2.0  2.0  2.0 |
        | 0.0  0.0  0.0  1.0  1.0  1.0  0.5  2.0  2.0 |
        | 0.0  0.0  0.0  1.0  1.0  1.0  1.0  0.5  2.0 |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  0.5 |
```

Call Statement and Input:

```
             N   AU  NU  IDU  AL  NL  IDL  IPARM  RPARM  AUX  NAUX  BX  LDBX MBX
             |   |   |    |   |   |    |     |      |     |     |    |    |    |
CALL DGKFS( 9 , AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX,  57 , BX , 12 , 3 )
```

```
AU       =  (2.0, 4.0, 2.0, 6.0, 4.0, 2.0, 6.0, 4.0, 2.0, 4.0, 6.0,
             4.0, 2.0, 6.0, 4.0, 2.0, 8.0, 8.0, 4.0, 4.0, 2.0, 8.0,
             6.0, 4.0, 2.0, 16.0, 10.0, 8.0, 10.0, 4.0, 6.0, 4.0, 2.0)
IDU      =  (1, 2, 4, 7, 10, 14, 17, 22, 26, 34)
AL       =  (0.0, 0.0, 2.0, 0.0, 4.0, 2.0, 0.0, 6.0, 4.0, 2.0, 0.0,
             2.0, 0.0, 8.0, 6.0, 4.0, 2.0, 0.0, 6.0, 4.0, 2.0, 0.0,
             8.0, 6.0, 4.0, 2.0, 0.0, 8.0, 10.0, 6.0, 8.0, 6.0, 6.0,
             4.0, 2.0)
IDL      =  (1, 2, 4, 7, 11, 13, 18, 22, 27, 36)
IPARM    =  (0, . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . , . , . )
```

RPARM =(not relevant)

```
         |  6.00   12.00   18.00 |
         | 16.00   32.00   48.00 |
         | 26.00   52.00   78.00 |
         | 36.00   72.00  108.00 |
         | 20.00   40.00   60.00 |
BX   =   | 48.00   96.00  144.00 |
         | 34.00   68.00  102.00 |
         | 38.00   76.00  114.00 |
         | 66.00  132.00  198.00 |
         |   .       .       .   |
         |   .       .       .   |
         |   .       .       .   |
```

Output:

AU = (0.5, 0.5, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 0.5,
2.0,
2.0, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 2.0, 2.0, 0.5,
2.0, 2.0, 2.0, 0.5, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0)
IDU =(same as input)
AL = (0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0,
1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0,
1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0)
IDL =(same as input)
IPARM = (0, . , . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
. , 9, . , . , . , . , 0, 0, 0, 0, 9)
RPARM = (. , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
. , 8.0, . , . , . , . , . , . , . , . , .)

```
         | 1.00  2.00  3.00 |
         | 1.00  2.00  3.00 |
         | 1.00  2.00  3.00 |
         | 1.00  2.00  3.00 |
         | 1.00  2.00  3.00 |
BX   =   | 1.00  2.00  3.00 |
         | 1.00  2.00  3.00 |
         | 1.00  2.00  3.00 |
         | 1.00  2.00  3.00 |
         |  .     .     .   |
         |  .     .     .   |
         |  .     .     .   |
```

Example 2

This example shows how to factor the 9 by 9 general sparse matrix ***A*** from Example 1, solve the system $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$ with three right-hand sides, and compute the determinant of ***A***. The default values for pivot processing are used for IPARM. Input matrix ***A*** is stored in profile-in skyline storage mode. Output matrix ***A*** is in ***LU*** factored form with $\boldsymbol{U}^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. It is the same as output matrix ***A*** in Example 1.

Call Statement and Input:

```
             N  AU  NU  IDU  AL  NL  IDL  IPARM  RPARM  AUX  NAUX  BX  LDBX MBX
             |  |   |    |   |   |    |     |      |     |    |    |    |   |
CALL DGKFS( 9, AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX,  57 , BX , 12 , 3 )
```

```
AU       =  (2.0, 2.0, 4.0, 2.0, 4.0, 6.0, 2.0, 4.0, 6.0, 2.0, 4.0,
             6.0, 4.0, 2.0, 4.0, 6.0, 2.0, 4.0, 4.0, 8.0, 8.0, 2.0,
             4.0, 6.0, 8.0, 2.0, 4.0, 6.0, 4.0, 10.0, 8.0, 10.0, 16.0)
IDU      =  (1, 3, 6, 9, 13, 16, 21, 25, 33, 34)
AL       =  (0.0, 2.0, 0.0, 2.0, 4.0, 0.0, 2.0, 4.0, 6.0, 0.0, 2.0,
             0.0, 2.0, 4.0, 6.0, 8.0, 0.0, 2.0, 4.0, 6.0, 0.0, 2.0,
             4.0, 6.0, 8.0, 0.0, 2.0, 4.0, 6.0, 6.0, 8.0, 6.0, 10.0,
             8.0, 0.0)
IDL      =  (1, 3, 6, 10, 12, 17, 21, 26, 35, 36)
IPARM    =  (1, 110, 0, 1, 0, . , . , . , . , . , 0, . , . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . , . )
```

RPARM =(not relevant)

```
        | 10.00   20.00   30.00 |
        | 20.00   40.00   60.00 |
        | 28.00   56.00   84.00 |
        | 30.00   60.00   90.00 |
        | 40.00   80.00  120.00 |
BX   =  | 30.00   60.00   90.00 |
        | 44.00   88.00  132.00 |
        | 28.00   56.00   84.00 |
        | 60.00  120.00  180.00 |
        |   .       .       .   |
        |   .       .       .   |
        |   .       .       .   |
```

Output:

AU =(same as output AU in Example 1)
IDU =(same as output IDU in Example 1)
AL =(same as output AL in Example 1)
IDL =(same as output IDL in Example 1)
IPARM = (1, 110, 0, 1, 0, . , . , . , . , . , 0, . , . , . , . , . , . ,
9, . , . , . , . , . , 0, 0, 0, 0, 9)
RPARM = (. , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
. , 8.0, 5.12, 2.0, . , . , . , . , . , . , .)
BX =(same as output BX in Example 1)

Example 3

This example shows how to factor a 9 by 9 negative-definite general sparse matrix ***A***, solve the system ***Ax*** = ***b*** with three right-hand sides, and compute the determinant of ***A***. (Default values for pivot processing are not used for IPARM because ***A*** is negative-definite.) Input matrix ***A***, shown here, is stored in diagonal-out skyline storage mode:

```
        | -2.0  -2.0  -2.0   0.0   0.0   0.0    0.0   0.0    0.0 |
        | -2.0  -4.0  -4.0  -2.0  -2.0   0.0    0.0   0.0   -2.0 |
        | -2.0  -4.0  -6.0  -4.0  -4.0   0.0   -2.0   0.0   -4.0 |
        | -2.0  -4.0  -6.0  -6.0  -6.0  -2.0   -4.0   0.0   -6.0 |
        |  0.0   0.0   0.0  -2.0  -4.0  -4.0   -4.0  -2.0   -4.0 |
        |  0.0  -2.0  -4.0  -6.0  -8.0  -6.0   -8.0  -4.0  -10.0 |
        |  0.0   0.0   0.0  -2.0  -4.0  -6.0   -8.0  -6.0   -8.0 |
         |  0.0   0.0   0.0  -2.0  -4.0  -6.0   -8.0  -8.0  -10.0 |
         | -2.0  -4.0  -6.0  -6.0  -8.0  -6.0  -10.0  -8.0  -16.0 |
```

Output matrix ***A***, shown here, is in ***LU*** factored form with ***U***$^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix ***A*** is:

```
          | -0.5  -2.0  -2.0   0.0   0.0   0.0   0.0   0.0   0.0 |
          |  1.0  -0.5  -2.0  -2.0  -2.0   0.0   0.0   0.0  -2.0 |
          |  1.0   1.0  -0.5  -2.0  -2.0   0.0  -2.0   0.0  -2.0 |
          |  1.0   1.0   1.0  -0.5  -2.0  -2.0  -2.0   0.0  -2.0 |
          |  0.0   0.0   0.0   1.0  -0.5  -2.0  -2.0  -2.0  -2.0 |
          |  0.0   1.0   1.0   1.0   1.0  -0.5  -2.0  -2.0  -2.0 |
          |  0.0   0.0   0.0   1.0   1.0   1.0  -0.5  -2.0  -2.0 |
          |  0.0   0.0   0.0   1.0   1.0   1.0   1.0  -0.5  -2.0 |
          |  1.0   1.0   1.0   1.0   1.0   1.0   1.0   1.0  -0.5 |
```

Call Statement and Input:

```
            N  AU  NU  IDU  AL  NL  IDL  IPARM  RPARM  AUX  NAUX  BX  LDBX MBX
            |  |   |    |   |   |    |     |      |     |    |    |    |    |
CALL DGKFS( 9, AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX,  57 , BX , 12 , 3 )
```

```
AU       =  (-2.0, -4.0, -2.0, -6.0, -4.0, -2.0, -6.0, -4.0, -2.0,
             -4.0, -6.0, -4.0, -2.0, -6.0, -4.0, -2.0, -8.0, -8.0,
             -4.0, -4.0, -2.0, -8.0, -6.0, -4.0, -2.0, -16.0, -10.0,
             -8.0, -10.0, -4.0, -6.0, -4.0, -2.0)
IDU      =  (1, 2, 4, 7, 10, 14, 17, 22, 26, 34)
AL       =  (0.0, 0.0, -2.0, 0.0, -4.0, -2.0, 0.0, -6.0, -4.0, -2.0,
             0.0, -2.0, 0.0, -8.0, -6.0, -4.0, -2.0, 0.0, -6.0, -4.0,
             -2.0, 0.0, -8.0, -6.0, -4.0, -2.0, 0.0, -8.0, -10.0,
             -6.0, -8.0, -6.0, -6.0, -4.0, -2.0)
IDL      =  (1, 2, 4, 7, 11, 13, 18, 22, 27, 36)
IPARM    =  (1, 10, 0, 0, 0, . , . , . , . , . , 1, 0, -1, -1, -1, -1, . ,
             . , . , . , . , . , . , . , . , . , . )
RPARM    =  ( . , . , . , . , . , . , . , . , . , . , 10^-15, . , . , . ,
              . , . , . , . , . , . , . , . , . , . , . , . , . , . )
BX       = (same as input BX in Example 1
```

Output:

```
AU       =  (-0.5, -0.5, -2.0, -0.5, -2.0, -2.0, -0.5, -2.0, -2.0,
             -0.5, -2.0, -2.0, -2.0, -0.5, -2.0, -2.0, -0.5, -2.0,
             -2.0, -2.0, -2.0, -0.5, -2.0, -2.0, -2.0, -0.5, -2.0,
             -2.0, -2.0, -2.0, -2.0, -2.0, -2.0)
IDU      =(same as input)
AL       =  (0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0,
             1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0,
             1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
             1.0, 1.0)
IDL      =(same as input)
IPARM    =  (1, 10, 0, 0, 0, . , . , . , . , . , 1, 0, -1, -1, -1, -1, 9,
             . , . , . , . , . , 9, 0, 0, 0, 0)
RPARM    =  ( . , . , . , . , . , . , . , . , . , . , 10^-15, . , . , . ,
. , . , . , 8.0, -5.12, 2.0, . , . , . , . , . , . , . , . )
```

```
         | -1.00  -2.00  -3.00 |
         | -1.00  -2.00  -3.00 |
         | -1.00  -2.00  -3.00 |
         | -1.00  -2.00  -3.00 |
         | -1.00  -2.00  -3.00 |
BX   =   | -1.00  -2.00  -3.00 |
         | -1.00  -2.00  -3.00 |
         | -1.00  -2.00  -3.00 |
         | -1.00  -2.00  -3.00 |
         |   .      .      .   |
         |   .      .      .   |
         |   .      .      .   |
```

Example 4

This example shows how to factor the first six rows and columns, referred to as matrix ***A1***, of the 9 by 9 general sparse matrix ***A*** from Example 1 and compute the determinant of ***A1***. Input matrix ***A1***, shown here, is stored in diagonal-out skyline storage mode. Input matrix ***A1*** is:

```
| 2.0  2.0  2.0  0.0  0.0  0.0 |
| 2.0  4.0  4.0  2.0  2.0  0.0 |
| 2.0  4.0  6.0  4.0  4.0  0.0 |
| 2.0  4.0  6.0  6.0  6.0  2.0 |
| 0.0  0.0  0.0  2.0  4.0  4.0 |
| 0.0  2.0  4.0  6.0  8.0  6.0 |
```

Output matrix ***A1***, shown here, is in ***LU*** factored form with $\boldsymbol{U}^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. Output matrix ***A1*** is:

```
| 0.5  2.0  2.0  0.0  0.0  0.0 |
| 1.0  0.5  2.0  2.0  2.0  0.0 |
| 1.0  1.0  0.5  2.0  2.0  0.0 |
| 1.0  1.0  1.0  0.5  2.0  2.0 |
| 0.0  0.0  0.0  1.0  0.5  2.0 |
| 0.0  1.0  1.0  1.0  1.0  0.5 |
```

Call Statement and Input:

```
            N  AU  NU  IDU  AL  NL  IDL  IPARM  RPARM  AUX  NAUX  BX   LDBX   MBX
            |  |   |    |   |   |    |     |      |     |    |    |     |      |
CALL DGKFS( 6, AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX,  45 , BX , LDBX , MBX )
```

AU =(same as input AU in Example 1)
`IDU = (1, 2, 4, 7, 10, 14, 17)`
AL =(same as input AL in Example 1)
`IDL = (1, 2, 4, 7, 11, 13, 18)`
`IPARM = (1, 11, 0, 0, 0, . , . , . , . , 0, . , . , . , . , . ,`
`. , . , . , . , . , . , . , . , . , . )`
RPARM =(not relevant)
BX =(not relevant)
LDBX =(not relevant)
MBX =(not relevant)

Output:

```
AU       =  (0.5, 0.5, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 0.5,
2.0,
             2.0, 2.0, 0.5, 2.0, 2.0, 8.0, 8.0, 4.0, 4.0, 2.0, 8.0,
             6.0, 4.0, 2.0, 16.0, 10.0, 8.0, 10.0, 4.0, 6.0, 4.0, 2.0)
IDU      =(same as input)
AL       =  (0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0,
             1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 6.0, 4.0, 2.0, 0.0,
             8.0, 6.0, 4.0, 2.0, 0.0, 8.0, 10.0, 6.0, 8.0, 6.0, 6.0,
             4.0, 2.0)
IDL      =(same as input)
IPARM    =  (1, 11, 0, 0, 0, . , . , . , . , 0, . , . , . , . , . , 3,
             . , . , . , . , 0, 0, 0, 0, 6)
RPARM    =  ( . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
             . , 3.0, 6.4, 1.0, . , . , . , . , . , . , . )
BX       =(same as input)
LDBX     =(same as input)
MBX      =(same as input)
```

Example 5

This example shows how to do a partial factorization of the 9 by 9 general sparse matrix ***A*** from Example 1, where the first six rows and columns were factored in Example 4. It factors the remaining three rows and columns and computes the determinant of that part of the matrix. The input matrix, referred to as ***A2***, shown here, is made up of the output factored matrix ***A1*** plus the three remaining unfactored rows and columns of matrix ***A***. Matrix ***A2*** is:

```
            | 0.5  2.0  2.0  0.0  0.0  0.0   0.0  0.0   0.0 |
            | 1.0  0.5  2.0  2.0  2.0  0.0   0.0  0.0   2.0 |
            | 1.0  1.0  0.5  2.0  2.0  0.0   2.0  0.0   4.0 |
            | 1.0  1.0  1.0  0.5  2.0  2.0   4.0  0.0   6.0 |
            | 0.0  0.0  0.0  1.0  0.5  2.0   4.0  2.0   4.0 |
            | 0.0  1.0  1.0  1.0  1.0  0.5   8.0  4.0  10.0 |
            | 0.0  0.0  0.0  2.0  4.0  6.0   8.0  6.0   8.0 |
            | 0.0  0.0  0.0  2.0  4.0  6.0   8.0  8.0  10.0 |
            | 2.0  4.0  6.0  6.0  8.0  6.0  10.0  8.0  16.0 |
```

Both parts of input matrix ***A2*** are stored in diagonal-out skyline storage mode.

Output matrix ***A2*** is the same as output matrix ***A*** in Example 1 and is stored in diagonal-out skyline storage mode.

Call Statement and Input:

```
            N  AU  NU  IDU  AL  NL  IDL  IPARM  RPARM  AUX  NAUX  BX   LDBX   MBX
            |  |   |    |   |   |    |     |      |     |    |    |     |      |
CALL DGKFS( 9, AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX,  45 , BX , LDBX , MBX )
```

```
AU       =(same as output AU in Example 4)
IDU      =(same as input IDU in Example 1)
AL       =(same as output AL in Example 4)
IDL      =(same as input IDL in Example 1)
IPARM    =  (1, 11, 6, 0, 0, . , . , . , . , 0, . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . )
RPARM    =(not relevant)
BX       =(not relevant)
LDBX     =(not relevant)
MBX      =(not relevant)
```

Output:

```
AU       =(same as output AU in Example 1)
IDU      =(same as output IDU in Example 1)
AL       =(same as output AL in Example 1)
IDL      =(same as output IDL in Example 1)
IPARM    =  (1, 11, 6, 0, 0, . , . , . , . , 0, . , . , . , . , . , 9,
             . , . , . , . , 0, 0, 0, 0, 3)
RPARM    =  ( . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
             . , 8.0, 8.0, 0.0, . , . , . , . , . , . , . , . )
BX       =(same as input)
LDBX     =(same as input)
MBX      =(same as input)
```

Example 6

This example shows how to solve the system ***Ax*** = ***b*** with one right-hand side for a general sparse matrix ***A***. Input matrix ***A***, used here, is the same as factored output matrix ***A*** from Example 1, stored in profile-in skyline storage mode. Here, output matrix ***A*** is unchanged on output and is stored in profile-in skyline storage mode.

Call Statement and Input:

```
            N  AU  NU  IDU  AL  NL  IDL  IPARM  RPARM  AUX NAUX  BX  LDBX MBX
            |  |   |    |   |   |    |     |      |     |   |     |    |    |
CALL DGKFS( 9, AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX, 49 , BX , 9 ,  1 )
```

```
AU       =  (0.5, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0,
2.0,
             2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 2.0, 2.0, 0.5, 2.0,
             2.0, 2.0, 0.5, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 0.5)
IDU      =  (1, 3, 6, 9, 13, 16, 21, 25, 33, 34)
AL       =  (0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0,
             0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0,
             1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
             1.0, 0.0)
```

```
IDL      =  (1, 3, 6, 10, 12, 17, 21, 26, 35, 36)
IPARM    =  (1, 2, 0, 1, 1, . , . , . , . , . , . , . , . , . , . , . , . ,
              . , . , . , . , . , . , . , . , . , . , . , . , . )
RPARM    =(not relevant)
BX       =  (12.0, 58.0, 114.0, 176.0, 132.0, 294.0, 240.0, 274.0,
              406.0)
```

Output:

```
AU       =(same as input)
IDU      =(same as input)
AL       =(same as input)
IDL      =(same as input)
IPARM    =(same as input)
RPARM    =(not relevant)
BX       =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0)
```

DSKFS (Symmetric Sparse Matrix Factorization, Determinant, and Solve Using Skyline Storage Mode)

Purpose

This subroutine can perform either or both of the following functions for symmetric sparse matrix ***A***, stored in skyline storage mode, and for vectors ***x*** and ***b***:

- Factor ***A*** and, optionally, compute the determinant of ***A***.
- Solve the system ***Ax*** = ***b*** using the results of the factorization of matrix ***A***, produced on this call or a preceding call to this subroutine.

You have the choice of using either Gaussian elimination or Cholesky decomposition. You also have the choice of using profile-in or diagonal-out skyline storage mode for ***A*** on input or output.

Note: The input to the solve performed by this subroutine must be the output from the factorization performed by this subroutine.

Syntax

Language	Syntax
Fortran	CALL DSKFS (*n*, *a*, *na*, *idiag*, *iparm*, *rparm*, *aux*, *naux*, *bx*, *ldbx*, *mbx*)
C and C++	dskfs (*n*, *a*, *na*, *idiag*, *iparm*, *rparm*, *aux*, *naux*, *bx*, *ldbx*, *mbx*);

On Entry

n

is the order of symmetric sparse matrix ***A***. Specified as: an integer; *n* ≥ 0.

a

is the array, referred to as A, containing one of three forms of the upper triangular part of symmetric sparse matrix ***A***, depending on the type of computation performed, where:

- If you are doing a **factor and solve** or a **factor only**, and if `IPARM(3)` = 0, then A contains the unfactored upper triangle of symmetric sparse matrix ***A***.
- If you are doing a **factor only**, and if `IPARM(3)` > 0, then A contains the partially factored upper triangle of symmetric sparse matrix ***A***. The first `IPARM(3)` columns in the upper triangle of ***A*** are already factored. The remaining columns are factored in this computation.
- If you are doing a **solve only**, then A contains the factored upper triangle of sparse matrix ***A***, produced by a preceding call to this subroutine.

In each case:

If `IPARM(4)` = 0, diagonal-out skyline storage mode is used for ***A***.

If `IPARM(4)` = 1, profile-in skyline storage mode is used for ***A***.

Specified as: a one-dimensional array of (at least) length *na*, containing long-precision real numbers.

na
is the length of array A.

Specified as: an integer; *na* ≥ 0 and *na* ≥ (`IDIAG`(*n*+1)-1).

idiag
is the array, referred to as `IDIAG`, containing the relative positions of the diagonal elements of matrix ***A*** (in one of its three forms) in array A.

Specified as: a one-dimensional array of (at least) length *n*+1, containing integers.

iparm
is an array of parameters, `IPARM`(*i*), where:

- `IPARM(1)` indicates whether certain default values for *iparm* and *rparm* are used by this subroutine, where:

  If `IPARM(1)` = 0, the following default values are used. For restrictions, see "Notes" on page 750.

  `IPARM(2)` = 0
  `IPARM(3)` = 0
  `IPARM(4)` = 0
  `IPARM(5)` = 0
  `IPARM(10)` = 0
  `IPARM(11)` = -1
  `IPARM(12)` = -1
  `IPARM(13)` = -1
  `IPARM(14)` = -1
  `IPARM(15)` = 0
  `RPARM(10)` = 10^{-12}

  If `IPARM(1)` = 1, the default values are not used.

- `IPARM(2)` indicates the type of computation performed by this subroutine. The following table gives the `IPARM(2)` values for each variation:

Type of Computation	**Gaussian Elimination *Ax* = *b***	**Gaussian Elimination *Ax* = *b* and Determinant(*A*)**	**Cholesky Decomposition *Ax* = *b***	**Cholesky Decomposition *Ax* = *b* and Determinant(*A*)**
Factor and Solve	0	10	100	110
Factor Only	1	11	101	111
Solve Only	2	N/A	102	N/A

- `IPARM(3)` indicates whether a full or partial factorization is performed on matrix ***A***, where:

  If `IPARM(3)` = 0, and:

  If you are doing a **factor and solve** or a **factor only**, then a full factorization is performed for matrix ***A*** on rows and columns 1 through *n*.

  If you are doing a **solve only**, this argument has no effect on the computation, but must be set to 0.

  If `IPARM(3)` > 0, and you are doing a **factor only**, then a partial factorization is performed on matrix ***A***. Rows 1 through `IPARM(3)` of columns 1 through `IPARM(3)` in matrix ***A*** must be in factored form from a preceding call to this subroutine. The factorization is performed on rows `IPARM(3)+1` through *n* and columns `IPARM(3)+1` through *n*. For an illustration, see "Notes" on page 750.

- `IPARM(4)` indicates the input storage mode used for matrix ***A***. This determines the arrangement of data in arrays A and `IDIAG` on input, where:

  If `IPARM(4)` = 0, diagonal-out skyline storage mode is used.

If IPARM(4) = 1, profile-in skyline storage mode is used.

- IPARM(5) indicates the output storage mode used for matrix ***A***. This determines the arrangement of data in arrays A and IDAIG on output, where:

  If IPARM(5) = 0, diagonal-out skyline storage mode is used.

  If IPARM(5) = 1, profile-in skyline storage mode is used.

- IPARM(6) through IPARM(9) are reserved.
- IPARM(10) has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, then IPARM(10) indicates whether certain default values for *iparm* and *rparm* are used by this subroutine, where:

  If IPARM(10) = 0, the following default values are used.
  For restrictions, see "Notes" on page 750.

  IPARM(11) = -1
  IPARM(12) = -1
  IPARM(13) = -1
  IPARM(14) = -1
  IPARM(15) = 0
  RPARM(10) = 10^{-12}

  If IPARM(10) = 1, the default values are not used.

  If you are doing a **solve only**, this argument is not used.

- IPARM(11) through IPARM(15) have the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, then IPARM(11) through IPARM(15) control the type of processing to apply to pivot elements occurring in regions 1 through 5, respectively. The pivot elements are d_{kk} for Gaussian elimination and r_{kk} for Cholesky decomposition for $k = 1, n$ when doing a full factorization, and they are k = IPARM(3)+1, n when doing a partial factorization. The region in which a pivot element falls depends on the sign and magnitude of the pivot element. The regions are determined by RPARM(10). For a description of the regions and associated pivot values, see "Notes" on page 750. For each region i for i = 1,5, where the pivot occurs in region i, the processing applied to the pivot element is determined by IPARM(10+i), where:

  If IPARM(10+i) = -1, the pivot element is trapped and computational error 2126 is generated. See "Error conditions" on page 752.

  If IPARM(10+i) = 0, processing continues normally.

  Note: A value of 0 is not permitted for region 3, because if processing continues, a divide-by-zero exception occurs. In addition, if you are doing a Cholesky decomposition, a value of 0 is not permitted in regions 1 and 2, because a square root exception occurs.

  If IPARM(10+i) = 1, the pivot element is replaced with the value in RPARM(10+i), and processing continues normally.

  If you are doing a **solve only**, these arguments are not used.

- IPARM(16) through IPARM(25), see On Return.

Specified as: a one-dimensional array of (at least) length 25, containing integers, where:

IPARM(1) = 0 or 1
IPARM(2) = 0, 1, 2, 10, 11, 100, 101, 102, 110, or 111
If IPARM(2) = 0, 2, 10, 100, 102, or 110, then IPARM(3) = 0
If IPARM(2) = 1, 11, 101, or 111, then 0 ≤ IPARM(3) ≤ n
IPARM(4), IPARM(5) = 0 or 1
If IPARM(2) = 0, 1, 10, or 11, then:

IPARM(10) = 0 or 1

IPARM(11), IPARM(12) = -1, 0, or 1
IPARM(13) = -1 or 1
IPARM(14), IPARM(15) = -1, 0, or 1

If IPARM(2) = 100, 101, 110, or 111, then:

IPARM(10) = 0 or 1
IPARM(11), IPARM(12), IPARM(13) = -1 or 1
IPARM(14), IPARM(15) = -1, 0, or 1

rparm
is an array of parameters, RPARM(*i*), where:

- RPARM(1) through RPARM(9) are reserved.
- RPARM(10) has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, RPARM(10) is the tolerance value for small pivots. This sets the bounds for the pivot regions, where pivots are processed according to the options you specify for the five regions in IPARM(11) through IPARM(15), respectively. The suggested value is $10^{-15} \le$ IPARM(10) ≤ 1.

  If you are doing a **solve only**, this argument is not used.
- RPARM(11) through RPARM(15) have the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, RPARM(11) through RPARM(15) are the fix-up values to use for the pivots in regions 1 through 5, respectively. For each RPARM(10+*i*) for *i* = 1,5, where the pivot occurs in region *i*:

  If IPARM(10+*i*) = 1, the pivot is replaced with RPARM(10+*i*), where |RPARM(10+*i*)| should be a sufficiently large nonzero value to avoid overflow when calculating the reciprocal of the pivot. For Gaussian elimination, the suggested value is $10^{-15} \le$ |RPARM(10+*i*)| ≤ 1. For Cholesky decomposition, the value must be RPARM(10+*i*) > 0.

  If IPARM(10+*i*) ≠ 1, RPARM(10+*i*) is not used.

  If you are doing a **solve only**, these arguments are not used.
- RPARM(16) through RPARM(25), see On Return.

Specified as: a one-dimensional array of (at least) length 25, containing long-precision real numbers, where if IPARM(2) = 0, 1, 10, 11, 100, 101, 110, or 111, then:

RPARM(10) ≥ 0.0

If IPARM(2) = 0, 1, 10, or 11, then RPARM(11) through RPARM(15) ≠ 0.0

If IPARM(2) = 100, 101, 110, or 111, then RPARM(11) through RPARM(15) > 0.0

aux
has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing long-precision real numbers.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, DSKFS dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, If you are doing a **factor only**

For 32-bit integer arguments
You can use *naux* ≥ *n*.

For 64-bit integer arguments
You can use *naux* ≥ 2*n*.

However, for optimal performance:

For 32-bit integer arguments
Use *naux* ≥ 3*n*.

For 64-bit integer arguments
Use *naux* ≥ 4*n*.

If you are doing a **factor and solve** or a **solve only**:

For 32-bit integer arguments
Use *naux* ≥ 3*n* + 4*mbx*.

For 64-bit integer arguments
Use *naux* ≥ 4*n* + 4*mbx*.

For further details on error handling and the special factor-only case, see "Notes" on page 750.

bx
has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *bx* is the array, containing the *mbx* right-hand side vectors ***b*** of the system ***Ax*** = ***b***. Each vector ***b*** is length *n* and is stored in the corresponding column of the array.

If you are doing a **factor only**, this argument is not used in the computation.

Specified as: an *ldbx* by (at least) *mbx* array, containing long-precision real numbers.

ldbx
has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *ldbx* is the leading dimension of the array specified for *bx*.

If you are doing a **factor only**, this argument is not used in the computation.

Specified as: an integer; *ldbx* ≥ *n* and:

If *mbx* ≠ 0, then *ldbx* > 0.

If *mbx* = 0, then *ldbx* ≥ 0.

mbx
has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *mbx* is the number of right-hand side vectors, ***b***, in the array specified for *bx*.

If you are doing a **factor only**, this argument is not used in the computation.

Specified as: an integer; *mbx* ≥ 0.

On Return

a
is the array, referred to as A, containing the upper triangular part of symmetric sparse matrix ***A*** in ***LDL***^{T} or ***R***^{T}***R*** factored form, where:

If `IPARM(5)` = 0, diagonal-out skyline storage mode is used for ***A***.

If `IPARM(5)` = 1, profile-in skyline storage mode is used for ***A***.

(If *mbx* = 0 and you are doing a solve only, then *a* is unchanged on output.) Returned as: a one-dimensional array of (at least) length *na*, containing long-precision real numbers.

idiag

is the array, referred to as IDIAG, containing the relative positions of the diagonal elements of the factored output matrix ***A*** in array A. (If *mbx* = 0 and you are doing a solve only, then *idiag* is unchanged on output.)

Returned as: a one-dimensional array of (at least) length *n*+1, containing integers.

iparm

is an array of parameters, IPARM(*i*), where:

- IPARM(1) through IPARM(15) are unchanged.
- IPARM(16) has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, and:

  If IPARM(16) = -1, your factorization did not complete successfully, resulting in computational error 2126.

  If IPARM(16) > 0, it is the row number *k*, in which the maximum absolute value of the ratio a_{kk}/d_{kk} for Gaussian elimination and a_{kk}/r_{kk} for Cholesky decomposition occurred, where:

  If IPARM(3) = 0, *k* can be any of the rows, 1 through *n*, in the full factorization.

  If IPARM(3) > 0, *k* can be any of the rows, IPARM(3)+1 through *n*, in the partial factorization.

  If you are doing a **solve only**, this argument is not used in the computation and is unchanged.
- IPARM(17) through IPARM(20) are reserved.
- IPARM(21) through IPARM(25) have the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, IPARM(21) through IPARM(25) have the following meanings for each region *i* for *i* = 1,5, respectively:

  If IPARM(20+*i*) = -1, your factorization did not complete successfully, resulting in computational error 2126.

  If IPARM(20+*i*) ≥ 0, it is the number of pivots in region *i* for the columns that were factored in matrix ***A***, where:

  If IPARM(3) = 0, columns 1 through *n* were factored in the full factorization.

  If IPARM(3) > 0, columns IPARM(3)+1 through *n* were factored in the partial factorization.

  If you are doing a **solve only**, these arguments are not used in the computation and are unchanged.

Returned as: a one-dimensional array of (at least) length 25, containing integers.

rparm

is an array of parameters, RPARM(*i*), where:

- RPARM(1) through RPARM(15) are unchanged.
- RPARM(16) has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, and:

  If RPARM(16) = 0.0, your factorization did not complete successfully, resulting in computational error 2126.

  If |RPARM(16)| > 0.0, it is the ratio for row *k*, a_{kk}/d_{kk} for Gaussian elimination and a_{kk}/r_{kk} for Cholesky decomposition, having the maximum absolute value. Row *k* is indicated in IPARM(16), and:

  If IPARM(3) = 0, the ratio corresponds to one of the rows, 1 through *n*, in the full factorization.

  If IPARM(3) > 0, the ratio corresponds to one of the rows, IPARM(3)+1 through *n*, in the partial factorization.

If you are doing a **solve only**, this argument is not used in the computation and is unchanged.

- RPARM(17) and RPARM(18) have the following meaning, where:

  If you are **computing the determinant** of matrix ***A***, then RPARM(17) is the mantissa, *detbas*, and RPARM(18) is the power of 10, *detpwr*, used to express the value of the determinant: $detbas(10^{detpwr})$, where $1 \le detbas < 10$. Also:

  If IPARM(3) = 0, the determinant is computed for columns 1 through *n* in the full factorization.

  If IPARM(3) > 0, the determinant is computed for columns IPARM(3)+1 through *n* in the partial factorization.

  If you are **not computing the determinant** of matrix ***A***, these arguments are not used in the computation and are unchanged.

- RPARM(19) through RPARM(25) are reserved.

Returned as: a one-dimensional array of (at least) length 25, containing long-precision real numbers.

bx

has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *bx* is the array, containing the *mbx* solution vectors ***x*** of the system ***Ax*** = ***b***. Each vector ***x*** is length *n* and is stored in the corresponding column of the array. (If *mbx* = 0, then *bx* is unchanged on output.)

If you are doing a **factor only**, this argument is not used in the computation and is unchanged.

Returned as: an *ldbx* by (at least) *mbx* array, containing long-precision real numbers.

Notes

1. When doing a **solve only**, you should specify the same factorization method in IPARM(2), Gaussian elimination or Cholesky decomposition, that you specified for your factorization on a previous call to this subroutine.
2. If you set either IPARM(1) = 0 or IPARM(10) = 0, indicating you want to use the default values for IPARM(11) through IPARM(15) and RPARM(10), then:
   - Matrix ***A*** must be positive definite.
   - No pivots are fixed, using RPARM(11) through RPARM(15) values.
   - No small pivots are tolerated; that is, the value should be |*pivot*| > RPARM(10).
3. Many of the input and output parameters for *iparm* and *rparm* are defined for the five pivot regions handled by this subroutine. The limits of the regions are based on RPARM(10), as shown in Figure 14 on page 750. The pivot values in each region are:

   Region 1: *pivot* < -RPARM(10)
   Region 2: -RPARM(10) ≤ *pivot* < 0
   Region 3: *pivot* = 0
   Region 4: 0 < *pivot* ≤ RPARM(10)
   Region 5: *pivot* > RPARM(10)

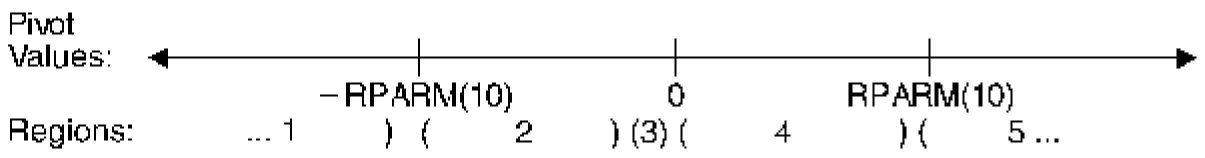


Figure 14: Five Pivot Regions

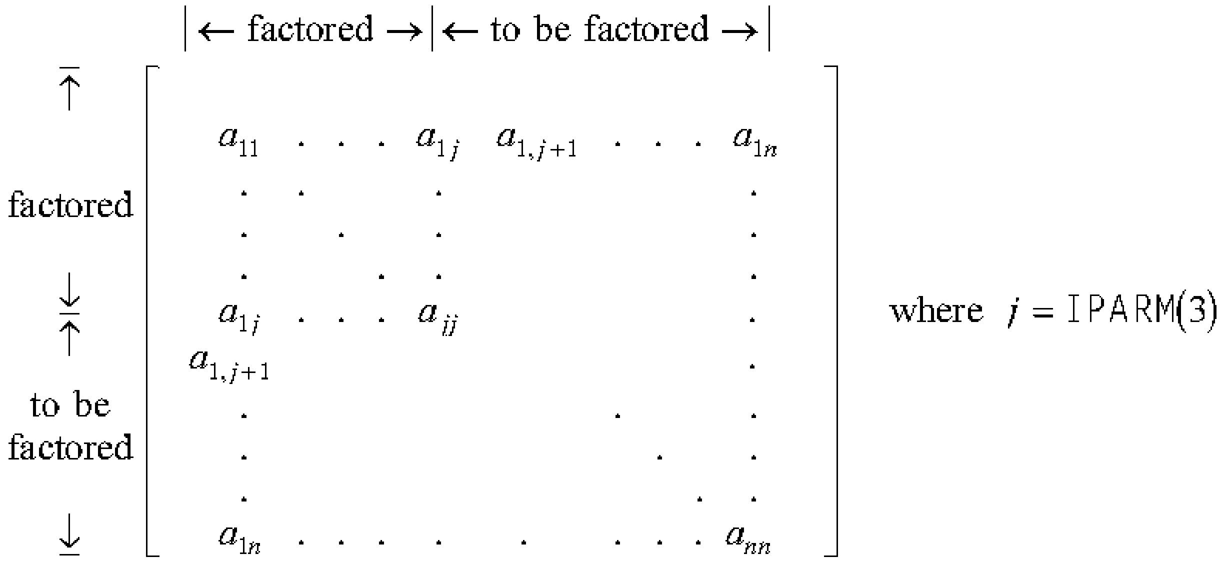
4. The IPARM(4) and IPARM(5) arguments allow you to specify the same or different skyline storage modes for your input and output arrays for matrix ***A***. This allows you to change storage modes as

needed. However, if you are concerned with performance, you should use diagonal-out skyline storage mode for both input and output, if possible, because there is less overhead.

For a description of how sparse matrices are stored in skyline storage mode, see "Profile-In Skyline Storage Mode" on page 119 and "Diagonal-Out Skyline Storage Mode" on page 117. Those descriptions use different array and variable names from the ones used here. To relate the two sets, use the following table:

Name Here	**Name in the Storage Description**
`A`	`AU`
na	*nu*
`IDIAG`	`IDU`

5. Following is an illustration of the portion of matrix ***A*** factored in the partial factorization when `IPARM(3)` > 0. In this case, the subroutine assumes that rows and columns 1 through `IPARM(3)` are already factored and that rows and columns `IPARM(3)+1` through *n* are to be factored in this computation.



You use the partial factorization function when, for design or storage reasons, you must factor the matrix ***A*** in stages. When doing a partial factorization, you must use the same skyline storage mode for all parts of the matrix as it is progressively factored.

6. Your various arrays must have no common elements; otherwise, results are unpredictable.
7. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

This subroutine can factor, compute the determinant of, and solve symmetric sparse matrix ***A***, stored in skyline storage mode. It can use either Gaussian elimination or Cholesky decomposition. For all computations, input matrix ***A*** can be stored in either diagonal-out or profile-in skyline storage mode. Output matrix ***A*** can also be stored in either of these modes and can be different from the mode used for input.

For Gaussian elimination, matrix ***A*** is factored into the following form using specified pivot processing:

$\boldsymbol{A} = \boldsymbol{LDL}^{\mathrm{T}}$

where:

D is a diagonal matrix.
L is a lower triangular matrix.

The transformed matrix ***A***, factored into its ***LDL***^{T} form, is stored in packed format in array A, such that the inverse of the diagonal matrix ***D*** is stored in the corresponding elements of array A. The off-diagonal elements of the unit upper triangular matrix ***L***^{T} are stored in the corresponding off-diagonal elements of array A.

For Cholesky decomposition, matrix ***A*** is factored into the following form using specified pivot processing:

A = ***R***^{T}***R***

where ***R*** is an upper triangular matrix

The transformed matrix ***A***, factored into its ***R***^{T}***R*** form, is stored in packed format in array A, such that the inverse of the diagonal elements of the upper triangular matrix ***R*** is stored in the corresponding elements of array A. The off-diagonal elements of matrix ***R*** are stored in the corresponding off-diagonal elements of array A.

The partial factorization of matrix ***A***, which you can do when you specify the factor-only option, assumes that the first IPARM(3) rows and columns are already factored in the input matrix. It factors the remaining *n*-IPARM(3) rows and columns in matrix ***A***. (See "Notes" on page 750 for an illustration.) It updates only the elements in array A corresponding to the part of matrix ***A*** that is factored.

The determinant can be computed with any of the factorization computations. With a full factorization, you get the determinant for the whole matrix. With a partial factorization, you get the determinant for only that part of the matrix factored in this computation.

The system ***Ax*** = ***b***, having multiple right-hand sides, is solved for ***x*** using the transformed matrix ***A*** produced by this call or a subsequent call to this subroutine.

See references ["12" on page 1253], ["20" on page 1254], ["33" on page 1254], ["58" on page 1256], ["85" on page 1257]. If *n* is 0, no computation is performed. If *mbx* is 0, no solve is performed.

Error conditions

Resource Errors

- Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.
- Unable to allocate internal work area.

Computational Errors

1. If a pivot occurs in region *i* for *i* = 1,5 and IPARM(10+*i*) = 1, the pivot value is replaced with RPARM(10+*i*), an attention message is issued, and processing continues.
2. Unacceptable pivot values occurred in the factorization of matrix ***A***.
   - One or more diagonal elements of ***D*** or ***R*** contains unacceptable pivots and no valid fixup is applicable. The row number *i* of the first unacceptable pivot element is identified in the computational error message.
   - The return code is set to 2.
   - *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2126 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

Input-Argument Errors

1. *n* < 0
2. *na* < 0

3. `IDIAG`(n+1) > na+1
4. `IDIAG`(i+1) ≤ `IDIAG`(i) for i = 1, n
5. `IDIAG`(i+1) > `IDIAG`(i)+i and `IPARM(4)` = 0 for i = 1, n
6. `IDIAG`(i) > `IDIAG`(i-1)+i and `IPARM(4)` = 1 for i = 2, n
7. `IPARM(1)` ≠ 0 or 1
8. `IPARM(2)` ≠ 0, 1, 2, 10, 11, 100, 101, 102, 110, or 111
9. `IPARM(3)` < 0
10. `IPARM(3)` > n
11. `IPARM(3)` > 0 and `IPARM(2)` ≠ 1, 11, 101, or 111
12. `IPARM(4)`, `IPARM(5)` ≠ 0 or 1
13. `IPARM(2)` = 0, 1, 10, or 11 and:

    `IPARM(10)` ≠ 0 or 1
    `IPARM(11)`, `IPARM(12)` ≠ -1, 0, or 1
    `IPARM(13)` ≠ -1 or 1
    `IPARM(14)`, `IPARM(15)` ≠ -1, 0, or 1
    `RPARM(10)` < 0.0
    `RPARM`(10+i) = 0.0 and `IPARM`(10+i) = 1 for i = 1,5
14. `IPARM(2)` = 100, 101, 110, or 111 and:

    `IPARM(10)` ≠ 0 or 1
    `IPARM(11)`, `IPARM(12)`, `IPARM(13)` ≠ -1 or 1
    `IPARM(14)`, `IPARM(15)` ≠ -1, 0, or 1
    `RPARM(10)` < 0.0
    `RPARM`(10+i) ≤ 0.0 and `IPARM`(10+i) = 1 for i = 1,5
15. `IPARM(2)` = 0, 2, 10, 100, 102, or 110 and:

    ldbx ≤ 0 and *mbx* ≠ 0 and *n* ≠ 0
    ldbx < 0 and *mbx* = 0
    ldbx < *n* and *mbx* ≠ 0
    mbx < 0
16. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows how to factor a 9 by 9 symmetric sparse matrix ***A*** and solve the system ***Ax*** = ***b*** with three right-hand sides. It uses Gaussian elimination. The default values are used for IPARM and RPARM. Input matrix ***A***, shown here, is stored in diagonal-out skyline storage mode. Matrix ***A*** is:

```
              | 1.0  1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0 |
              | 1.0  2.0  2.0  2.0  1.0  1.0  0.0  1.0  0.0 |
              | 1.0  2.0  3.0  3.0  2.0  2.0  0.0  2.0  0.0 |
              | 1.0  2.0  3.0  4.0  3.0  3.0  0.0  3.0  0.0 |
              | 0.0  1.0  2.0  3.0  4.0  4.0  1.0  4.0  0.0 |
              | 0.0  1.0  2.0  3.0  4.0  5.0  2.0  5.0  1.0 |
              | 0.0  0.0  0.0  0.0  1.0  2.0  3.0  3.0  2.0 |
              | 0.0  1.0  2.0  3.0  4.0  5.0  3.0  7.0  3.0 |
              | 0.0  0.0  0.0  0.0  0.0  1.0  2.0  3.0  4.0 |
```

Output matrix ***A***, shown here, is in ***LDL***^{T} factored form with ***D***$^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix ***A*** is:

```
              | 1.0  1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0 |
              | 1.0  1.0  1.0  1.0  1.0  1.0  0.0  1.0  0.0 |
              | 1.0  1.0  1.0  1.0  1.0  1.0  0.0  1.0  0.0 |
              | 1.0  1.0  1.0  1.0  1.0  1.0  0.0  1.0  0.0 |
              | 0.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  0.0 |
              | 0.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
              | 0.0  0.0  0.0  0.0  1.0  1.0  1.0  1.0  1.0 |
              | 0.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
              | 0.0  0.0  0.0  0.0  0.0  1.0  1.0  1.0  1.0 |
```

Call Statement and Input:

```
            N  A  NA  IDIAG  IPARM  RPARM  AUX NAUX  BX  LDBX MBX
            |  |  |     |      |      |     |   |    |    |    |
CALL DSKFS( 9, A, 33, IDIAG, IPARM, RPARM, AUX, 39 , BX , 12 , 3 )
```

```
A        =  (1.0, 2.0, 1.0, 3.0, 2.0, 1.0, 4.0, 3.0, 2.0, 1.0, 4.0,
             3.0, 2.0, 1.0, 5.0, 4.0, 3.0, 2.0, 1.0, 3.0, 2.0, 1.0,
             7.0, 3.0, 5.0, 4.0, 3.0, 2.0, 1.0, 4.0, 3.0, 2.0, 1.0)
IDIAG    =  (1, 2, 4, 7, 11, 15, 20, 23, 30, 34)
IPARM    =  (0, . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . , . , . )
```

RPARM =(not relevant)

```
        |  4.00   8.00  12.00 |
        | 10.00  20.00  30.00 |
        | 15.00  30.00  45.00 |
        | 19.00  38.00  57.00 |
        | 19.00  38.00  57.00 |
BX   =  | 23.00  46.00  69.00 |
        | 11.00  22.00  33.00 |
        | 28.00  56.00  84.00 |
        | 10.00  20.00  30.00 |
        |   .      .      .   |
        |   .      .      .   |
        |   .      .      .   |
```

Output:

```
A = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```
IDIAG =(same as input)
```
IPARM = (0, . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
. , 8, . , . , . , . , 0, 0, 0, 0, 9)
RPARM = ( . , . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
. , 7.0, . , . , . , . , . , . , . , . )
```

```
        | 1.00  2.00  3.00 |
        | 1.00  2.00  3.00 |
        | 1.00  2.00  3.00 |
        | 1.00  2.00  3.00 |
        | 1.00  2.00  3.00 |
BX   =  | 1.00  2.00  3.00 |
        | 1.00  2.00  3.00 |
        | 1.00  2.00  3.00 |
        | 1.00  2.00  3.00 |
        |  .     .     .   |
        |  .     .     .   |
        |  .     .     .   |
```

Example 2

This example shows how to factor the 9 by 9 symmetric sparse matrix ***A*** from Example 1, solve the system ***Ax*** = ***b*** with three right-hand sides, and compute the determinant of ***A***. It uses Gaussian elimination. The default values for pivot processing are used for IPARM. Input matrix ***A*** is stored in profile-in skyline storage mode. Output matrix ***A*** is in ***LDL***^{T} factored form with ***D***$^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. It is the same as output matrix ***A*** in Example 1.

Call Statement and Input:

```
             N  A  NA  IDIAG  IPARM  RPARM  AUX NAUX  BX  LDBX MBX
             |  |  |     |      |      |     |   |    |    |    |
CALL DSKFS( 9, A, 33, IDIAG, IPARM, RPARM, AUX, 39 , BX , 12 , 3 )
```

```
A        =  (1.0, 1.0, 2.0, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0, 4.0, 1.0,
             2.0, 3.0, 4.0, 1.0, 2.0, 3.0, 4.0, 5.0, 1.0, 2.0, 3.0,
             1.0, 2.0, 3.0, 4.0, 5.0, 3.0, 7.0, 1.0, 2.0, 3.0, 4.0)
IDIAG    =  (1, 3, 6, 10, 14, 19, 22, 29, 33, 34)
IPARM    =  (1, 10, 0, 1, 0, . , . , . , . , . , 0, . , . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . , . )
```

RPARM =(not relevant)

```
        |  4.00   8.00  12.00 |
        | 10.00  20.00  30.00 |
        | 15.00  30.00  45.00 |
        | 19.00  38.00  57.00 |
        | 19.00  38.00  57.00 |
BX   =  | 23.00  46.00  69.00 |
        | 11.00  22.00  33.00 |
        | 28.00  56.00  84.00 |
        | 10.00  20.00  30.00 |
        |   .      .      .   |
        |   .      .      .   |
        |   .      .      .   |
```

Output:

A =(same as output A in Example 1)
IDIAG =(same as input IDIAG in Example 1)
IPARM = (1, 10, 0, 1, 0, . , . , . , . , . , 0, . , . , . , . , . , . , 8,
. , . , . , . , . , 0, 0, 0, 0, 9)
RPARM = (. , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
. , 7.0, 1.0, 0.0, . , . , . , . , . , . , .)
BX =(same as output BX in Example 1)

Example 3

This example shows how to factor a 9 by 9 negative-definite symmetric sparse matrix ***A***, solve the system ***Ax*** = ***b*** with three right-hand sides, and compute the determinant of ***A***. It uses Gaussian elimination. (Default values for pivot processing are not used for IPARM because ***A*** is negative-definite.) Input matrix ***A***, shown here, is stored in diagonal-out skyline storage mode. Matrix ***A*** is:

```
| -1.0  -1.0  -1.0  -1.0   0.0   0.0   0.0   0.0   0.0 |
| -1.0  -2.0  -2.0  -2.0  -1.0  -1.0   0.0  -1.0   0.0 |
| -1.0  -2.0  -3.0  -3.0  -2.0  -2.0   0.0  -2.0   0.0 |
| -1.0  -2.0  -3.0  -4.0  -3.0  -3.0   0.0  -3.0   0.0 |
|  0.0  -1.0  -2.0  -3.0  -4.0  -4.0  -1.0  -4.0   0.0 |
|  0.0  -1.0  -2.0  -3.0  -4.0  -5.0  -2.0  -5.0  -1.0 |
|  0.0   0.0   0.0   0.0  -1.0  -2.0  -3.0  -3.0  -2.0 |
|  0.0  -1.0  -2.0  -3.0  -4.0  -5.0  -3.0  -7.0  -3.0 |
|  0.0   0.0   0.0   0.0   0.0  -1.0  -2.0  -3.0  -4.0 |
```

Output matrix ***A***, shown here, is in ***LDL***^{T} factored form with ***D***$^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix ***A*** is:

```
| -1.0   1.0   1.0   1.0   0.0   0.0   0.0   0.0   0.0 |
|  1.0  -1.0   1.0   1.0   1.0   1.0   0.0   1.0   0.0 |
|  1.0   1.0  -1.0   1.0   1.0   1.0   0.0   1.0   0.0 |
          |  1.0   1.0   1.0  -1.0   1.0   1.0   0.0   1.0   0.0 |
          |  0.0   1.0   1.0   1.0  -1.0   1.0   1.0   1.0   0.0 |
          |  0.0   1.0   1.0   1.0   1.0  -1.0   1.0   1.0   1.0 |
          |  0.0   0.0   0.0   0.0   1.0   1.0  -1.0   1.0   1.0 |
          |  0.0   1.0   1.0   1.0   1.0   1.0   1.0  -1.0   1.0 |
          |  0.0   0.0   0.0   0.0   0.0   1.0   1.0   1.0  -1.0 |
```

Call Statement and Input:

```
           N  A  NA  IDIAG  IPARM  RPARM  AUX  NAUX  BX  LDBX MBX
           |  |  |     |      |      |     |    |    |    |    |
CALL DSKFS(9, A, 33, IDIAG, IPARM, RPARM, AUX,  39 , BX , 12 , 3 )
```

```
A        =  (-1.0, -2.0, -1.0, -3.0, -2.0, -1.0, -4.0, -3.0, -2.0,
             -1.0, -4.0, -3.0, -2.0, -1.0, -5.0, -4.0, -3.0, -2.0,
             -1.0, -3.0, -2.0, -1.0, -7.0, -3.0, -5.0, -4.0, -3.0,
             -2.0, -1.0, -4.0, -3.0, -2.0, -1.0)
IDIAG    =  (1, 2, 4, 7, 11, 15, 20, 23, 30, 34)
IPARM    =  (1, 10, 0, 0, 0, . , . , . , . , . , 1, 0, -1, -1, -1, -1, . ,
             . , . , . , . , . , . , . , . , . , . )
```

RPARM = (. , . , . , . , . , . , . , . , . ,10^{-15}, . , . ,
. , . , . , . , . , . , . , . , . , . , . , . , .)
BX =(same as input BX in Example 1)

Output:

A = (-1.0, -1.0, 1.0, -1.0, 1.0, 1.0, -1.0, 1.0,
1.0, 1.0,
-1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0,
1.0, -1.0 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0,
1.0)
IDIAG =(same as input)
IPARM = (1, 10, 0, 0, 0, . , . , . , . , . , 1, 0, -1, -1, -1, -1, 8,
. , . , . , . , 9, 0, 0, 0, 0)
RPARM = (. , . , . , . , . , . , . , . , . ,10^{-15}, . , . ,
. , . , . , 7.0, -1.0, 0.0, . , . , . , . , . , . , .)

```
        | -1.00  -2.00  -3.00 |
        | -1.00  -2.00  -3.00 |
        | -1.00  -2.00  -3.00 |
        | -1.00  -2.00  -3.00 |
        | -1.00  -2.00  -3.00 |
BX   =  | -1.00  -2.00  -3.00 |
        | -1.00  -2.00  -3.00 |
        | -1.00  -2.00  -3.00 |
        | -1.00  -2.00  -3.00 |
        |   .      .      .   |
        |   .      .      .   |
        |   .      .      .   |
```

Example 4

This example shows how to factor the first six rows and columns, referred to as matrix ***A1***, of the 9 by 9 symmetric sparse matrix ***A*** from Example 1 and compute the determinant of ***A1***. It uses Gaussian elimination. Input matrix ***A1***, shown here, is stored in diagonal-out skyline storage mode. Input matrix ***A1*** is:

```
                      | 1.0  1.0  1.0  1.0  0.0  0.0 |
                      | 1.0  2.0  2.0  2.0  1.0  1.0 |
                      | 1.0  2.0  3.0  3.0  2.0  2.0 |
                      | 1.0  2.0  3.0  4.0  3.0  3.0 |
                      | 0.0  1.0  2.0  3.0  4.0  4.0 |
                      | 0.0  1.0  2.0  3.0  4.0  5.0 |
```

Output matrix ***A1***, shown here, is in ***LDL***^{T} factored form with ***D***$^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. Output matrix ***A1*** is:

```
                    | 1.0  1.0  1.0  1.0  0.0  0.0 |
                    | 1.0  1.0  1.0  1.0  1.0  1.0 |
                    | 1.0  1.0  1.0  1.0  1.0  1.0 |
                    | 1.0  1.0  1.0  1.0  1.0  1.0 |
                    | 0.0  1.0  1.0  1.0  1.0  1.0 |
                    | 0.0  1.0  1.0  1.0  1.0  1.0 |
```

Call Statement and Input:

```
            N   A   NA   IDIAG   IPARM   RPARM   AUX  NAUX  BX   LDBX   MBX
            |   |   |     |       |       |       |    |    |     |      |
 CALL DSKFS (6 , A , 33 , IDIAG , IPARM , RPARM , AUX , 27 , BX , LDBX , MBX )
```

A =(same as input A in Example 1)
IDIAG = (1, 2, 4, 7, 11, 15, 20)
IPARM = (1, 11, 0, 0, 0, . , . , . , . , 0, . , . , . , . , . ,
. , . , . , . , . , . , . , . , .)
RPARM =(not relevant)
BX =(not relevant)
LDBX =(not relevant)
MBX =(not relevant)

Output:

A = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 3.0, 2.0, 1.0,
7.0, 3.0, 5.0, 4.0, 3.0, 2.0, 1.0, 4.0, 3.0, 2.0, 1.0)
IDIAG =(same as input)
IPARM = (1, 11, 0, 0, 0, . , . , . , . , 0, . , . , . , . , . , 6,
. , . , . , . , 0, 0, 0, 0, 6)
RPARM = (. , . , . , . , . , . , . , . , . , . , . , . , . ,
. , 5.0, 1.0, 0.0, . , . , . , . , . , . , .)
BX =(same as input)
LDBX =(same as input)
MBX =(same as input)

Example 5

This example shows how to do a partial factorization of the 9 by 9 symmetric sparse matrix ***A*** from Example 1, where the first six rows and columns were factored in Example 4. It factors the remaining three rows and columns and computes the determinant of that part of the matrix. It uses Gaussian elimination. The input matrix, referred to as ***A2***, shown here, is made up of the output factored matrix ***A1*** plus the three remaining unfactored rows and columns of matrix ***A*** Matrix ***A2*** is:

```
              | 1.0  1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0 |
              | 1.0  1.0  1.0  1.0  1.0  1.0  0.0  1.0  0.0 |
              | 1.0  1.0  1.0  1.0  1.0  1.0  0.0  2.0  0.0 |
              | 1.0  1.0  1.0  1.0  1.0  1.0  0.0  3.0  0.0 |
              | 0.0  1.0  1.0  1.0  1.0  1.0  1.0  4.0  0.0 |
              | 0.0  1.0  1.0  1.0  1.0  1.0  2.0  5.0  1.0 |
              | 0.0  0.0  0.0  0.0  1.0  2.0  3.0  3.0  2.0 |
              | 0.0  1.0  2.0  3.0  4.0  5.0  3.0  7.0  3.0 |
              | 0.0  0.0  0.0  0.0  0.0  1.0  2.0  3.0  4.0 |
```

Both parts of input matrix ***A2*** are stored in diagonal-out skyline storage mode.

Output matrix ***A2*** is the same as output matrix ***A*** in Example 1 and is stored in diagonal-out skyline storage mode.

Call Statement and Input:

```
             N   A   NA   IDIAG   IPARM   RPARM   AUX   NAUX  BX   LDBX   MBX
             |   |   |      |       |       |      |     |    |     |      |
CALL DSKFS (9 , A , 33 , IDIAG , IPARM , RPARM , AUX ,  27 , BX , LDBX , MBX )
```

A =(same as output A in Example 4)
IDIAG =(same as input IDIAG in Example 1)
IPARM = (1, 11, 6, 0, 0, . , . , . , . , 0, . , . , . , . , . ,
. , . , . , . , . , . , . , . , . , .)
RPARM =(not relevant)
BX =(not relevant)
LDBX =(not relevant)
MBX =(not relevant)

Output:

A =(same as output A in Example 1)

IDIAG =(same as output IDIAG in Example 1)
IPARM = (1, 11, 6, 0, 0, . , . , . , . , 0, . , . , . , . , . , 8,
. , . , . , . , 0, 0, 0, 0, 3)
RPARM = (. , . , . , . , . , . , . , . , . , . , . , . , . , . ,
. , 7.0, 1.0, 0.0, . , . , . , . , . , . , .)
BX =(same as input)
LDBX =(same as input)
MBX =(same as input)

Example 6

This example shows how to solve the system ***Ax*** = ***b*** with one right-hand side for a symmetric sparse matrix ***A***. Input matrix ***A***, used here, is the same as factored output matrix ***A*** from Example 1, stored in profile-in skyline storage mode. It specifies Gaussian elimination, as used in Example 1. Here, output matrix ***A*** is unchanged on output and is stored in profile-in skyline storage mode.

Call Statement and Input:

```
             N  A  NA  IDIAG  IPARM  RPARM  AUX NAUX  BX  LDBX MBX
             |  |  |     |      |      |     |   |    |    |    |
CALL DSKFS (9, A, 33, IDIAG, IPARM, RPARM, AUX, 31 , BX , 9 ,  1 )
```

A = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IDIAG = (1, 3, 6, 10, 14, 19, 22, 29, 33, 34)
IPARM = (1, 2, 0, 1, 1, . , . , . , . , . , . , . , . , . , . ,
. , . , . , . , . , . , . , . , . , .)
RPARM =(not relevant)
BX = (10.0, 38.0, 64.0, 87.0, 103.0, 133.0, 80.0, 174.0, 80.0)

Output:

A =(same as input)
IDIAG =(same as input)
IPARM =(same as input)
APARM =(same as input)
BX = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0)

Example 7

This example shows how to factor a 9 by 9 symmetric sparse matrix ***A*** and solve the system ***Ax*** = ***b*** with four right-hand sides. It uses Cholesky decomposition. Input matrix ***A***, shown here, is stored in profile-in skyline storage mode Matrix ***A*** is:

```
          | 1.0  1.0   1.0   0.0   1.0   0.0   0.0   0.0   1.0 |
          | 1.0  5.0   3.0   0.0   3.0   0.0   0.0   0.0   3.0 |
          | 1.0  3.0  11.0   3.0   5.0   3.0   3.0   0.0   5.0 |
          | 0.0  0.0   3.0  17.0   5.0   5.0   5.0   0.0   5.0 |
          | 1.0  3.0   5.0   5.0  29.0   7.0   7.0   0.0   9.0 |
          | 0.0  0.0   3.0   5.0   7.0  39.0   9.0   6.0   9.0 |
          | 0.0  0.0   3.0   5.0   7.0   9.0  53.0   8.0  11.0 |
          | 0.0  0.0   0.0   0.0   0.0   6.0   8.0  66.0  10.0 |
          | 1.0  3.0   5.0   5.0   9.0   9.0  11.0  10.0  89.0 |
```

Output matrix ***A***, shown here, is in $\boldsymbol{R}^T\boldsymbol{R}$ factored form with the inverse of the diagonal of ***R*** on the diagonal, and is stored in profile-in skyline storage mode. Matrix ***A*** is:

```
           | 1.0  1.0   1.0  0.0  1.0   0.0   0.0   0.0   1.0 |
           | 1.0   .5   1.0  0.0  1.0   0.0   0.0   0.0   1.0 |
           | 1.0  1.0  .333  1.0  1.0   1.0   1.0   0.0   1.0 |
           | 0.0  0.0   1.0  .25  1.0   1.0   1.0   0.0   1.0 |
           | 1.0  1.0   1.0  1.0   .2   1.0   1.0   0.0   1.0 |
           | 0.0  0.0   1.0  1.0  1.0  .167   1.0   1.0   1.0 |
           | 0.0  0.0   1.0  1.0  1.0   1.0  .143   1.0   1.0 |
           | 0.0  0.0   0.0  0.0  0.0   1.0   1.0  .125   1.0 |
           | 1.0  1.0   1.0  1.0  1.0   1.0   1.0   1.0  .111 |
```

Call Statement and Input:

```
            N  A  NA  IDIAG  IPARM  RPARM  AUX NAUX  BX  LDBX MBX
            |  |  |     |      |      |     |   |    |    |    |
CALL DSKFS( 9, A, 34, IDIAG, IPARM, RPARM, AUX, 43 , BX , 10 , 4 )
```

```
A        =  (1.0, 1.0, 5.0, 1.0, 3.0, 11.0, 3.0, 17.0, 1.0, 3.0, 5.0,
             5.0, 29.0, 3.0, 5.0, 7.0, 39.0, 3.0, 5.0, 7.0, 9.0, 53.0,
             6.0, 8.0, 66.0, 1.0, 3.0, 5.0, 5.0, 9.0, 9.0, 11.0, 10.0,
             89.0)
IDIAG    =  (1, 3, 6, 8, 13, 17, 22, 25, 34, 35)
IPARM    =  (1, 110, 0, 1, 1, . , . , . , . , . , 0, . , . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . , . )
```

RPARM =(not relevant)

```
        |   5.00   10.00   15.00   20.00 |
        |  15.00   30.00   45.00   60.00 |
        |  34.00   68.00  102.00  136.00 |
        |  40.00   80.00  120.00  160.00 |
BX   =  |  66.00  132.00  198.00  264.00 |
        |  78.00  156.00  234.00  312.00 |
        |  96.00  192.00  288.00  384.00 |
        |  90.00  180.00  270.00  360.00 |
        | 142.00  284.00  426.00  568.00 |
        |    .       .       .       .   |
```

Output:

```
A = (1.0, 1.0, .5, 1.0, 1.0, .333, 1.0, .25, 1.0, 1.0,
1.0,
1.0, .2, 1.0, 1.0, 1.0, .167, 1.0, 1.0, 1.0, 1.0, .143,
1.0, 1.0, .125, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
.111)
IDIAG =(same as input)
IPARM = (1, 110, 0, 1, 1, . , . , . , . , . , 0, . , . , . , . , . ,
9, . , . , . , . , 0, 0, 0, 0, 9)
RPARM = ( . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
. , 9.89, 1.32, 11.0, . , . , . , . , . , . , . )
```

```
        | 1.00  2.00  3.00  4.00 |
        | 1.00  2.00  3.00  4.00 |
        | 1.00  2.00  3.00  4.00 |
        | 1.00  2.00  3.00  4.00 |
BX   =  | 1.00  2.00  3.00  4.00 |
        | 1.00  2.00  3.00  4.00 |
        | 1.00  2.00  3.00  4.00 |
        | 1.00  2.00  3.00  4.00 |
        | 1.00  2.00  3.00  4.00 |
        |  .     .     .     .   |
```

DSRIS (Iterative Linear System Solver for a General or Symmetric Sparse Matrix Stored by Rows)

Purpose

This subroutine solves a general or symmetric sparse linear system of equations, using an iterative algorithm, with or without preconditioning. The methods include conjugate gradient (CG), conjugate gradient squared (CGS), generalized minimum residual (GMRES), more smoothly converging variant of the CGS method (Bi-CGSTAB), or transpose-free quasi-minimal residual method (TFQMR). The preconditioners include an incomplete LU factorization, an incomplete Cholesky factorization (for positive definite symmetric matrices), diagonal scaling, or symmetric successive over-relaxation (SSOR) with two possible choices for the diagonal matrix: one uses the absolute values sum of the input matrix, and the other uses the diagonal obtained from the LU factorization. The sparse matrix is stored using storage-by-rows for general matrices and upper- or lower-storage-by-rows for symmetric matrices. Matrix ***A*** and vectors ***x*** and ***b*** are used:

Ax = ***b***

where ***A***, ***x***, and ***b*** contain long-precision real numbers.

Syntax

Language	Syntax
Fortran	CALL DSRIS (*stor*, *init*, *n*, *ar*, *ja*, *ia*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	dsris (*stor*, *init*, *n*, *ar*, *ja*, *ia*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

stor

indicates the form of sparse matrix ***A*** and the storage mode used, where:

If *stor* = 'G', ***A*** is a general sparse matrix, stored using storage-by-rows.

If *stor* = 'U', ***A*** is a symmetric sparse matrix, stored using upper-storage-by-rows.

If *stor* = 'L', ***A*** is a symmetric sparse matrix, stored using lower-storage-by-rows.

Specified as: a single character. It must be 'G', 'U', or 'L'.

init

indicates the type of computation to be performed, where:

If *init* = 'I', the preconditioning matrix is computed, the internal representation of the sparse matrix is generated, and the iteration procedure is performed. The coefficient matrix and preconditioner in internal format are saved in *aux1*.

If *init* = 'S', the iteration procedure is performed using the coefficient matrix and the preconditioner in internal format, stored in *aux1*, created in a preceding call to this subroutine with *init* = 'I'. You use this option to solve the same matrix for different right-hand sides, ***b***, optimizing your performance. As long as you do not change the coefficient matrix and preconditioner in *aux1*, any number of calls can be made with *init* = 'S'.

Specified as: a single character. It must be 'I' or 'S'.

n

is the order of the linear system ***Ax*** = ***b*** and the number of rows and columns in sparse matrix ***A***.

Specified as: an integer; $n \geq 0$.

ar

is the sparse matrix ***A*** of order *n*, stored by rows in an array, referred to as AR. The *stor* argument indicates the storage variation used for storing matrix ***A***.

Specified as: a one-dimensional array, containing long-precision real numbers. The number of elements in this array can be determined by subtracting 1 from the value in `IA`(*n*+1).

ja

is the array, referred to as `JA`, containing the column numbers of each nonzero element in sparse matrix **A**.

Specified as: a one-dimensional array, containing integers; 1 ≤ (`JA` elements) ≤ *n*. The number of elements in this array can be determined by subtracting 1 from the value in `IA`(*n*+1).

ia

is the row pointer array, referred to as `IA`, containing the starting positions of each row of matrix ***A*** in array AR and one position past the end of array AR. Specified as: a one-dimensional array of (at least) length *n*+1, containing integers; `IA`(*i*+1) ≥ `IA`(*i*) for *i* = 1, *n*+1.

b

is the vector ***b*** of length *n*, containing the right-hand side of the matrix problem.

Specified as: a one-dimensional array of (at least) length *n*, containing long-precision real numbers.

x

is the vector ***x*** of length *n*, containing your initial guess of the solution of the linear system.

Specified as: a one-dimensional array of (at least) length *n*, containing long-precision real numbers. The elements can have any value, and if no guess is available, the value can be zero.

iparm

is an array of parameters, `IPARM`(*i*), where:

- `IPARM(1)` controls the number of iterations.

  If `IPARM(1)` > 0, `IPARM(1)` is the maximum number of iterations allowed.

  If `IPARM(1)` = 0, the following default values are used:

  `IPARM(1)` = 300
  `IPARM(2)` = 4
  `IPARM(4)` = 4
  `IPARM(5)` = 1
  `RPARM(1)` = 10^{-6}
  `RPARM(2)` = 1

- `IPARM(2)` is the flag used to select the iterative procedure used in this subroutine.

  If `IPARM(2)` = 1, the conjugate gradient (CG) method is used. Note that this algorithm should only be used with positive definite symmetric matrices.

  If `IPARM(2)` = 2, the conjugate gradient squared (CGS) method is used.

  If `IPARM(2)` = 3, the generalized minimum residual (GMRES) method, restarted after *k* steps, is used.

If `IPARM(2)` = 4, the more smoothly converging variant of the CGS method (Bi-CGSTAB) is used.

If `IPARM(2)` = 5, the transpose-free quasi-minimal residual method (TFQMR) is used.

- `IPARM(3)` has the following meaning, where:

  If `IPARM(2)` ≠ 3, then `IPARM(3)` is not used.

  If `IPARM(2)` = 3, then `IPARM(3)` = *k*, where *k* is the number of steps after which the generalized minimum residual method is restarted. A value for *k* in the range of 5 to 10 is suitable for most problems.
- `IPARM(4)` is the flag that determines the type of preconditioning.

  If `IPARM(4)` = 1, the system is not preconditioned.

  If `IPARM(4)` = 2, the system is preconditioned by a diagonal matrix.

  If `IPARM(4)` = 3, the system is preconditioned by SSOR splitting with the diagonal given by the absolute values sum of the input matrix.

  If `IPARM(4)` = 4, the system is preconditioned by an incomplete LU factorization.

  If `IPARM(4)` = 5, the system is preconditioned by SSOR splitting with the diagonal given by the incomplete LU factorization.

  Note: The multithreaded version of DSRIS only runs on multiple threads when `IPARM(4)` = 1 or 2.
- `IPARM(5)` is the flag used to select the stopping criterion used in the computation, where the following items are used in the definitions of the stopping criteria below:
  - ε is the desired relative accuracy and is stored in `RPARM(1)`.
  - x_j is the solution found at the *j*-th iteration.
  - r_j and r_0 are the preconditioned residuals obtained at iterations *j* and 0, respectively. (The residual at iteration *j* is given by $\boldsymbol{b}$-$\boldsymbol{A}x_j$.)

  If `IPARM(5)` = 1, the iterative method is stopped when:

  $\|r_j\|_2 / \|x_j\|_2 < \varepsilon$

  Note: `IPARM(5)` = 1 is the default value assumed by ESSL if you do not specify one of the values described here; therefore, if you do not update your program to set an `IPARM(5)` value, you, by default, use the above stopping criterion.

  If `IPARM(5)` = 2, the iterative method is stopped when:

  $\|r_j\|_2 / \|r_0\|_2 < \varepsilon$

  If `IPARM(5)` = 3, the iterative method is stopped when:

  $\|x_j - x_{j-1}\|_2 / \|x_j\|_2 < \varepsilon$

  Note: Stopping criterion 3 performs poorly with the TFQMR method; therefore, if you specify TFQMR (`IPARM(2)` = 5), you should not specify stopping criterion 3.
- `IPARM(6)`, see On Return.

Specified as: an array of (at least) length 6, containing integers, where:

`IPARM(1)` ≥ 0
`IPARM(2)` = 1, 2, 3, 4, or 5
If `IPARM(2)` = 3, then `IPARM(3)` > 0
`IPARM(4)` = 1, 2, 3, 4, or 5
`IPARM(5)` = 1, 2, or 3 (Other values default to stopping criterion 1.)

rparm

is an array of parameters, RPARM(*i*), where:

`RPARM(1)` has the following meaning, where:

- if RPARM(1) > 0, then RPARM(1) is the relative accuracy ε used in the stopping criterion.
- if RPARM(1) = 0, then the solver is forced to evaluate at most IPARM(1) iterations.

See "5" on page 765.

RPARM(2), see On Return.

RPARM(3) has the following meaning, where:

- If IPARM(4) ≠ 3, then RPARM(3) is not used.
- If IPARM(4) = 3, then RPARM(3) is the acceleration parameter used in SSOR. (A value in the range 0.5 to 2.0 is suitable for most problems.)

Specified as: a one-dimensional array of (at least) length 3, containing long-precision real numbers, where:

RPARM(1) ≥ 0
If IPARM(4) = 3, RPARM(3) > 0

aux1
is working storage for this subroutine, where:

If *init* = 'I', the working storage is computed. It can contain any values.

If *init* = 'S', the working storage is used in solving the linear system. It contains the coefficient matrix and preconditioner in internal format, computed in an earlier call to this subroutine.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*.

Specified as: an integer, where:

In these formulas *nw* has the following value:

If *stor* = 'G', then *nw* = IA(*n*+1)-1+*n*.
If *stor* = 'U' or 'L', then *nw* = 2(IA(*n*+1)-1).

For 32-bit integer arguments

If IPARM(4) = 1, use *naux1* ≥ (3/2)*nw* + (1/2)*n* + 20.
If IPARM(4) = 2, use *naux1* ≥ (3/2)*nw* + (3/2)*n* + 20.
If IPARM(4) = 3, 4, or 5, use *naux1* ≥ 3*nw* + *n* + 20.

For 64-bit integer arguments

If IPARM(4) = 1, use *naux1* ≥ 2*nw* + *n* + 40.
If IPARM(4) = 2, use *naux1* ≥ 2*nw* + 2*n* + 40.
If IPARM(4) = 3, 4, or 5, use *naux1* ≥ 4*nw* + 4*n* + 40.

Note: If you receive an attention message, you have not specified sufficient auxiliary storage to achieve optimal performance, but it is enough to perform the computation. To obtain optimal performance, you need to use the amount given by the attention message.

aux2
has the following meaning:

If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is working storage used by this subroutine that is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers.

naux2
is the number of doublewords in the working storage specified in *aux2*.

Specified as: an integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, DSRIS dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise,

If `IPARM(2)` = 1, use *naux*2 ≥ 4*n*.
If `IPARM(2)` = 2, use *naux*2 ≥ 7*n*.
If `IPARM(2)` = 3, use *naux*2 ≥ (*k*+2)*n*+*k*(*k*+4)+1, where *k* = `IPARM(3)`.
If `IPARM(2)` = 4, use *naux*2 ≥ 7*n*.
If `IPARM(2)` = 5, use *naux*2 ≥ 9*n*.

On Return

ar

is the sparse matrix ***A*** of order *n*, stored by rows in an array, referred to as AR. The *stor* argument indicates the storage variation used for storing matrix ***A***. The order of the elements in each row of ***A*** in AR may be changed on output.

Returned as: a one-dimensional array, containing long-precision real numbers. The number of elements in this array can be determined by subtracting 1 from the value in `IA`(*n*+1).

ja

is the array, referred to as `JA`, containing the column numbers of each nonzero element in sparse matrix **A**. These elements correspond to the arrangement of the contents of AR on output.

Returned as: a one-dimensional array, containing integers; 1 ≤ (`JA` elements) ≤ *n*. The number of elements in this array can be determined by subtracting 1 from the value in `IA`(*n*+1).

x

is the vector ***x*** of length *n*, containing the solution of the system ***Ax*** = ***b***. Returned as: a one-dimensional array of (at least) length *n*, containing long-precision real numbers.

iparm

is an array of parameters, `IPARM`(*i*), where:

`IPARM(1)` through `IPARM(5)` are unchanged.

`IPARM(6)` contains the number of iterations performed by this subroutine.

Returned as: a one-dimensional array of length 6, containing integers.

rparm

is an array of parameters, `RPARM`(*i*), where:

`RPARM(1)` is unchanged.

`RPARM(2)` contains the estimate of the error of the solution. If the process converged, `RPARM(2)` ≤ ε.

`RPARM(3)` is unchanged.

Returned as: a one-dimensional array of length 3, containing long-precision real numbers.

aux1

is working storage for this subroutine, containing the coefficient matrix and preconditioner in internal format, ready to be passed in a subsequent invocation of this subroutine. Returned as: an area of storage, containing *naux1* long-precision real numbers.

Notes

1. If you want to solve the same sparse linear system of equations multiple times using a different algorithm with the same preconditioner and using a different right-hand side each time, you get the best performance by using the following technique. Call DSRIS the first time with *init* = 'I'. This solves the system, and then stores the coefficient matrix and preconditioner in internal format in *aux1*. On the subsequent invocations of DSRIS with different right-hand sides, specify *init* = 'S'. This indicates to

DSRIS to use the contents of *aux1*, saving the time to convert your coefficient matrix and preconditioner to internal format. If you use this technique, you should not modify the contents of *aux1* between calls to DSRIS.

In some cases, you can specify a different algorithm in `IPARM(2)` when making calls with *init* = 'S'. (See Example 2.) However, DSRIS sometimes needs different information in *aux1* for different algorithms. When this occurs, DSRIS issues an attention message, continues processing the computation, and then resets the contents of *aux1*. Your performance is not improved in this case, which is functionally equivalent to calling DSRIS with *init* = 'I'.

2. If you use the CG method with *init* = 'I', you must use the CG method when you specify *init* = 'S'. However, if you use a different method with *init* = 'I', you can use any other method, except CG, when you specify *init* = 'S'.
3. These subroutines accept lowercase letters for the *stor* and *init* arguments.
4. Matrix ***A***, vector ***x***, and vector ***b*** must have no common elements; otherwise, results are unpredictable.
5. The algorithm computes a sequence of approximate solution vectors ***x*** that converge to the solution. The iterative procedure is stopped when the selected stopping criterion is satisfied or when more than the maximum number of iterations (in `IPARM(1)`) is reached.

   For the stopping criteria specified in `IPARM(5)`, the relative accuracy ε (in `RPARM(1)`) must be specified reasonably (10^{-4} to 10^{-8}). If you specify a larger ε, the algorithm takes fewer iterations to converge to a solution. If you specify a smaller ε, the algorithm requires more iterations and computer time, but converges to a more precise solution. If the value you specify is unreasonably small, the algorithm may fail to converge within the number of iterations it is allowed to perform.
6. For a description of how sparse matrices are stored by rows, see "Storage-by-Rows" on page 115.
7. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

The linear system:

Ax = ***b***

is solved using one of the following methods: conjugate gradient (CG), conjugate gradient squared (CGS), generalized minimum residual (GMRES), more smoothly converging variant of the CGS method (Bi-CGSTAB), or transpose-free quasi-minimal residual method (TFQMR), where:

A is a sparse matrix of order *n*. The matrix is stored in arrays AR, IA, and JA. If it is general, it is stored by rows. If it is symmetric, it can be stored using upper- or lower-storage-by-rows.

x is a vector of length *n*.

b is a vector of length *n*.

One of the following preconditioners is used:

- an incomplete LU factorization
- an incomplete Cholesky factorization (for positive definite symmetric matrices)
- diagonal scaling
- symmetric successive over-relaxation (SSOR) with two possible choices for the diagonal matrix:
  - the absolute values sum of the input matrix
  - the diagonal obtained from the LU factorization

See references ["46" on page 1255], ["69" on page 1257], ["101" on page 1258], ["107" on page 1258], ["110" on page 1259], and ["116" on page 1259].

When you call this subroutine to solve a system for the first time, you specify *init* = 'I'. After that, you can solve the same system any number of times by calling this subroutine each time with *init* = 'S'. These subsequent calls use the coefficient matrix and preconditioner, stored in internal format in *aux1*. You optimize performance by doing this, because certain portions of the computation have already been performed.

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors
The following errors, with their corresponding return codes, can occur in this subroutine. For details on error handling, see "What Can You Do about ESSL Computational Errors?" on page 65.

- For error 2110, if `RPARM(1)` > , return code 1 indicates that the subroutine exceeded `IPARM(1)` iterations without converging. Vector ***x*** contains the approximate solution computed at the last iteration.
- For error 2130, return code 2 indicates that the incomplete LU factorization of ***A*** could not be completed, because one pivot was 0.
- For error 2124, the subroutine has been called with *init* = 'S', but the data contained in *aux1* was computed for a different algorithm. An attention message is issued. Processing continues, and the contents of *aux1* are reset correctly.
- For error 2134, return code 3 indicates that the data contained in *aux1* is not consistent with the input sparse matrix. The subroutine has been called with *init* = 'S', and *aux1* contains an incomplete factorization and internal data storage for the input matrix ***A*** that was computed by a previous call to the subroutine when *init* = 'I'. This error indicates that *aux1* has been modified since the last call to the subroutine, or that the input matrix is not the same as the one that was factored. If the default action has been overridden, the subroutine can be called again with the same parameters, with the exception of `IPARM(4)` = 1 or 4.
- For error 2131, return code 4 indicates that the matrix is singular, because all elements in one row of the matrix contain zero.
- For error 2129, return code 5 indicates that the matrix is not positive definite.
- For error 2128, return code 8 indicates an internal ESSL error. Please contact your IBM Representative.

Input-Argument Errors

1. $n < 0$
2. *stor* ≠ 'G', 'U', or 'L'
3. *init* ≠ 'I' or 'S'
4. `IA`(*n*+1) < 1
5. `IA`(*i*+1)-`IA`(*i*) < 0, for any *i* = 1, *n*
6. `IPARM(1)` < 0
7. `IPARM(2)` ≠ 1, 2, 3, 4, or 5
8. `IPARM(3)` ≤ 0 and `IPARM(2)` = 3
9. `IPARM(4)` ≠ 1, 2, 3, 4, or 5
10. `RPARM(1)` < 0
11. `RPARM(3)` ≤ 0 and `IPARM(4)` = 3
12. *naux1* is too small—that is, less than the minimum required value. Return code 6 is returned if error 2015 is recoverable.
13. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 7 is returned for *naux2* if error 2015 is recoverable.

Examples

Example 1

This example finds the solution of the linear system ***Ax*** = ***b*** for the sparse matrix ***A***, which is stored by rows in arrays AR, IA, and JA. The system is solved using the Bi-CGSTAB algorithm. The iteration is stopped when the norm of the residual is less than the given threshold specified in RPARM(1). The algorithm is allowed to perform 20 iterations. The process converges after 9 iterations. Matrix ***A*** is:

```
          | 2.0  0.0   0.0  0.0   0.0   0.0   0.0   0.0   0.0 |
          | 0.0  2.0  -1.0  0.0   0.0   0.0   0.0   0.0   0.0 |
          | 0.0  1.0   2.0  0.0   0.0   0.0   0.0   0.0   0.0 |
          | 1.0  0.0   0.0  2.0  -1.0   0.0   0.0   0.0   0.0 |
          | 0.0  0.0   0.0  1.0   2.0  -1.0   0.0   0.0   0.0 |
          | 0.0  0.0   0.0  0.0   1.0   2.0  -1.0   0.0   0.0 |
          | 0.0  0.0   0.0  0.0   0.0   1.0   2.0  -1.0   0.0 |
          | 0.0  0.0   0.0  0.0   0.0   0.0   1.0   2.0  -1.0 |
          | 0.0  0.0   0.0  0.0   0.0   0.0   0.0   1.0   2.0 |
```

Call Statement and Input:

```
            STOR INIT   N   AR   JA   IA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2
NAUX2
             |     |    |   |    |    |    |   |     |       |      |      |      |      |
CALL DSRIS( 'G' , 'I' , 9 , AR , JA , IA , B , X , IPARM , RPARM , AUX1 ,   98  , AUX2 ,
63 )
```

```
AR       =  (2.0, 2.0, -1.0, 1.0, 2.0, 1.0, 2.0, -1.0, 1.0, 2.0, -1.0,
             1.0, 2.0, -1.0, 1.0, 2.0, -1.0, 1.0, 2.0, -1.0, 1.0, 2.0)
JA       =  (1, 2, 3, 2, 3, 1, 4, 5, 4, 5, 6, 5, 6, 7, 6, 7, 8, 7, 8,
             9, 8, 9)
IA       =  (1, 2, 4, 6, 9, 12, 15, 18, 21, 23)
B        =  (2.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 3.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
IPARM(1) =  20
IPARM(2) =  4
IPARM(3) =  0
IPARM(4) =  1
IPARM(5) =  10
RPARM(1) =  1.D-7
RPARM(3) =  1.0
```

Output:

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(6) =  9
RPARM(2) =  0.29D-16
```

Example 2

This example finds the solution of the linear system ***Ax*** = ***b*** for the same sparse matrix ***A*** used in Example 1. It also uses the same right-hand side in ***b*** and the same initial guesses in ***x***. However, the system is solved using a different algorithm, conjugate gradient squared (CGS). Because INIT is 'S', the best performance is achieved. The iteration is stopped when the norm of the residual is less than the given threshold specified in RPARM(1). The algorithm is allowed to perform 20 iterations. The process converges after 9 iterations.

Call Statement and Input:

```
            STOR INIT   N   AR   JA   IA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2
NAUX2
             |     |    |   |    |    |    |   |     |       |      |      |      |      |
CALL DSRIS( 'G' , 'S' , 9 , AR , JA , IA , B , X , IPARM , RPARM , AUX1 ,   98  , AUX2 ,
63 )
```

AR =(same as input AR in Example 1)
JA =(same as input JA in Example 1)
IA =(same as input IA in Example 1)
B =(same as input B in Example 1)

X =(same as input X in Example 1)
IPARM(1) = 20
IPARM(2) = 2
IPARM(3) = 0
IPARM(4) = 1
IPARM(5) = 10
RPARM(1) = 1.D-7
RPARM(3) = 1.0

Output:

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(6) =  9
RPARM(2) =  0.42D-19
```

Example 3

This example finds the solution of the linear system ***Ax*** = ***b*** for the sparse matrix ***A***, which is stored by rows in arrays AR, IA, and JA. The system is solved using the two-term conjugate gradient method (CG), preconditioned by incomplete LU factorization. The iteration is stopped when the norm of the residual is less than the given threshold specified in RPARM(1). The algorithm is allowed to perform 20 iterations. The process converges after 1 iteration. Matrix ***A*** is:

```
          |  2.0   0.0  -1.0   0.0   0.0   0.0   0.0   0.0   0.0 |
          |  0.0   2.0   0.0  -1.0   0.0   0.0   0.0   0.0   0.0 |
          | -1.0   0.0   2.0   0.0  -1.0   0.0   0.0   0.0   0.0 |
          |  0.0  -1.0   0.0   2.0   0.0  -1.0   0.0   0.0   0.0 |
          |  0.0   0.0  -1.0   0.0   2.0   0.0  -1.0   0.0   0.0 |
          |  0.0   0.0   0.0  -1.0   0.0   2.0   0.0  -1.0   0.0 |
          |  0.0   0.0   0.0   0.0  -1.0   0.0   2.0   0.0  -1.0 |
          |  0.0   0.0   0.0   0.0   0.0  -1.0   0.0   2.0   0.0 |
          |  0.0   0.0   0.0   0.0   0.0   0.0  -1.0   0.0   2.0 |
```

Call Statement Input:

```
            STOR INIT   N   AR   JA   IA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2
NAUX2
             |    |    |   |    |    |    |   |     |       |       |       |       |       |
CALL DSRIS( 'G' , 'I' , 9 , AR , JA , IA , B , X , IPARM , RPARM , AUX1 ,  223  , AUX2 ,
36 )
```

```
AR       =  (2.0, -1.0, 2.0, -1.0, -1.0, 2.0, -1.0, -1.0, 2.0, -1.0,
             -1.0, 2.0, -1.0, -1.0, 2.0, -1.0, -1.0, 2.0, -1.0, -1.0,
             2.0, -1.0, 2.0)
JA       =  (1, 3, 2, 4, 1, 3, 5, 2, 4, 6, 3, 5, 7, 4, 6, 8, 5, 7, 9,
             6, 8, 7, 9)
IA       =  (1, 3, 5, 8, 11, 14, 17, 20, 22, 24)
B        =  (1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
IPARM(1) =  20
IPARM(2) =  1
IPARM(3) =  0
IPARM(4) =  4
IPARM(5) =  1
RPARM(1) =  1.D-7
RPARM(3) =  1.0
```

Output:

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(6) =  1
RPARM(2) =  0.16D-15
```

Example 4

This example finds the solution of the linear system ***Ax*** = ***b*** for the same sparse matrix ***A*** used in Example 3. However, matrix ***A*** is stored using upper-storage-by-rows in arrays AR, IA, and JA. The system is solved using the generalized minimum residual (GMRES), restarted after 5 steps and preconditioned with SSOR splitting. The iteration is stopped when the norm of the residual is less than the given threshold specified in RPARM(1). The algorithm is allowed to perform 20 iterations. The process converges after 12 iterations.

Call Statement Input

```
            STOR INIT   N   AR   JA   IA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2   
NAUX2
             |     |    |   |    |    |    |   |     |       |       |      |       |       |
CALL DSRIS( 'U' , 'I' , 9 , AR , JA , IA , B , X , IPARM , RPARM , AUX1 ,  219  , AUX2 ,
109 )
```

```
AR       =  (2.0, -1.0, 2.0, -1.0, 2.0, -1.0, 2.0, -1.0, 2.0, -1.0,
             2.0, -1.0, 2.0, -1.0, 2.0, 2.0)
JA       =  (1, 3, 2, 4, 3, 5, 4, 6, 5, 7, 6, 8, 7, 9, 8, 9)
IA       =  (1, 3, 5, 7, 9, 11, 13, 15, 16, 17)
B        =  (1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
IPARM(1) =  20
IPARM(2) =  3
IPARM(3) =  5
IPARM(4) =  3
IPARM(5) =  1
RPARM(1) =  1.D-7
RPARM(3) =  2.0
```

Output:

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(6) =  12
RPARM(2) =  0.33D-7
```

DSMCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Matrix Storage Mode)

Purpose

This subroutine solves a symmetric, positive definite or negative definite linear system, using the conjugate gradient method, with or without preconditioning by an incomplete Cholesky factorization, for a sparse matrix stored in compressed-matrix storage mode. Matrix ***A*** and vectors ***x*** and ***b*** are used:

Ax = ***b***

where ***A***, ***x***, and ***b*** contain long-precision real numbers.

Note:

1. These subroutines are provided only for migration purposes. You get better performance and a wider choice of algorithms if you use the DSRIS subroutine.
2. If your sparse matrix is stored by rows, as defined in "Storage-by-Rows" on page 115, you should first use the utility subroutine DSRSM to convert your sparse matrix to compressed-matrix storage mode. See "DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)" on page 1212

Syntax

Language	Syntax
Fortran	CALL DSMCG (*m*, *nz*, *ac*, *ka*, *lda*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	dsmcg (*m*, *nz*, *ac*, *ka*, *lda*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

m

is the order of the linear system ***Ax*** = ***b*** and the number of rows in sparse matrix ***A***.

Specified as: an integer; $m \geq 0$.

nz

is the maximum number of nonzero elements in each row of sparse matrix ***A***.

Specified as: an integer; $nz \geq 0$.

ac

is the array, referred to as AC, containing the values of the nonzero elements of the sparse matrix, stored in compressed-matrix storage mode.

Specified as: an *lda* by (at least) *nz* array, containing long-precision real numbers.

ka

is the array, referred to as KA, containing the column numbers of the matrix ***A*** elements stored in the corresponding positions in array AC.

Specified as: an *lda* by (at least) *nz* array, containing integers, where $1 \leq$ (elements of KA) $\leq m$.

lda

is the leading dimension of the arrays specified for *ac* and *ka*.

Specified as: an integer; $lda > 0$ and $lda \geq m$.

b

is the vector ***b*** of length *m*, containing the right-hand side of the matrix problem.

Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

x

is the vector ***x*** of length *m*, containing your initial guess of the solution of the linear system.

Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers. The elements can have any value, and if no guess is available, the value can be zero.

iparm

is an array of parameters, `IPARM(`*i*`)`, where:

- `IPARM(1)` controls the number of iterations.

  If `IPARM(1)` > 0, `IPARM(1)` is the maximum number of iterations allowed.

  If `IPARM(1)` = 0, the following default values are used:

  `IPARM(1)` = 300
  `IPARM(2)` = 1
  `IPARM(3)` = 0
  `RPARM(1)` = 10^{-6}

- `IPARM(2)` is the flag used to select the stopping criterion.

  If `IPARM(2)` = 0, the conjugate gradient iterative procedure is stopped when:

  $\|\boldsymbol{r}\|_2 / \|\boldsymbol{x}\|_2 < \varepsilon$

  where $\boldsymbol{r} = \boldsymbol{b}\text{-}\boldsymbol{Ax}$ is the residual, and ε is the desired relative accuracy. ε is stored in `RPARM(1)`.

  If `IPARM(2)` = 1, the conjugate gradient iterative procedure is stopped when:

  $\|\boldsymbol{r}\|_2 / \lambda\|\boldsymbol{x}\|_2 < \varepsilon$

  where λ is an estimate to the minimum eigenvalue of the iteration matrix. λ is computed adaptively by this program and, on output, is stored in `RPARM(2)`.

  If `IPARM(2)` = 2, the conjugate gradient iterative procedure is stopped when:

  $\|\boldsymbol{r}\|_2 / \lambda\|\boldsymbol{x}\|_2 < \varepsilon$

where λ is a predetermined estimate to the minimum eigenvalue of the iteration matrix. This eigenvalue estimate, on input, is stored in RPARM(2) and may be obtained by an earlier call to this subroutine with the same matrix.

- IPARM(3) is the flag that determines whether the system is to be solved using the conjugate gradient method, preconditioned by an incomplete Cholesky factorization with no fill-in.

  If IPARM(3) = 0, the system is not preconditioned.

  If IPARM(3) = 10, the system is preconditioned by an incomplete Cholesky factorization.

  If IPARM(3) = -10, the system is preconditioned by an incomplete Cholesky factorization, where the factorization matrix was computed in an earlier call to this subroutine and is stored in *aux2*.
- IPARM(4), see On Return.

Specified as: an array of (at least) length 4, containing integers, where:

IPARM(1) ≥ 0
IPARM(2) = 0, 1, or 2
IPARM(3) = 0, 10, or -10

rparm
is an array of parameters, RPARM(*i*), where ε is stored in RPARM(1), and λ is stored in RPARM(2).

RPARM(1) > 0, is the relative accuracy ε used in the stopping criterion.

RPARM(2) > 0, is the estimate of the smallest eigenvalue, λ, of the iteration matrix. It is only used when IPARM(2) = 2.

RPARM(3), see On Return.

Specified as: a one-dimensional array of (at least) length 3, containing long-precision real numbers.

aux1
has the following meaning:

If *naux1* = 0 and error 2015 is unrecoverable, *aux1* is ignored.

Otherwise, it is a storage work area used by this subroutine, which is available for use by the calling program between calls to this subroutine. Its size is specified by *naux1*.

Specified as: an area of storage, containing long-precision real numbers.

naux1
is the size of the work area specified by *aux1*—that is, the number of elements in *aux1*.

Specified as: an integer, where:

If *naux1* = 0 and error 2015 is unrecoverable, DSMCG dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux1* must have at least the following value, where:

If IPARM(2) = 0 or 2, use *naux1* ≥ 3*m*.

If IPARM(2) = 1 and IPARM(1) ≠ 0, use *naux1* ≥ 3*m*+2(IPARM(1)).

If IPARM(2) = 1 and IPARM(1) = 0, use *naux1* ≥ 3*m*+600.

aux2
is a storage work area used by this subroutine. If IPARM(3) = -10, *aux2* must contain the incomplete Cholesky factorization of matrix ***A***, computed in an earlier call to DSMCG. The size of *aux2* is specified by *naux2*.

Specified as: an area of storage, containing long-precision real numbers.

naux2
is the size of the work area specified by *aux2*—that is, the number of elements in *aux2*.

Specified as: an integer. When IPARM(3) = 10 or -10, *naux2* must have at least the following value:

For 32-bit integer arguments
naux2 ≥ *m*(*nz*-1)1.5+2(*m*+6).

For 64-bit integer arguments
naux2 ≥ *m*(*nz*-1)2.0+3(*m*+6).

On Return

x
is the vector ***x*** of length *m*, containing the solution of the system ***Ax*** = ***b***. Returned as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

iparm
is an array of parameters, `IPARM`(*i*), where:

`IPARM(1)` is unchanged.

`IPARM(2)` is unchanged.

`IPARM(3)` is unchanged.

`IPARM(4)` contains the number of iterations performed by this subroutine.

Returned as: a one-dimensional array of length 4, containing integers.

rparm
is an array of parameters, `RPARM`(*i*), where:

`RPARM(1)` is unchanged.

`RPARM(2)` is unchanged if `IPARM(2)` = 0 or 2. If `IPARM(2)` = 1, `RPARM(2)` contains λ, an estimate of the smallest eigenvalue of the iteration matrix.

`RPARM(3)` contains the estimate of the error of the solution. If the process converged, `RPARM(3)` ≤ ε.

Returned as: a one-dimensional array of length 3, containing long-precision real numbers; λ > 0.

aux2
is the storage work area used by this subroutine.

If `IPARM(3)` = 10, *aux2* contains the incomplete Cholesky factorization of matrix ***A***.

If `IPARM(3)` = -10, *aux2* is unchanged.

See "Notes" on page 772 for additional information on *aux2*. Returned as: an area of storage, containing long-precision real numbers.

Notes

1. When `IPARM(3)` = -10, this subroutine uses the incomplete Cholesky factorization in *aux2*, computed in an earlier call to this subroutine. When `IPARM(3)` = 10, this subroutine computes the incomplete Cholesky factorization and stores it in *aux2*.
2. If you solve the same sparse linear system of equations several times with different right-hand sides using the preconditioned algorithm, specify `IPARM(3)` = 10 on the first invocation. The incomplete factorization is stored in *aux2*. You may save computing time on subsequent calls by setting `IPARM(3)` = -10. In this way, the algorithm reutilizes the incomplete factorization that was computed the first time. Therefore, you should not modify the contents of *aux2* between calls.
3. Matrix ***A*** must have no common elements with vectors ***x*** and ***b***; otherwise, results are unpredictable.
4. In the iterative solvers for sparse matrices, the relative accuracy ε (`RPARM(1)`) must be specified "reasonably" (10^{-4} to 10^{-8}). The algorithm computes a sequence of approximate solution vectors ***x*** that converge to the solution. The iterative procedure is stopped when the norm of the residual is sufficiently small—that is, when:

   $\|\boldsymbol{b}-\boldsymbol{Ax}\|_2 / \lambda\|\boldsymbol{x}\|_2 < \varepsilon$

where λ is an estimate of the minimum eigenvalue of the iteration matrix, which is either estimated adaptively or given by the user. As a result, if you specify a larger ε, the algorithm takes fewer iterations to converge to a solution. If you specify a smaller ε, the algorithm requires more iterations and computer time, but converges to a more precise solution. If the value you specify is unreasonably small, the algorithm may fail to converge within the number of iterations it is allowed to perform.

5. For a description of how sparse matrices are stored in compressed-matrix storage mode, see "Compressed-Matrix Storage Mode" on page 110.
6. On output, array AC and vector ***b*** are not bitwise identical to what they were on input, because the matrix ***A*** and the right-hand side are scaled before starting the iterative process and are unscaled before returning control to the user. In addition, arrays AC and KA may be rearranged on output, but still contain a mathematically equivalent mapping of the elements in matrix ***A***.
7. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

The sparse positive definite or negative definite linear system:

Ax = ***b***

is solved, where:

A is a symmetric, positive definite or negative definite sparse matrix of order *m*, stored in compressed-matrix storage mode in AC and KA.

x is a vector of length *m*.

b is a vector of length *m*.

The system is solved using the two-term conjugate gradient method, with or without preconditioning by an incomplete Cholesky factorization. In both cases, the matrix is scaled by the square root of the diagonal.

See references ["75" on page 1257] and ["82" on page 1257]. ["46" on page 1255].

If your program uses a sparse matrix stored by rows and you want to use this subroutine, first convert your sparse matrix to compressed-matrix storage mode by using the subroutine DSRSM (see "DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)" on page 1212).

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux1* = 0, and unable to allocate work area.

Computational Errors
The following errors, with their corresponding return codes, can occur in this subroutine. Where a value of *i* is indicated, it can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for that particular error code in the ESSL error option table; otherwise, the default value causes your program to terminate when the error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

- For error 2110, return code 1 indicates that the subroutine exceeded `IPARM(1)` iterations without converging. Vector ***x*** contains the approximate solution computed at the last iteration.
- For error 2111, return code 2 indicates that *aux2* contains an incorrect factorization. The subroutine has been called with `IPARM(3)` = -10, and *aux2* contains an incomplete factorization of the input matrix ***A*** that was computed by a previous call to the subroutine when `IPARM(3)` = 10. This error indicates that *aux2* has been modified since the last call to the subroutine, or that the input matrix is not the same as the one that was factored. If the default action has been overridden, the subroutine can be called again with the same parameters, with the exception of `IPARM(3)` = 0 or 10.

- For error 2109, return code 3 indicates that the inner product (***y***,***Ay***) is negative in the iterative procedure after iteration *i*. This should not occur, because the input matrix is assumed to be positive or negative definite. Vector ***x*** contains the results of the last iteration. The value *i* is identified in the computational error message.
- For error 2108, return code 4 indicates that the matrix is not positive definite. AC is partially modified and does not represent the same matrix as on entry.

Input-Argument Errors

1. *m* < 0
2. *lda* < 1
3. *lda* < *m*
4. *nz* < 0
5. *nz* = 0 and *m* > 0
6. `IPARM(1)` < 0
7. `IPARM(2)` ≠ 0, 1, or 2
8. `IPARM(3)` ≠ 0, 10, or -10
9. `RPARM(1)` < 0
10. `RPARM(2)` < 0
11. Error 2015 is recoverable or *naux1*≠0, and *naux1* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.
12. *naux2* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.

Examples

Example 1

This example finds the solution of the linear system ***Ax*** = ***b*** for the sparse matrix ***A***, which is stored in compressed-matrix storage mode in arrays AC and KA. The system is solved using the conjugate gradient method. Matrix ***A*** is:

```
          |  2.0   0.0   0.0  -1.0   0.0   0.0   0.0   0.0   0.0 |
          |  0.0   2.0  -1.0   0.0   0.0   0.0   0.0   0.0   0.0 |
          |  0.0  -1.0   2.0   0.0   0.0   0.0   0.0   0.0   0.0 |
          | -1.0   0.0   0.0   2.0  -1.0   0.0   0.0   0.0   0.0 |
          |  0.0   0.0   0.0  -1.0   2.0  -1.0   0.0   0.0   0.0 |
          |  0.0   0.0   0.0   0.0  -1.0   2.0  -1.0   0.0   0.0 |
          |  0.0   0.0   0.0   0.0   0.0  -1.0   2.0  -1.0   0.0 |
          |  0.0   0.0   0.0   0.0   0.0   0.0  -1.0   2.0  -1.0 |
          |  0.0   0.0   0.0   0.0   0.0   0.0   0.0  -1.0   2.0 |
```

Note: For input matrix KA, (.) indicates any value between 1 and 9.

Call Statement and Input:

```
            M   NZ  AC   KA LDA   B   X  IPARM  RPARM  AUX1 NAUX1  AUX2 NAUX2
            |   |   |    |   |    |   |    |      |     |     |     |     |
CALL DSMCG( 9 , 3 , AC, KA, 9 , B , X, IPARM, RPARM, AUX1, 27  , AUX2,  0  )
```

```
IPARM(1) =  20
IPARM(2) =  0
IPARM(3) =  0
RPARM(1) =  1.D-7
```

```
         |  2.0  -1.0   0.0 |
         |  2.0  -1.0   0.0 |
         | -1.0   2.0   0.0 |
         | -1.0   2.0  -1.0 |
AC   =  | -1.0   2.0  -1.0 |
        | -1.0   2.0  -1.0 |
        | -1.0   2.0  -1.0 |
        | -1.0   2.0  -1.0 |
        | -1.0   2.0   0.0 |
```

```
        | 1  4  . |
        | 2  3  . |
        | 2  3  . |
        | 1  4  5 |
KA   =  | 4  5  6 |
        | 5  6  7 |
        | 6  7  8 |
        | 7  8  9 |
        | 8  9  . |
```

```
B        =  (1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

Output:

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  5
RPARM(2) =  0
RPARM(3) =  0.351D-15
```

Example 2

This example finds the solution of the linear system ***Ax*** = ***b*** for the same sparse matrix ***A*** as in Example 1, which is stored in compressed-matrix storage mode in arrays AC and KA. The system is solved using the conjugate gradient method, preconditioned with an incomplete Cholesky factorization. The smallest eigenvalue of the iteration matrix is computed and used in stopping the computation.

Note: For input matrix KA, (.) indicates any value between 1 and 9.

Call Statement and Input:

```
           M   NZ  AC  KA LDA  B   X  IPARM  RPARM  AUX1 NAUX1  AUX2 NAUX2
           |   |   |   |   |   |   |    |      |      |     |     |     |
CALL DSMCG( 9 , 3 , AC, KA, 9 , B , X, IPARM, RPARM, AUX1, 67  , AUX2, 74  )
```

`IPARM(1) = 20`
`IPARM(2) = 1`
`IPARM(3) = 10`
`RPARM(1) = 1.D-7`
AC =(same as input AC in Example 1)
KA =(same as input KA in Example 1)
`B = (1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)`
`X = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)`

Output:

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  1
RPARM(2) =  1
RPARM(3) =  0.100D-15
```

DSDCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)

Purpose

This subroutine solves a symmetric, positive definite or negative definite linear system, using the two-term conjugate gradient method, with or without preconditioning by an incomplete Cholesky factorization, for a sparse matrix stored in compressed-diagonal storage mode. Matrix ***A*** and vectors ***x*** and ***b*** are used:

Ax = ***b***

where ***A***, ***x***, and ***b*** contain long-precision real numbers.

Syntax

Language	Syntax
Fortran	CALL DSDCG (*iopt*, *m*, *nd*, *ad*, *lda*, *la*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	dsdcg (*iopt*, *m*, *nd*, *ad*, *lda*, *la*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

iopt

indicates the type of storage used, where:

If *iopt* = 0, all the nonzero diagonals of the sparse matrix are stored in compressed-diagonal storage mode.

If *iopt* = 1, the sparse matrix, stored in compressed-diagonal storage mode, is symmetric. Only the main diagonal and one of each pair of identical diagonals are stored in array AD.

Specified as: an integer; *iopt* = 0 or 1.

m

is the order of the linear system ***Ax*** = ***b*** and the number of rows in sparse matrix ***A***.

Specified as: an integer; *m* ≥ 0.

nd

is the number of nonzero diagonals stored in the columns of array AD, the number of columns in the array AD, and the number of elements in array LA.

Specified as: an integer; it must have the following value, where:

If *m* > 0, then *nd* > 0.

If *m* = 0, then *nd* ≥ 0.

ad

is the array, referred to as AD, containing the values of the nonzero elements of the sparse matrix stored in compressed-diagonal storage mode. If *iopt* = 1, the main diagonal and one of each pair of identical diagonals is stored in this array.

Specified as: an *lda* by (at least) *nd* array, containing long-precision real numbers.

lda

is the leading dimension of the array specified for *ad*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *m*.

la

is the array, referred to as LA, containing the diagonal numbers *k* for the diagonals stored in each corresponding column in array AD. For an explanation of how diagonal numbers are assigned, see "Compressed-Diagonal Storage Mode" on page 111.

Specified as: a one-dimensional array of (at least) length *nd*, containing integers, where 1-*m* ≤ (elements of LA) ≤ *m*-1.

b

is the vector ***b*** of length *m*, containing the right-hand side of the matrix problem.

Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

x

is the vector ***x*** of length *m*, containing your initial guess of the solution of the linear system.

Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers. The elements can have any value, and if no guess is available, the value can be zero.

iparm

is an array of parameters, `IPARM(`*i*`)`, where:

- `IPARM(1)` controls the number of iterations.

  If `IPARM(1)` > 0, `IPARM(1)` is the maximum number of iterations allowed.

  If `IPARM(1)` = 0, the following default values are used:

  `IPARM(1)` = 300
  `IPARM(2)` = 1
  `IPARM(3)` = 0
  `RPARM(1)` = 10^{-6}

- `IPARM(2)` is the flag used to select the stopping criterion.

  If `IPARM(2)` = 0, the conjugate gradient iterative procedure is stopped when:

  $\|\boldsymbol{r}\|_2 / \|\boldsymbol{x}\|_2 < \varepsilon$

  where $\boldsymbol{r} = \boldsymbol{b}\text{-}\boldsymbol{Ax}$ is the residual and ε is the desired relative accuracy. ε is stored in `RPARM(1)`.

  If `IPARM(2)` = 1, the conjugate gradient iterative procedure is stopped when:

  $\|\boldsymbol{r}\|_2 / \lambda\|\boldsymbol{x}\|_2 < \varepsilon$

  where λ is an estimate to the minimum eigenvalue of the iteration matrix. λ is computed adaptively by this program and, on output, is stored in `RPARM(2)`.

  If `IPARM(2)` = 2, the conjugate gradient iterative procedure is stopped when:

  $\|\boldsymbol{r}\|_2 / \lambda\|\boldsymbol{x}\|_2 < \varepsilon$

  where λ is a predetermined estimate to the minimum eigenvalue of the iteration matrix. This eigenvalue estimate, on input, is stored in `RPARM(2)` and may be obtained by an earlier call to this subroutine with the same matrix.

- `IPARM(3)` is the flag that determines whether the system is to be solved using the conjugate gradient method, preconditioned by an incomplete Cholesky factorization with no fill-in.

  If `IPARM(3)` = 0, the system is not preconditioned.

  If `IPARM(3)` = 10, the system is preconditioned by an incomplete Cholesky factorization.

  If `IPARM(3)` = -10, the system is preconditioned by an incomplete Cholesky factorization, where the factorization matrix was computed in an earlier call to this subroutine and is stored in *aux2*.

- `IPARM(4)`, see On Return.

Specified as: an array of (at least) length 4, containing integers, where:

`IPARM(1)` = 0
`IPARM(2)` = 0, 1, or 2
`IPARM(3)` = 0, 10, or -10

rparm
is an array of parameters, `RPARM(`*i*`)`, where ε is stored in `RPARM(1)`, and λ is stored in `RPARM(2)`.

`RPARM(1)` > 0, is the relative accuracy ε used in the stopping criterion.

`RPARM(2)` > 0, is the estimate of the smallest eigenvalue, λ, of the iteration matrix. It is only used when `IPARM(2)` = 2.

`RPARM(3)`, see On Return.

Specified as: a one-dimensional array of (at least) length 3, containing long-precision real numbers.

aux1
has the following meaning:

If *naux1* = 0 and error 2015 is unrecoverable, *aux1* is ignored.

Otherwise, it is a storage work area used by this subroutine, which is available for use by the calling program between calls to this subroutine. Its size is specified by *naux1*.

Specified as: an area of storage, containing long-precision real numbers.

naux1
is the size of the work area specified by *aux1*—that is, the number of elements in *aux1*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, DSDCG dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, it must have at least the following value, where:

If `IPARM(2)` = 0 or 2, use *naux1* ≥ 3*m*.

If `IPARM(2)` = 1 and `IPARM(1)` ≠ 0, use *naux1* ≥ 3*m*+2(`IPARM(1)`).

If `IPARM(2)` = 1 and `IPARM(1)` = 0, use *naux1* ≥ 3*m*+600.

aux2
is the storage work area used by this subroutine. If `IPARM(3)` = -10, *aux2* must contain the incomplete Cholesky factorization of matrix ***A***, computed in an earlier call to DSDCG. Its size is specified by *naux2*.

Specified as: an area of storage, containing long-precision real numbers.

naux2
is the size of the work area specified by *aux2*—that is, the number of elements in *aux2*.

Specified as: an integer. When `IPARM(3)` = 10 or -10, *naux2* must have at least the following value:

For 32-bit integer arguments
$naux2 \geq m(3nd + 2) + 8$

For 64-bit integer arguments
$naux2 \geq m(4nd + 3) + 12$

On Return

x
is the vector ***x*** of length *m*, containing the solution of the system ***Ax*** = ***b***. Returned as: a one-dimensional array, containing long-precision real numbers.

iparm
As an array of parameters, `IPARM(`*i*`)`, where:

`IPARM(1)` is unchanged.

`IPARM(2)` is unchanged.

`IPARM(3)` is unchanged.

`IPARM(4)` contains the number of iterations performed by this subroutine.

Returned as: a one-dimensional array of length 4, containing integers.

rparm

is an array of parameters, `RPARM`(*i*), where:

`RPARM(1)` is unchanged.

`RPARM(2)` is unchanged if `IPARM(2)` = 0 or 2. If `IPARM(2)` = 1, `RPARM(2)` contains λ, an estimate of the smallest eigenvalue of the iteration matrix.

`RPARM(3)` contains the estimate of the error of the solution. If the process converged, `RPARM(3)` ≤ ε.

Returned as: a one-dimensional array of length 3, containing long-precision real numbers; λ > 0.

aux2

is the storage work area used by this subroutine.

If `IPARM(3)` = 10, *aux2* contains the incomplete Cholesky factorization of matrix ***A***.

If `IPARM(3)` = -10, *aux2* is unchanged.

See "Notes" on page 779 for additional information on *aux2*. Returned as: an area of storage, containing long-precision real numbers.

Notes

1. When `IPARM(3)` = -10, this subroutine uses the incomplete Cholesky factorization in *aux2*, computed in an earlier call to this subroutine. When `IPARM(3)` = 10, this subroutine computes the incomplete Cholesky factorization and stores it in *aux2*.
2. If you solve the same sparse linear system of equations several times with different right-hand sides using the preconditioned algorithm, specify `IPARM(3)` = 10 on the first invocation. The incomplete factorization is stored in *aux2*. You may save computing time on subsequent calls by setting `IPARM(3)` = -10. In this way, the algorithm reutilizes the incomplete factorization that was computed the first time. Therefore, you should not modify the contents of *aux2* between calls.
3. Matrix ***A*** must have no common elements with vectors ***x*** and ***b***; otherwise, results are unpredictable.
4. In the iterative solvers for sparse matrices, the relative accuracy ε (`RPARM(1)`) must be specified "reasonably" (10^{-4} to 10^{-8}). The algorithm computes a sequence of approximate solution vectors ***x*** that converge to the solution. The iterative procedure is stopped when the norm of the residual is sufficiently small—that is, when:

   $\|\boldsymbol{b}-\boldsymbol{Ax}\|_2 / \lambda\|\boldsymbol{x}\|_2 < \varepsilon$

   where λ is an estimate of the minimum eigenvalue of the iteration matrix, which is either estimated adaptively or given by the user. As a result, if you specify a larger ε, the algorithm takes fewer iterations to converge to a solution. If you specify a smaller ε, the algorithm requires more iterations and computer time, but converges to a more precise solution. If the value you specify is unreasonably small, the algorithm may fail to converge within the number of iterations it is allowed to perform.
5. For a description of how sparse matrices are stored in compressed-matrix storage mode, see "Compressed-Matrix Storage Mode" on page 110.
6. On output, array AD and vector ***b*** are not bitwise identical to what they were on input, because the matrix ***A*** and the right-hand side are scaled before starting the iterative process and are unscaled before returning control to the user. In addition, arrays AD and LA may be rearranged on output, but still contain a mathematically equivalent mapping of the elements in matrix ***A***.
7. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

The sparse positive definite or negative definite linear system:

Ax = ***b***

is solved, where:

A is a symmetric, positive definite or negative definite sparse matrix of order *m*, stored in compressed-diagonal storage mode in arrays AD and LA.

x is a vector of length *m*.

b is a vector of length *m*.

The system is solved using the two-term conjugate gradient method, with or without preconditioning by an incomplete Cholesky factorization. In both cases, the matrix is scaled by the square root of the diagonal.

See references ["75" on page 1257] and ["82" on page 1257]. ["46" on page 1255].

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux1* = 0, and unable to allocate work area.

Computational Errors
The following errors, with their corresponding return codes, can occur in this subroutine. Where a value of *i* is indicated, it can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for that particular error code in the ESSL error option table; otherwise, the default value causes your program to terminate when the error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

- For error 2110, return code 1 indicates that the subroutine exceeded `IPARM(1)` iterations without converging. Vector ***x*** contains the approximate solution computed at the last iteration.
- For error 2111, return code 2 indicates that *aux2* contains an incorrect factorization. The subroutine has been called with `IPARM(3)` = -10, and *aux2* contains an incomplete factorization of the input matrix ***A*** that was computed by a previous call to the subroutine when `IPARM(3)` = 10. This error indicates that *aux2* has been modified since the last call to the subroutine, or that the input matrix is not the same as the one that was factored. If the default action has been overridden, the subroutine can be called again with the same parameters, with the exception of `IPARM(3)` = 0 or 10.
- For error 2109, return code 3 indicates that the inner product (***y***,***Ay***) is negative in the iterative procedure after iteration *i*. This should not occur, because the input matrix is assumed to be positive or negative definite. Vector ***x*** contains the results of the last iteration. The value *i* is identified in the computational error message.
- For error 2108, return code 4 indicates that the matrix is not positive definite. AC is partially modified and does not represent the same matrix as on entry.

Input-Argument Errors

1. *iopt* ≠ 0 or 1
2. *m* < 0
3. *lda* < 1
4. *lda* < *m*
5. *nd* < 0
6. *nd* = 0 and *m* > 0
7. |λ(*i*)| > *m*-1 for *i* = 1, *nd*
8. `IPARM(1)` < 0
9. `IPARM(2)` ≠ 0, 1, or 2
10. `IPARM(3)` ≠ 0, 10, or -10
11. `RPARM(1)` < 0

12. RPARM(2) < 0
13. Error 2015 is recoverable or *naux1*≠0, and *naux1* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.
14. *naux2* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.

Examples

Example 1

This example finds the solution of the linear system ***Ax*** = ***b*** for sparse matrix ***A***, which is stored in compressed-diagonal storage mode in arrays AD and LA. The system is solved using the two-term conjugate gradient method. In this example, IOPT = 0.. Matrix ***A*** is:

```
          |  2.0   0.0  -1.0   0.0   0.0   0.0   0.0   0.0   0.0 |
          |  0.0   2.0   0.0  -1.0   0.0   0.0   0.0   0.0   0.0 |
          | -1.0   0.0   2.0   0.0  -1.0   0.0   0.0   0.0   0.0 |
          |  0.0  -1.0   0.0   2.0   0.0  -1.0   0.0   0.0   0.0 |
          |  0.0   0.0  -1.0   0.0   2.0   0.0  -1.0   0.0   0.0 |
          |  0.0   0.0   0.0  -1.0   0.0   2.0   0.0  -1.0   0.0 |
          |  0.0   0.0   0.0   0.0  -1.0   0.0   2.0   0.0  -1.0 |
          |  0.0   0.0   0.0   0.0   0.0  -1.0   0.0   2.0   0.0 |
          |  0.0   0.0   0.0   0.0   0.0   0.0  -1.0   0.0   2.0 |
```

Call Statement and Input:

```
            IOPT  M   ND  AD  LDA  LA   B   X   IPARM   RPARM   AUX1  NAUX1  AUX2  NAUX2
             |    |   |   |    |   |    |   |     |       |       |     |     |      |
CALL DSDCG( 0  , 9 , 3 , AD , 9 , LA , B , X , IPARM , RPARM , AUX1 , 283 , AUX2 ,  0  )
```

```
IPARM(1) =  20
IPARM(2) =  0
IPARM(3) =  0
RPARM(1) =  1.D-7
```

```
        | 2.0   0.0  -1.0 |
        | 2.0   0.0  -1.0 |
        | 2.0  -1.0  -1.0 |
        | 2.0  -1.0  -1.0 |
AD   =  | 2.0  -1.0  -1.0 |
        | 2.0  -1.0  -1.0 |
        | 2.0  -1.0  -1.0 |
        | 2.0  -1.0   0.0 |
        | 2.0  -1.0   0.0 |
```

```
LA       =  (0, -2, 2)
B        =  (1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

Output:

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  5
RPARM(2) =  0
RPARM(3) =  0.46D-16
```

Example 2

This example finds the solution of the linear system ***Ax*** = ***b*** for the same sparse matrix ***A*** as in Example 1, which is stored in compressed-diagonal storage mode in arrays AD and LA. The system is solved using the two-term conjugate gradient method. In this example, IOPT = 1, indicating that the matrix is symmetric, and only the main diagonal and one of each pair of identical diagonals are stored in array AD.

Call Statement and Input:

```
            IOPT  M   ND  AD  LDA  LA   B   X   IPARM   RPARM   AUX1  NAUX1  AUX2  NAUX2
             |    |   |   |    |   |    |   |     |       |       |      |      |     |
CALL DSDCG( 1  , 9 , 2 , AD , 9 , LA , B , X , IPARM , RPARM , AUX1 , 283 , AUX2 , 80  )
```

```
IPARM(1) =  20
IPARM(2) =  0
IPARM(3) =  10
RPARM(1) =  1.D-7
```

```
        | 2.0   0.0 |
        | 2.0   0.0 |
        | 2.0  -1.0 |
        | 2.0  -1.0 |
AD   =  | 2.0  -1.0 |
        | 2.0  -1.0 |
        | 2.0  -1.0 |
        | 2.0  -1.0 |
        | 2.0  -1.0 |
```

```
LA       =  (0, -2)
B        =  (1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

Output:

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  1
RPARM(2) =  0
RPARM(3) =  0.89D-16
```

DSMGCG (General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode)

Purpose

This subroutine solves a general sparse linear system of equations using an iterative algorithm, conjugate gradient squared or generalized minimum residual, with or without preconditioning by an incomplete LU factorization. The subroutine is suitable for positive real matrices—that is, when the symmetric part of the matrix, $(\boldsymbol{A}+\boldsymbol{A}^T)/2$, is positive definite. The sparse matrix is stored in compressed-matrix storage mode. Matrix ***A*** and vectors ***x*** and ***b*** are used:

$\boldsymbol{Ax} = \boldsymbol{b}$

where ***A***, ***x***, and ***b*** contain long-precision real numbers.

Note:

1. These subroutines are provided only for migration purposes. You get better performance and a wider choice of algorithms if you use the DSRIS subroutine.
2. If your sparse matrix is stored by rows, as defined in "Storage-by-Rows" on page 115, you should first use the utility subroutine DSRSM to convert your sparse matrix to compressed-matrix storage mode. See "DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)" on page 1212.

Syntax

Language	Syntax
Fortran	CALL DSMGCG (*m*, *nz*, *ac*, *ka*, *lda*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	dsmgcg (*m*, *nz*, *ac*, *ka*, *lda*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

m

is the order of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ and the number of rows in sparse matrix $\boldsymbol{A}$.

Specified as: an integer; $m \geq 0$.

nz

is the maximum number of nonzero elements in each row of sparse matrix $\boldsymbol{A}$.

Specified as: an integer; $nz \geq 0$.

ac

is the array, referred to as AC, containing the values of the nonzero elements of the sparse matrix, stored in compressed-matrix storage mode.

Specified as: an *lda* by (at least) *nz* array, containing long-precision real numbers.

ka

is the array, referred to as KA, containing the column numbers of the matrix $\boldsymbol{A}$ elements stored in the corresponding positions in array AC.

Specified as: an *lda* by (at least) *nz* array, containing integers, where 1 ≤ (elements of KA) ≤ *m*.

lda

is the leading dimension of the arrays specified for *ac* and *ka*.

Specified as: an integer; $lda > 0$ and $lda \geq m$.

b

is the vector $\boldsymbol{b}$ of length *m*, containing the right-hand side of the matrix problem.

Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

x

is the vector $\boldsymbol{x}$ of length *m*, containing your initial guess of the solution of the linear system.

Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers. The elements can have any value, and if no guess is available, the value can be zero.

iparm

is an array of parameters, `IPARM`(*i*), where:

- `IPARM(1)` controls the number of iterations.

  If `IPARM(1)` > 0, `IPARM(1)` is the maximum number of iterations allowed.

  If `IPARM(1)` = 0, the following default values are used:

  `IPARM(1)` = 300
  `IPARM(2)` = 0
  `IPARM(3)` = 10
  `RPARM(1)` = 10^{-6}

- `IPARM(2)` is the flag used to select the iterative procedure used in this subroutine.

  If `IPARM(2)` = 0, the conjugate gradient squared method is used.

  If `IPARM(2)` = *k*, the generalized minimum residual method, restarted after *k* steps, is used. Note that the size of the work area *aux1* becomes larger as *k* increases. A value for *k* in the range of 5 to 10 is suitable for most problems.

- `IPARM(3)` is the flag that determines whether the system is to be preconditioned by an incomplete LU factorization with no fill-in.

  If `IPARM(3)` = 0, the system is not preconditioned.

  If `IPARM(3)` = 10, the system is preconditioned by an incomplete LU factorization.

  If `IPARM(3)` = -10, the system is preconditioned by an incomplete LU factorization, where the factorization matrix was computed in an earlier call to this subroutine and is stored in *aux2*.

- `IPARM(4)`, see On Return.

Specified as: an array of (at least) length 4, containing integers, where:

`IPARM(1)` ≥ 0
`IPARM(2)` ≥ 0
`IPARM(3)` = 0, 10, or -10

rparm

is an array of parameters, RPARM(*i*), where:

`RPARM(1)` > 0, is the relative accuracy ε used in the stopping criterion. The iterative procedure is stopped when:

$\|\boldsymbol{b}\text{-}\boldsymbol{Ax}\|_2 / \|\boldsymbol{x}\|_2 < \varepsilon$

`RPARM(2)` is reserved.

`RPARM(3)`, see On Return.

Specified as: a one-dimensional array of (at least) length 3, containing long-precision real numbers.

aux1

has the following meaning:

If *naux1* = 0 and error 2015 is unrecoverable, *aux1* is ignored.

Otherwise, it is a storage work area used by this subroutine, which is available for use by the calling program between calls to this subroutine. Its size is specified by *naux1*.

Specified as: an area of storage, containing long-precision real numbers.

naux1

is the size of the work area specified by *aux1*—that is, the number of elements in *aux1*.

Specified as: an integer, where:

If *naux1* = 0 and error 2015 is unrecoverable, DSMGCG dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, it must have at least the following value, where:

If `IPARM(2)` = 0, use *naux1* ≥ 7*m*.

If `IPARM(2)` > 0, use *naux1* ≥ (*k*+2)*m*+*k*(*k*+4)+1, where *k* = `IPARM(2)`.

aux2

is the storage work area used by this subroutine. If `IPARM(3)` = -10, *aux2* must contain the incomplete LU factorization of matrix ***A***, computed in an earlier call to DSMGCG. The size of *aux2* is specified by *naux2*.

Specified as: an area of storage, containing long-precision real numbers.

naux2

is the size of the work area specified by *aux2*—that is, the number of elements in *aux2*.

Specified as: an integer. When `IPARM(3)` = 10, *naux2* must have at least the following value:

For 32-bit integer arguments
naux2 ≥ 3 + 2*m* + 1.5*nz*(*m*)

For 64-bit integer arguments
naux2 ≥ 12 + 3*m* + 2*nz*(*m*)

On Return

x

is the vector ***x*** of length *m*, containing the solution of the system ***Ax*** = ***b***. Returned as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

iparm

is an array of parameters, `IPARM`(*i*), where:

`IPARM(1)` is unchanged.

`IPARM(2)` is unchanged.

`IPARM(3)` is unchanged.

`IPARM(4)` contains the number of iterations performed by this subroutine.

Returned as: a one-dimensional array of length 4, containing integers.

rparm

is an array of parameters, `RPARM`(*i*), where:

`RPARM(1)` is unchanged.

`RPARM(2)` is reserved.

`RPARM(3)` contains the estimate of the error of the solution. If the process converged, `RPARM(3)` ≤ `RPARM(1)`

Returned as: a one-dimensional array of length 3, containing long-precision real numbers.

aux2

is the storage work area used by this subroutine.

If `IPARM(3)` = 10, *aux2* contains the incomplete LU factorization of matrix ***A***.

If `IPARM(3)` = -10, *aux2* is unchanged.

See "Notes" on page 785 for additional information on *aux2*. Returned as: an area of storage, containing long-precision real numbers.

Notes

1. When `IPARM(3)` = -10, this subroutine uses the incomplete LU factorization in *aux2*, computed in an earlier call to this subroutine. When `IPARM(3)` = 10, this subroutine computes the incomplete LU factorization and stores it in *aux2*.
2. If you solve the same sparse linear system of equations several times with different right-hand sides using the preconditioned algorithm, specify `IPARM(2)` = 10 on the first invocation. The incomplete factorization is stored in *aux2*. You may save computing time on subsequent calls by setting `IPARM(3)` equal to -10. In this way, the algorithm reutilizes the incomplete factorization that was computed the first time. Therefore, you should not modify the contents of *aux2* between calls.
3. Matrix ***A*** must have no common elements with vectors ***x*** and ***b***; otherwise, results are unpredictable.
4. In the iterative solvers for sparse matrices, the relative accuracy ε (`RPARM(1)`) must be specified "reasonably" (10^{-4} to 10^{-8}). The algorithm computes a sequence of approximate solution vectors ***x*** that converge to the solution. The iterative procedure is stopped when the norm of the residual is sufficiently small—that is, when:

   $\|\boldsymbol{b}-\boldsymbol{Ax}\|_2 / \|\boldsymbol{x}\|_2 < \varepsilon$

   As a result, if you specify a larger ε, the algorithm takes fewer iterations to converge to a solution. If you specify a smaller ε, the algorithm requires more iterations and computer time, but converges to a more precise solution. If the value you specify is unreasonably small, the algorithm may fail to converge within the number of iterations it is allowed to perform.
5. For a description of how sparse matrices are stored in compressed-matrix storage mode, see "Compressed-Matrix Storage Mode" on page 110.
6. On output, array AC is not bitwise identical to what it was on input because the matrix ***A*** is scaled before starting the iterative process and is unscaled before returning control to the user.

7. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

The linear system:

Ax = ***b***

is solved using either the conjugate gradient squared method or the generalized minimum residual method, with or without preconditioning by an incomplete LU factorization, where:

A is a sparse matrix of order *m*, stored in compressed-matrix storage mode in arrays AC and KA.

x is a vector of length *m*.

b is a vector of length *m*.

See references ["107" on page 1258] and ["109" on page 1258]. ["46" on page 1255].

If your program uses a sparse matrix stored by rows and you want to use this subroutine, first convert your sparse matrix to compressed-matrix storage mode by using the subroutine DSRSM (see "DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)" on page 1212).

Error conditions

Resource Errors

Error 2015 is unrecoverable, *naux1* = 0, and unable to allocate work area.

Computational Errors

The following errors, with their corresponding return codes, can occur in this subroutine. For details on error handling, see "What Can You Do about ESSL Computational Errors?" on page 65.

- For error 2110, return code 1 indicates that the subroutine exceeded `IPARM(1)` iterations without converging. Vector ***x*** contains the approximate solution computed at the last iteration.
- For error 2111, return code 2 indicates that *aux2* contains an incorrect factorization. The subroutine has been called with `IPARM(3)` = -10, and *aux2* contains an incomplete factorization of the input matrix ***A*** that was computed by a previous call to the subroutine when `IPARM(3)` = 10. This error indicates that *aux2* has been modified since the last call to the subroutine, or that the input matrix is not the same as the one that was factored. If the default action has been overridden, the subroutine can be called again with the same parameters, with the exception of `IPARM(3)` = 0 or 10.
- For error 2112, return code 3 indicates that the incomplete LU factorization of ***A*** could not be completed, because one pivot was 0.
- For error 2116, return code 4 indicates that the matrix is singular, because all elements in one row of the matrix contain 0. Array AC is partially modified and does not represent the same matrix as on entry.

Input-Argument Errors

1. *m* < 0
2. *lda* < 1
3. *lda* < *m*
4. *nz* < 0
5. *nz* = 0 and *m* > 0
6. `IPARM(1)` < 0
7. `IPARM(2)` < 0
8. `IPARM(3)` ≠ 0, 10, or -10
9. `RPARM(1)` < 0
10. `RPARM(2)` < 0

11. Error 2015 is recoverable or *naux1*≠0, and *naux1* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.
12. *naux2* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.

Examples

Example 1

This example finds the solution of the linear system ***Ax*** = ***b*** for the sparse matrix ***A***, which is stored in compressed-matrix storage mode in arrays AC and KA. The system is solved using the conjugate gradient squared method. Matrix ***A*** is:

```
        | 2.0  0.0   0.0  0.0   0.0   0.0   0.0   0.0   0.0 |
        | 0.0  2.0  -1.0  0.0   0.0   0.0   0.0   0.0   0.0 |
        | 0.0  1.0   2.0  0.0   0.0   0.0   0.0   0.0   0.0 |
        | 1.0  0.0   0.0  2.0  -1.0   0.0   0.0   0.0   0.0 |
        | 0.0  0.0   0.0  1.0   2.0  -1.0   0.0   0.0   0.0 |
        | 0.0  0.0   0.0  0.0   1.0   2.0  -1.0   0.0   0.0 |
        | 0.0  0.0   0.0  0.0   0.0   1.0   2.0  -1.0   0.0 |
        | 0.0  0.0   0.0  0.0   0.0   0.0   1.0   2.0  -1.0 |
        | 0.0  0.0   0.0  0.0   0.0   0.0   0.0   1.0   2.0 |
```

Note: For input matrix KA, (.) indicates any value between 1 and 9.

Call Statement and Input:

```
              M   NZ   AC   KA   LDA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2   NAUX2
              |    |    |    |    |    |   |     |       |       |      |       |       |
CALL DSMGCG( 9 ,  3 , AC , KA ,  9  , B , X , IPARM , RPARM , AUX1 ,  63   , AUX2 ,   0 )
```

```
IPARM(1) =  20
IPARM(2) =  0
IPARM(3) =  0
RPARM(1) =  1.D-7
```

```
        | 2.0   0.0   0.0 |
        | 2.0  -1.0   0.0 |
        | 1.0   2.0   0.0 |
        | 1.0   2.0  -1.0 |
AC   =  | 1.0   2.0  -1.0 |
        | 1.0   2.0  -1.0 |
        | 1.0   2.0  -1.0 |
        | 1.0   2.0  -1.0 |
        | 1.0   2.0   0.0 |
```

```
        | 1  .  . |
        | 2  3  . |
        | 2  3  . |
        | 1  4  5 |
KA   =  | 4  5  6 |
        | 5  6  7 |
        | 6  7  8 |
        | 7  8  9 |
        | 8  9  . |
```

```
B        =  (2.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 3.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

Output:

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  9
RPARM(3) =  0.150D-19
```

Example 2

This example finds the solution of the linear system ***Ax*** = ***b*** for the same sparse matrix ***A*** as in Example 1, which is stored in compressed-matrix storage mode in arrays AC and KA. The system is solved using the generalized minimum residual method, restarted after 5 steps and preconditioned with an incomplete LU factorization. Most of the input is the same as in Example 1.

Note: For input matrix KA, (.) indicates any value between 1 and 9.

Call Statement and Input:

```
               M    NZ    AC    KA    LDA    B    X    IPARM    RPARM    AUX1    NAUX1    AUX2    NAUX2
               |    |     |     |     |      |    |    |        |        |       |        |       |
CALL DSMGCG( 9 ,  3 , AC , KA ,  9  , B , X , IPARM , RPARM , AUX1 ,  109  , AUX2 ,  46 )
```

```
IPARM(1) = 20
IPARM(2) = 5
IPARM(3) = 10
RPARM(1) = 1.D-7
```

AC =(same as input AC in Example 1)
KA =(same as input KA in Example 1)

```
B = (2.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 3.0)
X = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

Output:

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  2
RPARM(3) =  0.290D-15
```

DSDGCG (General Sparse Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)

Purpose

This subroutine solves a general sparse linear system of equations using an iterative algorithm, conjugate gradient squared or generalized minimum residual, with or without preconditioning by an incomplete LU factorization. The subroutine is suitable for positive real matrices—that is, when the symmetric part of the matrix, ($\boldsymbol{A}+\boldsymbol{A}^T$)/2, is positive definite. The sparse matrix is stored in compressed-diagonal storage mode. Matrix ***A*** and vectors ***x*** and ***b*** are used:

Ax = ***b***

where ***A***, ***x***, and ***b*** contain long-precision real numbers.

Syntax

Language	**Syntax**
Fortran	CALL DSDGCG (*m*, *nd*, *ad*, *lda*, *la*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	dsdgcg (*m*, *nd*, *ad*, *lda*, *la*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

m

is the order of the linear system ***Ax*** = ***b*** and the number of rows in sparse matrix ***A***.

Specified as: an integer; *m* ≥ 0.

nd

is the number of nonzero diagonals stored in the columns of array AD, the number of columns in array AD, and the number of elements in array LA.

Specified as: an integer; it must have the following value, where:

If *m* > 0, then *nd* > 0.

If *m* = 0, then *nd* ≥ 0.

ad

is the array, referred to as AD, containing the values of the nonzero elements of the sparse matrix, stored in compressed-matrix storage mode.

Specified as: an *lda* by (at least) *nd* array, containing long-precision real numbers.

lda

is the leading dimension of the arrays specified for *ad*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *m*.

la

is the array, referred to as LA, containing the diagonal numbers *k* for the diagonals stored in each corresponding column in array AD. For an explanation of how diagonal numbers are stored, see "Compressed-Diagonal Storage Mode" on page 111.

Specified as: a one-dimensional array of (at least) length *nd*, containing integers, where 1-*m* ≤ (elements of LA) ≤ (*m*-1).

b

is the vector ***b*** of length *m*, containing the right-hand side of the matrix problem.

Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

x

is the vector ***x*** of length *m*, containing your initial guess of the solution of the linear system.

Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers. The elements can have any value, and if no guess is available, the value can be zero.

iparm

is an array of parameters, `IPARM`(*i*), where:

- `IPARM(1)` controls the number of iterations.

  If `IPARM(1)` > 0, `IPARM(1)` is the maximum number of iterations allowed.

  If `IPARM(1)` = 0, the following default values are used:

  `IPARM(1)` = 300
  `IPARM(2)` = 0
  `IPARM(3)` = 10
  `RPARM(1)` = 10^{-6}

- `IPARM(2)` is the flag used to select the iterative procedure used in this subroutine.

  If `IPARM(2)` = 0, the conjugate gradient squared method is used.

  If `IPARM(2)` = *k*, the generalized minimum residual method, restarted after *k* steps, is used. Note that the size of the work area *aux1* becomes larger as *k* increases. A value for *k* in the range of 5 to 10 is suitable for most problems.

- `IPARM(3)` is the flag that determines whether the system is to be preconditioned by an incomplete LU factorization with no fill-in.

  If `IPARM(3)` = 0, the system is not preconditioned.

  If `IPARM(3)` = 10, the system is preconditioned by an incomplete LU factorization.

  If `IPARM(3)` = -10, the system is preconditioned by an incomplete LU factorization, where the factorization matrix was computed in an earlier call to this subroutine and is stored in *aux2*.

- `IPARM(4)`, see On Return.

Specified as: an array of (at least) length 4, containing integers, where:

IPARM(1) ≥ 0
IPARM(2) ≥ 0
IPARM(3) = 0, 10, or -10

rparm
is an array of parameters, RPARM(*i*), where:

If RPARM(1) > 0, is the relative accuracy ε used in the stopping criterion. The iterative procedure is stopped when:

$\|\boldsymbol{b}-\boldsymbol{Ax}\|_2 / \|\boldsymbol{x}\|_2 < \varepsilon$

RPARM(2) is reserved.

RPARM(3), see On Return.

Specified as: a one-dimensional array of (at least) length 3, containing long-precision real numbers.

aux1
has the following meaning:

If *naux1* = 0 and error 2015 is unrecoverable, *aux1* is ignored.

Otherwise, it is a storage work area used by this subroutine, which is available for use by the calling program between calls to this subroutine. Its size is specified by *naux1*.

Specified as: an area of storage, containing long-precision real numbers.

naux1
is the size of the work area specified by *aux1*—that is, the number of elements in *aux1*.

Specified as: an integer, where:

If *naux1* = 0 and error 2015 is unrecoverable, DSDGCG dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux1* > 0 and must have at least the following value, where:

If IPARM(2) = 0, use *naux1* ≥ 7*m*.

If IPARM(2) > 0, use *naux1* ≥ (*k*+2)*m*+*k*(*k*+4)+1, where *k* = PARM(2).

aux2
is a storage work area used by this subroutine. If IPARM(3) = -10, *aux2* must contain the incomplete LU factorization of matrix ***A***, computed in an earlier call to DSDGCG. The size of *aux2* is specified by *naux2*.

Specified as: an area of storage, containing long-precision real numbers.

naux2
is the size of the work area specified by *aux2*—that is, the number of elements in *aux2*.

Specified as: an integer. When IPARM(3) = 10 or -10, *naux2* must have at least the following value:

For 32-bit integer arguments
naux2 ≥ 3 + 2*m* + 1.5*nd*(*m*)

For 64-bit integer arguments
naux2 ≥ 12 + 3*m* + 2*nd*(*m*)

On Return

x
is the vector ***x*** of length *m*, containing the solution of the system ***Ax*** = ***b***. Returned as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

iparm
is an array of parameters, IPARM(*i*), where:

IPARM(1) is unchanged.

`IPARM(2)` is unchanged.

`IPARM(3)` is unchanged.

`IPARM(4)` contains the number of iterations performed by this subroutine.

Returned as: a one-dimensional array of length 4, containing integers.

rparm

is an array of parameters, RPARM(*i*), where:

`RPARM(1)` is unchanged.

`RPARM(2)` is reserved.

`RPARM(3)` contains the estimate of the error of the solution. If the process converged, `RPARM(3)` ≤ `RPARM(1)`.

Returned as: a one-dimensional array of length 3, containing long-precision real numbers.

aux2

is the storage work area used by this subroutine.

If `IPARM(3)` = 10, *aux2* contains the incomplete LU factorization of matrix ***A***.

If `IPARM(3)` = -10, *aux2* is unchanged.

See "Notes" on page 791 for additional information on *aux2*. Returned as: an area of storage, containing long-precision real numbers.

Notes

1. When `IPARM(3)` = -10, this subroutine uses the incomplete LU factorization in *aux2*, computed in an earlier call to this subroutine. When `IPARM(3)` = 10, this subroutine computes the incomplete LU factorization and stores it in *aux2*.
2. If you solve the same sparse linear system of equations several times with different right-hand sides, using the preconditioned algorithm, specify `IPARM(3)` = 10 on the first invocation. The incomplete factorization is stored in *aux2*. You may save computing time on subsequent calls by setting `IPARM(3)` = -10. In this way, the algorithm reutilizes the incomplete factorization that was computed the first time. Therefore, you should not modify the contents of *aux2* between calls.
3. Matrix ***A*** must have no common elements with vectors ***x*** and ***b***; otherwise, results are unpredictable.
4. In the iterative solvers for sparse matrices, the relative accuracy ε (`RPARM(1)`) must be specified "reasonably" (10^{-4} to 10^{-8}). The algorithm computes a sequence of approximate solution vectors ***x*** that converge to the solution. The iterative procedure is stopped when the norm of the residual is sufficiently small—that is, when:

   $\|\boldsymbol{b}-\boldsymbol{Ax}\|_2 / \|\boldsymbol{x}\|_2 < \varepsilon$

   As a result, if you specify a larger ε, the algorithm takes fewer iterations to converge to a solution. If you specify a smaller ε, the algorithm requires more iterations and computer time, but converges to a more precise solution. If the value you specify is unreasonably small, the algorithm may fail to converge within the number of iterations it is allowed to perform.
5. For a description of how sparse matrices are stored in compressed-diagonal storage mode, see "Compressed-Diagonal Storage Mode" on page 111.
6. On output, array AD is not bitwise identical to what it was on input, because matrix ***A*** is scaled before starting the iterative process and is unscaled before returning control to the user.
7. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

The linear system:

Ax = ***b***

is solved using either the conjugate gradient squared method or the generalized minimum residual method, with or without preconditioning by an incomplete LU factorization, where:

A is a sparse matrix of order *m*, stored in compressed-diagonal storage mode in arrays AD and LA.

x is a vector of length *m*.

b is a vector of length *m*.

See references [“107” on page 1258] and [“109” on page 1258]. [“46” on page 1255].

Error conditions

Resource Errors

Error 2015 is unrecoverable, *naux1* = 0, and unable to allocate work area.

Computational Errors

The following errors, with their corresponding return codes, can occur in this subroutine. For details on error handling, see “What Can You Do about ESSL Computational Errors?” on page 65.

- For error 2110, return code 1 indicates that the subroutine exceeded `IPARM(1)` iterations without converging. Vector ***x*** contains the approximate solution computed at the last iteration.
- For error 2111, return code 2 indicates that *aux2* contains an incorrect factorization. The subroutine has been called with `IPARM(3)` = -10, and *aux2* contains an incomplete factorization of the input matrix ***A*** that was computed by a previous call to the subroutine when `IPARM(3)` = 10. This error indicates that *aux2* has been modified since the last call to the subroutine, or that the input matrix is not the same as the one that was factored. If the default action has been overridden, the subroutine can be called again with the same parameters, with the exception of `IPARM(3)` = 0 or 10.
- For error 2112, return code 3 indicates that the incomplete LU factorization of ***A*** could not be completed, because one pivot was 0.
- For error 2116, return code 4 indicates that the matrix is singular, because all elements in one row of the matrix contain 0. Array AC is partially modified and does not represent the same matrix as on entry.

Input-Argument Errors

1. *m* < 0
2. *lda* < 1
3. *lda* < *m*
4. *nd* < 0
5. *nd* = 0 and *m* > 0
6. `IPARM(1)` < 0
7. `IPARM(2)` < 0
8. `IPARM(3)` ≠ 0, 10, or -10
9. `RPARM(1)` < 1.D0
10. Error 2015 is recoverable or *naux1*≠0, and *naux1* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.
11. *naux2* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.

Examples

Example 1

This example finds the solution of the linear system ***Ax*** = ***b*** for the sparse matrix ***A***, which is stored in compressed-diagonal storage mode in arrays AD and LA. The system is solved using the conjugate gradient squared method. Matrix ***A*** is:

```
        | 2.0  0.0  -1.0   0.0   0.0   0.0   0.0   0.0   0.0 |
        | 0.0  2.0   0.0  -1.0   0.0   0.0   0.0   0.0   0.0 |
        | 0.0  0.0   2.0   0.0  -1.0   0.0   0.0   0.0   0.0 |
        | 0.0  0.0   0.0   2.0   0.0  -1.0   0.0   0.0   0.0 |
        | 1.0  0.0   0.0   0.0   2.0   0.0  -1.0   0.0   0.0 |
        | 0.0  1.0   0.0   0.0   0.0   2.0   0.0  -1.0   0.0 |
        | 0.0  0.0   1.0   0.0   0.0   0.0   2.0   0.0  -1.0 |
        | 0.0  0.0   0.0   1.0   0.0   0.0   0.0   2.0   0.0 |
        | 0.0  0.0   0.0   0.0   1.0   0.0   0.0   0.0   2.0 |
```

Call Statement and Input:

```
             M   ND   AD   LDA   LA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2   NAUX2
             |    |    |    |     |   |   |     |       |       |      |       |       |
CALL DSDGCG( 9 ,  3 , AD ,  9   , LA , B , X , IPARM , RPARM , AUX1 ,  63   , AUX2 ,   0 )
```

```
IPARM(1) =  20
IPARM(2) =  0
IPARM(3) =  0
RPARM(1) =  1.D-7
```

```
        | 2.0  -1.0  0.0 |
        | 2.0  -1.0  0.0 |
        | 2.0  -1.0  0.0 |
        | 2.0  -1.0  0.0 |
AD   =  | 2.0  -1.0  1.0 |
        | 2.0  -1.0  1.0 |
        | 2.0  -1.0  1.0 |
        | 2.0   0.0  1.0 |
        | 2.0   0.0  1.0 |
```

```
LA       =  (0, 2, -4)
B        =  (1, 1, 1, 1, 2, 2, 2, 3, 3)
X        =  (0, 0, 0, 0, 0, 0, 0, 0, 0)
```

Output:

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  8
RPARM(3) =  0.308D-17
```

Example 2

This example finds the solution of the linear system ***Ax*** = ***b*** for the same sparse matrix ***A*** as in Example 1, which is stored in compressed-diagonal storage mode in arrays AD and LA. The system is solved using the generalized minimum residual method, restarted after 5 steps and preconditioned with an incomplete LU factorization. Most of the input is the same as in Example 1.

Call Statement and Input:

```
             M   ND   AD   LDA   LA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2   NAUX2
             |    |    |    |     |   |   |     |       |       |      |       |       |
CALL DSDGCG( 9 ,  3 , AD ,  9   , LA , B , X , IPARM , RPARM , AUX1 ,  109  , AUX2 ,  46 )
```

```
IPARM(1) = 20
IPARM(2) = 5
IPARM(3) = 10
RPARM(1) = 1.D-7
```

AD =(same as input AD in Example 1)
LA =(same as input LA in Example 1)
B = (1, 1, 1, 1, 2, 2, 2, 3, 3)
X = (0, 0, 0, 0, 0, 0, 0, 0, 0)

Output:

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  6
RPARM(3) =  0.250D-15
```

Linear Least Squares Subroutines

This contains the linear least squares subroutine descriptions.

SGESVD, DGESVD, CGESVD, ZGESVD, SGESDD, DGESDD, CGESDD, and ZGESDD (Singular Value Decomposition for a General Matrix)

Purpose

These subroutines compute the singular value decomposition of a general matrix ***A***, optionally computing the left and/or right singular vectors. The singular value decomposition is written:

For SGESVD, DGESVD, SGESDD, and DGESDD, $\boldsymbol{A} = \boldsymbol{U}\Sigma\boldsymbol{V}^{T}$, where $\boldsymbol{U}^{T} = \boldsymbol{U}^{-1}$ and $\boldsymbol{V}^{T} = \boldsymbol{V}^{-1}$

For CGESVD, ZGESVD, CGESDD, and ZGESDD, $\boldsymbol{A} = \boldsymbol{U}\Sigma\boldsymbol{V}^{H}$, where $\boldsymbol{U}^{H} = \boldsymbol{U}^{-1}$ and $\boldsymbol{V}^{H} = \boldsymbol{V}^{-1}$

In the formulas above:

- ***U*** and ***V*** are general matrices whose first min(*m*, *n*) columns are the left and right singular vectors of ***A***.
- ***Σ*** is a diagonal matrix whose min(*m*, *n*) diagonal elements are the singular values of ***A***.

Table 192: Data Types.

A, U, vt, work	***s, superb, rwork***	**Subroutine**
Short-precision real	Short-precision real	SGESVD$^{\Delta}$, SGESDD$^{\Delta}$
Long-precision real	Long-precision real	DGESVD$^{\Delta}$, DGESDD$^{\Delta}$
Short-precision complex	Short-precision real	CGESVD$^{\Delta}$, CGESDD$^{\Delta}$
Long-precision complex	Long-precision real	ZGESVD$^{\Delta}$, ZGESDD$^{\Delta}$
$^{\Delta}$LAPACK		

Syntax

Table 193: Syntax for supported computing languages.

Language	Syntax
Fortran	CALL SGESVD | DGESVD (*jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, info*) CALL CGESVD | ZGESVD (*jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, info*) CALL SGESDD | DGESDD (*jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info*) CALL CGESDD | ZGESDD (*jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, iwork, info*)
C and C++	sgesvd | dgesvd (*jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, info*); cgesvd | zgesvd (*jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, info*); sgesdd | dgesdd (*jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info*); cgesdd | zgesdd (*jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, iwork, info*);

LAPACKE	info = LAPACKE_sgesvd | LAPACKE_dgesvd (*matrix_layout, jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, superb*);
	info = LAPACKE_cgesvd | LAPACKE_zgesvd (*matrix_layout, jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, superb*);
	info = LAPACKE_sgesdd | LAPACKE_dgesdd (*matrix_layout, jobz, m, n, a, lda, s, u, ldu, vt, ldvt*);
	info = LAPACKE_cgesdd | LAPACKE_zgesdd (*matrix_layout, jobz, m, n, a, lda, s, u, ldu, vt, ldvt*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

jobu

indicates the options for computing all or part of matrix ***U***, where:

- If *jobu* = 'A', all *m* columns (the left singluar vectors) are returned in array *u*.
- If *jobu* = 'S', the first min(*m*, *n*) columns (the left singular vectors) are returned in array *u*.
- If *jobu* = 'O', the first min(*m*, *n*) columns (the left singular vectors) are overwritten on the array *a*.
- If *jobu* = 'N', no columns (no left singular vectors) are computed.

jobvt

indicates the options for computing all or part of $\boldsymbol{V}^T$ (for SGESVD/DGESVD) or $\boldsymbol{V}^H$ (for CGESVD/ZGESVD), where:

- If *jobvt* = 'A', all *n* rows (the right singular vectors) are returned in array *vt*.
- If *jobvt* = 'S', the first min(*m*, *n*) rows (the right singular vectors) are returned in array *vt*.
- If *jobvt* = 'O', the first min(*m*, *n*) rows (the right singular vectors) are overwritten in array *a*.
- If *jobvt* = 'N', no rows (no right singular vectors) are computed.

jobz
: indicates the options for computing all or part of ***U*** and $\boldsymbol{V}^T$ (for SGESDD/DGESDD) or $\boldsymbol{V}^H$ (for CGESDD/ZGESDD), where:

- If *jobz* = 'A', all *m* columns (the left singular vectors) are returned in array *u* and all *n* rows (the right singular vectors) are returned in array *vt*.
- If *jobz* = 'S', the first min(*m*, *n*) columns (the left singular vectors) are returned in array *u* and the first min(*m*, *n*) rows (the right singular vectors) are returned in array *vt*.
- If *jobz* = 'O' and *m* ≥ *n*, the first *n* columns (the left singular vectors) are overwritten in the array *a* and all rows (the right singular vectors) are returned in array *vt*.

  If *jobz* = 'O' and *m* < *n*, all columns (the left singular vectors) are returned in array *u* and first m rows (the right singular vectors) are overwritten on the array *a*.
- If *jobz* = 'N', no columns (the left singular vectors) or rows (the right singular vectors) are computed.

m
is the number of rows in matrix ***A*.**

Specified as: an integer; *m* ≥ 0.

n
is the number of columns in matrix ***A*.**

Specified as: an integer; *n* ≥ 0.

a
is the *m* by *n* general matrix ***A***, whose singular value decomposition is to be computed.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 192 on page 794.

lda
is the leading dimension of the array specified for ***A***.

Specified as: an integer; *lda* > 0 and *lda* ≥ *m*.

s
See "On Return".

u
See "On Return".

ldu
is the leading dimension of the array specified for ***U***.

Specified as: an integer; *ldu*>0 and:

If *jobu* = 'A' or 'S', *ldu* ≥ max(1, *m)*.

If *jobz* = 'A' or 'S', *ldu* ≥ max(1, *m*).

If *jobz* = 'O' and *m* < *n*, *ldu* ≥ max(1, *m*).

vt
See "On Return".

ldvt
is the leading dimension of the array specified for ***VT***.

Specified as: an integer; *ldvt*>0 and:

If *jobvt* = 'A', *ldvt* ≥ max(1, *n)*.

If *jobvt* = 'S', *ldvt* ≥ max(1, min(*m*, *n*)).

If *jobz* = 'A', *ldvt* ≥ max(1, *n*).

If *jobz* = 'S', *ldvt* ≥ max(1, min(*m*, *n*)).

If *jobz* = 'O', and *m* ≥ *n*, *ldvt* ≥ max(1, *n)*.

superb

is a work area of size min(*m*, *n*) - 1.

Specified as: an area of storage containing numbers of the data type indicated in Table 192 on page 794.

work

is the work area used by these subroutines, where:

If *lwork* = 0, *work* is ignored.

If *lwork* ≠ 0, the size of *work* is determined as follows:

- If *lwork* ≠ -1, *work* is (at least) of length *lwork*.
- If *lwork* = -1, *work* is (at least) of length 1.

Specified as: an area of storage, containing numbers of the data type indicated in Table 192 on page 794.

lwork

is the number of elements in array `WORK`.

Specified as: an integer; where:

- If *lwork* = 0, the subroutine dynamically allocates the workspace needed for use during this computation. The work area is deallocated before control is returned to the calling program.
- If *lwork* = -1, the subroutine performs a workspace query and returns the optimal required size of *work* in $work_1$. No computation is performed and the subroutine returns after error checking is complete.
- Otherwise:

For SGESVD and DGESVD

lwork ≥ max(1, 3*min(*m*, *n*) + max(*m*,*n*), 5*min(*m*, *n*))

Note: If *jobu* = 'N' or *jobvt* = 'N', depending on *m*, *n* and the implementation, *lwork* ≥ max(1, 5*min(*m*, *n*)) may be sufficient and less than the value shown above.

For CGESVD and ZGESVD

lwork ≥ max(1, 2*min(*m*, *n*) + max(*m*,*n*))

Note: If *jobu* = 'N' or *jobvt* = 'N', depending on *m*, *n* and the implementation, *lwork* ≥ max(1, 3*min(*m*, *n*)) may be sufficient and less than the value shown above.

For SGESDD and DGESDD

If *jobz* = 'N', *lwork* ≥ 3*min(*m*, *n*) + max(max(*m*, *n*), 7*min(*m*, *n*))

If *jobz* = 'O':

lwork ≥ 3*min(*m*, *n*) + max(max(*m*, *n*), 5*min(*m*, *n*)*min(*m*, *n*) + 4*min(*m*, *n*))

If *jobz* = 'S', *lwork* ≥ 4*min(*m*, *n*)*min(*m*, *n*) + 7*min(*m*, *n*)

If *jobz* = 'A', *lwork* ≥ 4*min(*m*, *n*)*min(*m*, *n*) + 6*min(*m*, *n*) + max(*m*, *n*)

For CGESDD and ZGESDD

If *jobz* = 'N', *lwork* ≥ 2*min(*m*, *n*) + max(*m*, *n*)

If *jobz* = 'O', *lwork* ≥ 2*min(*m*, *n*)*min(*m*, *n*) + 2*min(*m*, *n*) + max(*m*, *n*)

If *jobz* = 'S', *lwork* ≥ min(*m*, *n*)*min(*m*, *n*) + 3*min(*m*, *n*)

If *jobz* = 'A', *lwork* ≥ min(*m*, *n*)*min(*m*, *n*) + 2*min(*m*, *n*) + max(*m*, *n*)

iwork

is a storage work area of size 8*min(*m, n*).

Specified as: an integer array.

rwork

For CGESVD and ZGESVD

rwork is a work area of size (at least) 5*min(*m, n*).

For CGESDD and ZGESDD

rwork is a work area of size (at least) max(1, *lrwork*).

If *jobz* = 'N', *lrwork* ≥ 5*min(*m, n*).

If max(*m, n*) >> min(*m, n*), *lwork* ≥ 5*min(*m, n*)*min(*m, n*) + 5*min(*m, n*).

Otherwise, *lrwork* ≥ max(5*min(*m, n*)*min(*m, n*) + 5*min(*m, n*), 2*max(*m, n*)*min(*m, n*) + 2*min(*m, n*)*min(*m, n*) + min(*m, n*)).

Note: LRWORK depends on whether or not max(*m*,*n*) >> min(*m*, *n*). ESSL is compatible with LAPACK 3.6.1 where max(*m*,*n*) >> min(*m*, *n*) is defined as max(*m*,*n*) >= 17.0*min(*m*, *n*)/9.0. If this definition changes in future LAPACK releases, IBM will consider updating these subroutines. If IBM updates these subroutines, the updates could require modifications of the calling application program.

Specified as: an area of storage containing numbers of the data type indicated in Table 192 on page 794.

On Return

a

is overwritten as follows:

If *jobu* = 'O' or *jobz* = 'O' and *m* ≥ *n*, *u* is not referenced. Instead, *a* is overwritten with the first min(*m*, *n*) columns of ***U***. These are the left singular vectors, stored column-wise.

If *jobvt* = 'O' or *jobz* = 'O' and *m* < *n*, *vt* is not referenced. Instead, *a* is overwritten with the first min(*m*, *n*) rows of $\boldsymbol{V}^{\mathbf{T}}$ (for SGESVD, DGESVD, SGESDD, or DGESDD) or $\boldsymbol{V}^{H}$ (for CGESVD, ZGESVD, CGESDD, or ZGESDD). These are the right singular vectors, stored row-wise.

If *jobu* ≠ 'O' and *jobvt* ≠ 'O' and *jobz* ≠ 'O', the contents of *a* are overwritten on return.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 192 on page 794.

s

is the vector ***s*** containing the non-negative singular values in descending order in the first min(*m*, *n*) elements of ***s***.

Returned as: a one-dimensional array of (at least) length min(*m*, *n*) containing numbers of the data type indicated in Table 192 on page 794 .

u

If *jobu* = 'A' or if *jobz* = 'A' or if *jobz* = 'O' and *m* < *n*, *u* contains the *m* by *m* matrix ***U***.

If *jobu* = 'S' or if *jobz* = 'S', *u* contains first min(*m*, *n*) columns of ***U***. These are the left singular vectors, stored column-wise.

If *jobu* = 'O' or 'N' or *jobz* = 'N' or *jobz* = 'O' and *m* ≥ *n*, *u* is not referenced.

Returned as: an *ldu* by (at least) *m* (if *jobu* = 'A' or if *jobz* = 'A' or if *jobz* = 'O' and *m* < *n*) or min(*m*, *n*) (if *jobu* = 'S' or if *jobz* = 'S') array containing numbers of the data type indicated in Table 192 on page 794.

vt

For SGESVD, DGESVD, SGESDD, and DGESDD:

- If *jobvt* = 'A' or *jobz* = 'A' or *jobz* = 'O' and *m* ≥ *n*, *vt* contains the matrix $\boldsymbol{V}^{\mathbf{T}}$ of order *n*.
- If *jobvt* = 'S' or *jobz* = 'S', *vt* contains the first min(*m*, *n*) rows of $\boldsymbol{V}^{\mathbf{T}}$. These are the right singular vector, stored row-wise.
- If *jobvt* = 'O' or 'N' or *jobz* = 'N' or *jobz* = 'O' and *m* < *n*, *vt* is not referenced.

Note: These subroutines return $\boldsymbol{V}^{\mathbf{T}}$ instead of ***V***

For CGESVD, ZGESVD, CGESDD, and ZGESDD:

- If *jobvt* = 'A' or *jobz* = 'A' or *jobz* = 'O' *and m* ≥ *n*, *vt* contains the matrix $\boldsymbol{V^H}$ of order *n*.
- If *jobvt* = 'S' or *jobz* = 'S', *vt* contains the first min(*m*, *n*) rows of $\boldsymbol{V^H}$. These are the right singular vector, stored row-wise.
- If *jobvt* = 'O' or 'N' or *jobz* = 'N' or *jobz* = 'O' and *m* < *n*, *vt* is not referenced.

Note: These subroutines return $\boldsymbol{V^H}$ instead of ***V***

Returned as: an *ldvt* by (at least) *n* array, containing numbers of the data type indicated in Table 192 on page 794.

superb

is a work area used by this subroutine.

Returned as: an area of storage. If info > 0, *superb* contains the unconverged superdiagonal elements of the upper bidiagonal matrix ***B*** whose diagonal is in *s*.

work

is a work area used by this subroutine if *lwork* ≠ 0, where:

If *lwork* ≠ 0 and *lwork* ≠ -1, its size is (at least) of length *lwork*.

If *lwork* = -1, its size is (at least) of length 1.

If *lwork* = 0, *work* is not referenced.

Returned as: an area of storage where:

If *lwork* ≥ 1 or *lwork* = -1, then $work_1$ is set to the optimal *lwork* value.

For SGESVD or DGESVD, if *lwork* ≥ 1 and *info* > 0, $work_{2:\min(m,\ n)}$ contains the unconverged superdiagonal elements of an upper bidiagonal matrix ***B***, whose diagonal is in array S (not necessarily sorted). ***B*** satisfies $\boldsymbol{A} = \boldsymbol{UBV}^T$, so it has the same singular values as ***A***, and singular vectors related by ***U*** and $\boldsymbol{V}^T$.

rwork

is a work area used by this subroutine.

Returned as: an area of storage where:

For CGESVD and ZGESVD, if *info* > 0, $rwork_{1:\min(m,\ n)-1}$ contains the unconverged superdiagonal elements of an upper bidiagonal matrix ***B*** whose diagonal is in array *s* (not necessarily sorted). ***B*** satisfies $\boldsymbol{A} = \boldsymbol{UBV}^H$, so it has the same singular values as ***A***, and singular vectors related by ***U*** and $\boldsymbol{V}^H$.

info

has the following meaning:

If *info* = 0, the subroutine completed successfully.

For SGESVD, DGESVD, CGESVD, ZGESVD, LAPACKE_sgesvd, LAPACKE_dgesvd, LAPACKE_cgesvd, and LAPACKE_zgesvd:

If *info* > 0, *info* specifies how many superdiagonals of an intermediate bidiagonal form ***B*** did not converge to zero.

For SGESDD, DGESDD, CGESDD, ZGESDD, LAPACKE_sgesdd, LAPACKE_dgesdd, LAPACKE_cgesdd, and LAPACKE_zgesdd:

If *info* > 0, the algorithm did not converge to zero and the update process failed. The singular values and singular vectors did not converge.

Returned as:

- For SGESVD, DGESVD, CGESVD, ZGESVD, SGESDD, DGESDD, CGESDD, and ZGESDD, returned as: an integer; info ≥ 0.
- For LAPACKE_sgesvd, LAPACKE_dgesvd, LAPACKE_cgesvd, LAPACKE_zgesvd, LAPACKE_sgesdd, LAPACKE_dgesdd, LAPACKE_cgesdd, and LAPACKE_zgesdd, returned as an integer function value; info ≥ 0.

Returned as: an integer, *info* ≥ 0.

Notes:

1. These subroutines accept lowercase letters for the *jobu*, *jobvt*, and *jobz* arguments.
2. In your C program, argument *info* must be passed by reference.
3. When you specify *jobu* = 'O' or 'N' or *jobz* = 'O' or 'N', you must specify a dummy argument for *u*.
4. When you specify *jobvt* = 'O' or 'N' or *jobz* = 'O' or 'N', you must specify a dummy argument for *vt*.
5. You cannot specify both *jobu* = 'O' and *jobvt* = 'O'.
6. *a*, *s*, *u*, *vt*, *work*, *rwork*, and *iwork* must have no common elements; otherwise, results are unpredictable.
7. For best performance, specify *lwork* = 0.

Function

These subroutines compute the singular value decomposition of a general matrix ***A***, optionally computing the left and/or right singular vectors. The singular value decomposition is written:

For SGESVD, DGESVD, SGESDD, and DGESDD, $\boldsymbol{A} = \boldsymbol{U}\Sigma\boldsymbol{V}^T$, where $\boldsymbol{U}^T = \boldsymbol{U}^{-1}$ and $\boldsymbol{V}^T = \boldsymbol{V}^{-1}$

For CGESVD, ZGESVD, CGESDD, and ZGESDD, $\boldsymbol{A} = \boldsymbol{U}\Sigma\boldsymbol{V}^H$, where $\boldsymbol{U}^H = \boldsymbol{U}^{-1}$ and $\boldsymbol{V}^H = \boldsymbol{V}^{-1}$

In the formulas above:

- ***U*** and ***V*** are general matrices whose first min(*m*, *n*) columns are the left and right singular vectors of ***A***.
- ***Σ*** is a diagonal matrix whose min(*m*, *n*) diagonal elements are the singular values of ***A***.

The computation involves the following steps:

1. If necessary, scale ***A***
2. If necessary, compute QR or LQ factorization
3. Bidiagonalize the matrix
4. For SGESVD, DGESVD, CGESVD, and ZGESD, compute the singular values and, optionally, the left and/or right singular vectors from the bidiagonalized matrix.

   For SGESDD, DGESDD, CGESDD, and ZGESDD, compute the singular values and, optionally, the left and/or right singular vectors from the bidiagonalized matrix using the divide and conquer algorithm.
5. If necessary, update the singular vectors
6. If necessary, undo scaling

If *m* or *n* is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See references [“75” on page 1257, “106” on page 1258].

Error conditions

Resource Errors
lwork = 0 and unable to allocate work space

Computational Errors
For SGESVD, DGESVD, CGESVD, and ZGESD, at least *info* superdiagonals of an intermediate bidiagonal form ***B*** did not converge to zero.

For SGESDD, DGESDD, CGESDD, and ZGESDD, singular values and/or singular vectors did not converge.

Input-Argument Errors

1. *jobu* ≠ 'A', 'S', 'O', or 'N'

2. *jobvt* ≠ 'A', 'S', 'O', or 'N'
3. *jobu* = 'O' and *jobvt* = 'O'
4. *jobz* ≠ 'A', 'S', 'O', or 'N'
5. *m* < 0
6. *n* < 0
7. *lda* ≤ 0
8. *m* > *lda*
9. *ldu* ≤ 0
10. *m* > *ldu* and (*jobu* = 'A' or *jobz* = 'S')
11. *m* > *ldu* and (*jobz* = 'A' or *jobz* = 'S')
12. *m* > *ldu* and (*jobz* = 'O' and *m* < *n*
13. *ldvt* ≤ 0
14. *n* > *ldvt* and *jobvt* = 'A'
15. *n* > *ldvt* and *jobz* = 'A'
16. *n* > *ldvt* and *jobz* = 'O' and *m* ≥ *n*
17. min(*m*, *n*) > *ldvt* and *jobvt* = 'S'
18. min(*m*, *n*) > *ldvt* and *jobz* = 'S'
19. *lwork* ≠ 0 and *lwork* ≠ -1 and *lwork* < the required value
20. The size of a work array is greater than 2147483647 when 32-bit integers are used.

Examples

Example 1

This example shows how to find the singular values only of the real general matrix ***A***.

Notes:

1. For DGESVD, because *jobu* = 'N', argument *u* is not referenced.
2. For DGESVD, because *jobvt* = 'N', argument *vt* is not referenced.
3. For DGESDD, because *jobz* = 'N', arguments *u* and *vt* are not referenced.
4. Because *lwork* = 0, the subroutine dynamically allocates `WORK`.
5. *iwork* is an integer work array of size 64.

Call Statement and Input:

```
             JOBU  JOBVT  M   N    A  LDA  S   U  LDU  VT  LDVT  WORK  LWORK  INFO
              |     |     |   |    |   |   |   |   |    |    |     |     |      |
CALL DGESVD( 'N' , 'N' , 4 , 4 ,  A , 4 , S , U , 1 , VT ,  1 , WORK ,  0 ,  INFO )
```

–or–

```
             JOBZ   M   N    A  LDA  S   U  LDU  VT  LDVT  WORK  LWORK IWORK   INFO
              |     |   |    |   |   |   |   |    |    |     |     |     |       |
CALL DGESDD( 'N' , 4 , 4 ,  A , 4 , S , U , 1 , VT ,  1 , WORK ,  0 , IWORK , INFO )
```

```
        |  1.0    1.0    0.0    0.0 |
A    =  |  0.0    2.0    1.0    0.0 |
        |  0.0    0.0    3.0    1.0 |
        |  0.0    0.0    0.0    4.0 |
```

Output:

Array A is overwritten.

```
        | 4.260007 |
S    =  | 3.107349 |
        | 2.111785 |
        | 0.858542 |


INFO = 0
```

Example 2

This example shows how to find the singular values of the real general matrix ***A*** and return all its left and right singular vectors ***U*** and ***VT*** in arrays U and V^T.

Note:

1. Because *lwork* = 0, the subroutine dynamically allocates WORK.
2. *iwork* is an integer work array of size 64.

Call Statement and Input:

```
             JOBU JOBVT  M   N    A  LDA  S   U  LDU  VT  LDVT  WORK  LWORK INFO
              |     |    |   |    |   |   |   |   |    |    |     |     |     |
CALL DGESVD( 'A' , 'A' , 3 , 3 ,  A , 3 , S , U , 3 , VT ,  3 , WORK ,  0 , INFO )
```

–or–

```
            JOBZ   M   N    A  LDA  S   U  LDU  VT  LDVT  WORK  LWORK IWORK   INFO
              |    |   |    |   |   |   |   |    |    |     |     |     |       |
CALL DGESDD( 'A' , 3 , 3 ,  A , 3 , S , U , 3 , VT ,  3 , WORK ,  0 , IWORK , INFO )
```

```
        |  1.0    2.0    3.0  |
A    =  |  2.0    4.0    5.0  |
        |  3.0    5.0    6.0  |
```

Output:

Array A is overwritten.

```
        | 11.344814 |
S    =  |  0.515729 |
        |  0.170915 |
```

```
        | -0.327985 -0.736976 -0.591009 |
U    =  | -0.591009 -0.327985  0.736976 |
        | -0.736976  0.591009 -0.327985 |
```

```
        | -0.327985 -0.591009 -0.736976 |
VT   =  |  0.736976  0.327985 -0.591009 |
        | -0.591009  0.736976 -0.327985 |
INFO = 0
```

Example 3

This example shows how to find the singular values of the real general matrix ***A***. Additionally:

The first min(*m*, *n*) columns of its left singular vectors *U* are returned in array U.

The first min(*m*, *n*) rows of its right singular vectors $\boldsymbol{V}^T$ are returned in array A.

Notes:

1. Because *jobvt* = 'O', argument *vt* is not referenced.
2. Because *lwork* = 0, the subroutine dynamically allocates WORK.
3. *iwork* is an integer work array of size 64.

Call Statement and Input:

```
            JOBU  JOBVT  M   N    A  LDA  S   U  LDU  VT  LDVT  WORK LWORK INFO
              |     |    |   |    |   |   |   |   |    |    |     |    |     |
CALL DGESVD( 'S' , 'O' , 2 , 4 ,  A , 2 , S , U , 2 , VT ,  1 , WORK , 0 , INFO )
```

–or–

```
            JOBZ   M   N    A  LDA  S   U  LDU  VT  LDVT  WORK  LWORK IWORK   INFO
              |    |   |    |   |   |   |   |    |    |     |     |     |       |
CALL DGESDD( 'S' , 2 , 4 ,  A , 2 , S , U , 2 , VT ,  2 , WORK ,  0 , IWORK , INFO )
```

```
 A    = |  1.0    2.0    3.0    4.0  |
        |  5.0    6.0    7.0    8.0  |
```

Output:

- For DGESVD:

```
A    =  | -0.352062 -0.443626 -0.535190 -0.626754 |
        |  0.758981  0.321242 -0.116498 -0.554238 |
```

- For DGESDD, A has been overwritten on output.

```
S    =  | 14.227407 |
        |  1.257330 |
```

```
U    =  | -0.376168 -0.926551 |
        | -0.926551  0.376168 |
```

- For DGESVD, VT is not referenced.
- For DGESDD, VT is:

```
VT   =  | -0.352062 -0.443626 -0.535190 -0.626754 |
        |  0.758981  0.321242 -0.116498 -0.554238 |
```

```
INFO = 0
```

Example 4

This example shows how to find the singular values only of the complex general matrix ***A***.

Notes:

1. For ZGESVD, because *jobu* = 'N', argument *u* is not referenced.
2. For ZGESVD, because *jobvt* = 'N', argument *vt* is not referenced.
3. For ZGESDD, because *jobz* = 'N', arguments *u* and *vt* are not referenced.
4. Because *lwork* = 0, the subroutine dynamically allocates WORK.
5. *iwork* is an integer work array of size 64.
6. *rwork* is an integer work array of size 64.

Call Statement and Input:

```
             JOBU  JOBVT  M   N    A  LDA  S   U  LDU  VT  LDVT  WORK LWORK  RWORK   INFO
              |     |     |   |    |   |   |   |   |   |     |     |    |      |       |
CALL ZGESVD( 'N' , 'N' , 4 , 4 ,  A , 4 , S , U , 1 , VT ,  1 , WORK , 0 ,  RWORK , INFO )
```

–or–

```
             JOBZ   M   N    A  LDA  S   U  LDU  VT LDVT  WORK LWORK RWORK  IWORK   INFO
              |     |   |    |   |   |   |   |   |   |     |    |     |      |       |
CALL ZGESDD( 'N' , 4 , 4 ,  A , 4 , S , U , 1 , VT , 1 , WORK , 0 , RWORK , IWORK , INFO )
```

```
        |  (1.0, 1.0)  (1.0, 0.0)  (0.0, 0.0)  (0.0, 0.0)  |
A    =  |  (0.0, 0.0)  (2.0,-1.0)  (1.0, 0.0)  (0.0, 0.0)  |
        |  (0.0, 0.0)  (0.0, 0.0)  (3.0, 1.0)  (1.0, 0.0)  |
        |  (0.0, 0.0)  (0.0, 0.0)  (0.0, 0.0)  (4.0,-1.0)  |
```

Output:

Array A is overwritten.

```
        | 4.389511 |
S    =  | 3.276236 |
        | 2.346361 |
        | 1.221907 |


INFO = 0
```

Example 5

This example shows how to find the singular values of the complex general matrix ***A*** and its left and right singular vectors.

Note:

1. Because *lwork* = 0, the subroutine dynamically allocates WORK.
2. *iwork* is an integer work array of size 64.

3. *rwork* is an integer work array of size 60.

Call Statement and Input:

```
            JOBU  JOBVT  M   N    A  LDA  S   U  LDU  VT  LDVT  WORK  LWORK RWORK   INFO
              |     |    |   |    |   |   |   |   |    |    |     |     |     |       |
CALL ZGESVD( 'A' , 'A' , 3 , 3 ,  A , 3 , S , U , 3 , VT ,  3 , WORK ,  0 , RWORK , INFO )
```

–or–

```
            JOBZ   M   N    A  LDA  S   U  LDU  VT  LDVT  WORK  LWORK RWORK   IWORK  INFO
              |    |   |    |   |   |   |   |    |    |     |     |     |       |      |
CALL ZGESDD( 'A' , 3 , 3 ,  A , 3 , S , U , 3 , VT ,  3 , WORK ,  0 , RWORK , IWORK  INFO )
```

```
        |  (1.0, 1.0)  (2.0,-1.0)  (3.0, 0.0)  |
A    =  |  (2.0,-1.0)  (4.0, 1.0)  (5.0,-1.0)  |
        |  (3.0, 0.0)  (5.0,-1.0)  (6.0, 1.0)  |
```

Output:

Array A is overwritten.

```
        | 11.370686 |
S    =  |  2.386257 |
        |  1.006620 |
```

```
        | (-0.3265, 0.0409) ( 0.0558, 0.4814) ( 0.3504,-0.7308) |
U    =  | (-0.5822, 0.0725) (-0.0823,-0.7730) ( 0.1017,-0.2026) |
        | (-0.7396, 0.0233) ( 0.0036, 0.4009) (-0.2805, 0.4616) |
```

```
        | (-0.3290, 0.0000) (-0.5867,-0.0004) (-0.7367,-0.0688) |
VT   =  | ( 0.4846, 0.0000) (-0.7774,-0.0071) ( 0.3987, 0.0425) |
        | (-0.8105, 0.0000) (-0.2267,-0.0041) ( 0.5375, 0.0533) |


INFO = 0
```

Example 6

This example shows how to find the singular values of the complex general matrix ***A***. Additionally:

The first min(*m*, *n*) columns of its left singular vectors ***U*** are returned in array U.

The first min(*m*, *n*) rows of its right singular vectors $\boldsymbol{V}^H$ are returned in array A.

Notes:

1. Because *jobvt* = 'O', argument *vt* is not referenced.
2. Because *lwork* = 0, the subroutine dynamically allocates WORK.
3. *rwork* is a work array of size 60.

Call Statement and Input:

```
            JOBU  JOBVT  M   N    A  LDA  S   U  LDU  VT  LDVT  WORK LWORK RWORK   INFO
              |     |    |   |    |   |   |   |   |    |    |     |    |     |       |
CALL ZGESVD( 'S' , 'O' , 4 , 2 ,  A , 4 , S , U , 4 , VT ,  1 , WORK , 0 , RWORK , INFO )
```

```
        |  (1.0, 1.0)  (2.0, 0.0)  |
 A    = |  (3.0, 0.0)  (4.0,-1.0)  |
        |  (5.0, 1.0)  (6.0, 0.0)  |
        |  (7.0, 0.0)  (8.0,-1.0)  |
```

Output:

```
        | (-0.642481, 0.000000) (-0.754181, 0.135753) |
A    =  | (-0.766302, 0.000000) ( 0.632319,-0.113817) |
        |           .                     .           |
        |           .                     .           |
```

```
S    =  | 14.394066 |
        |  0.900474 |
```

```
        | (-0.149426,-0.063497) ( 0.553415,-0.598204) |
U    =  | (-0.352918, 0.014671) ( 0.382229,-0.196618) |
        | (-0.537547,-0.101222) (-0.041751,-0.092615) |
        | (-0.741039,-0.023054) (-0.212937, 0.308971) |
```

```
INFO = 0
```

Example 7

This example shows how to find the singular values of the complex general matrix ***A***. Additionally:

All the columns of its left singular vectors ***U*** are returned in array U.

The first *m* rows of its right singular vectors $\boldsymbol{V}^H$ are returned in array A.

Notes:

1. Because *jobvt* = 'O', argument *vt* is not referenced.
2. Because *lwork* = 0, the subroutine dynamically allocates WORK.
3. *iwork* is an integer work array of size 64.
4. *rwork* is a work array of size 60.

Call Statement and Input:

```
            JOBZ   M   N    A  LDA  S   U  LDU  VT  LDVT  WORK  LWORK RWORK   IWORK  INFO
              |    |   |    |   |   |   |   |    |    |     |     |     |       |     |
CALL ZGESDD( 'O' , 2 , 3 ,  A , 2 , S , U , 2 , VT ,  1 , WORK ,  0 , RWORK , IWORK  INFO )
```

```
 A    = |  (1.0, 1.0)  (2.0, 0.0)  (3.0, 0.0)  |
        |  (0.0, 0.0)  (5.0,-1.0)  (6.0, 0.0)  |
```

Output:

```
A    =  | (-0.0488,-0.0488) (-0.6265, 0.0670) (-0.7722,-0.0447) |
        | (0.65909, 0.6590) (-0.2388, 0.1973) ( 0.1352,-0.1316) |
```

```
S    =  | 8.666569 |
        | 1.374982 |
```

```
U    =  | (-0.4231, 0.0000) ( 0.9061, 0.0000) |
        | (-0.9038, 0.0646) (-0.4221, 0.0301) |
```

```
INFO = 0
```

SGEQRF, DGEQRF, CGEQRF, and ZGEQRF (General Matrix QR Factorization)

Purpose

This subroutine computes the ***QR*** factorization of a general matrix

A = ***QR***

where:

For SGEQRF and DGEQRF, ***Q*** is an orthogonal matrix.
For CGEQRF and ZGEQRF, ***Q*** is a unitary matrix.
For *m* ≥ *n*, ***R*** is an upper triangular matrix.
For *m* < *n*, ***R*** is an upper trapezoidal matrix.

Table 194: Data Types.

A, ***τ***, ***work***	**Subroutine**
Short-precision real	SGEQRF$^{\triangle}$
Long-precision real	DGEQRF$^{\triangle}$
Short-precision complex	CGEQRF$^{\triangle}$
Long-precision complex	ZGEQRF$^{\triangle}$
$^{\triangle}$LAPACK	

Syntax

Table 195: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SGEQRF | DGEQRF | CGEQRF | ZGEQRF (*m*, *n*, *a*, *lda*, *tau*, *work*, *lwork*, *info*)
C and C++	sgeqrf | dgeqrf | cgeqrf | zgeqrf (*m*, *n*, *a*, *lda*, *tau*, *work*, *lwork*, *info*);
LAPACKE	info = LAPACKE_sgeqrf | LAPACKE_dgeqrf | LAPACKE_cgeqrf | LAPACKE_zgeqrf (*matrix_layout*, *m*, *n*, *a*, *lda*, *tau*);

On Entry

matrix_layout
indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

m

is the number of rows in matrix ***A*** used in the computation.

Specified as: an integer; $m \ge 0$.

n

is the number of columns in matrix ***A*** used in the computation.

Specified as: an integer; $n \ge 0$.

a

is the *m* by *n* general matrix ***A*** whose ***QR*** factorization is to be computed.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 194 on page 807.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and $lda \ge m$.

tau

See On Return.

work

has the following meaning:

If *lwork* = 0, *work* is ignored.

If *lwork* ≠ 0, *work* is the work area used by this subroutine, where:

- If *lwork* ≠ -1, its size is (at least) of length *lwork*.
- If *lwork* = -1, its size is (at least) of length 1.

Specified as: an area of storage containing numbers of data type indicated in Table 194 on page 807.

lwork

is the number of elements in array `WORK`.

Specified as: an integer; where:

- If *lwork* = 0, these subroutines dynamically allocate the work area used by this subroutine. The work area is deallocated before control is returned to the calling program. This option is an extension to the LAPACK standard.
- If *lwork* = -1, these subroutines perform a work area query and return the optimal size of *work* in $work_1$. No computation is performed and the subroutine returns after error checking is complete.
- Otherwise, it must be:

  $lwork \ge \max(1, n)$

info

See On Return.

On Return

a

is the updated general matrix ***A***, containing the results of the computation.

The elements on and above the diagonal of the array contain the min(*m*, *n*) × *n* upper trapezoidal matrix ***R*** (***R*** is upper triangular if $m \ge n$). The elements below the diagonal with τ represent the matrix ***Q*** as a product of min(*m*, *n*) elementary reflectors.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 194 on page 807.

tau
is the vector $\boldsymbol{\tau}$, of length min(*m*, *n*), containing the scalar factors of the elementary reflectors.

Returned as: a one-dimensional array of (at least) length min(*m*, *n*), containing numbers of the data type indicated in Table 194 on page 807.

work
is the work area used by this subroutine if $lwork \neq 0$, where:

If $lwork \neq 0$ and $lwork \neq -1$, its size is (at least) of length *lwork*.

If $lwork = -1$, its size is (at least) of length 1.

Returned as: an area of storage, where:

If $lwork \geq 1$ or $lwork = -1$, then $work_1$ is set to the optimal *lwork* value and contains numbers of the data type indicated in Table 194 on page 807. Except for $work_1$, the contents of *work* are overwritten on return.

info
indicates that a successful computation occurred.

Returned as:

- For SGEQRF, DGEQRF, CGEQRF, and ZGEQRF, returned as: an integer; info ≥ 0.
- For LAPACKE_sgeqrf, LAPACKE_dgeqrf, LAPACKE_cgeqrf, and LAPACKE_zgeqrf, returned as an integer function value; info ≥ 0.

Notes and Coding Rules

1. In your C program, argument *info* must be passed by reference.
2. The vectors and matrices used in the computation must have no common elements; otherwise, results are unpredictable.
3. For best performance, specify *lwork* = 0.

Function

Compute the ***QR*** factorization of a general matrix ***A***

$\boldsymbol{A} = \boldsymbol{QR}$

where:

The matrix ***Q*** is represented as a product of elementary reflectors:
$\boldsymbol{Q} = \boldsymbol{H}_1 \boldsymbol{H}_2 \ldots \boldsymbol{H}_k$

where:

$k = \min(m, n)$
For each i:
For SGEQRF and DGEQRF, $\boldsymbol{H}_i = \boldsymbol{I} - \tau \boldsymbol{v}\boldsymbol{v}^T$
For CGEQRF and ZGEQRF, $\boldsymbol{H}_i = \boldsymbol{I} - \tau \boldsymbol{v}\boldsymbol{v}^H$
$\boldsymbol{\tau}$ is a scalar, stored on return in $\boldsymbol{\tau}_i$
$\boldsymbol{v}$ is a real vector with $\boldsymbol{v}_{1:i-1}$ = zero, $\boldsymbol{v}_i$ = one.
$\boldsymbol{v}_{i+1:m}$ is stored on return in $\boldsymbol{A}_{i+1:m,\, i}$
$\boldsymbol{I}$ is the identity matrix

For $m \geq n$, ***R*** is an upper triangular matrix.
For $m < n$, ***R*** is an upper trapezoidal matrix.

If $m = 0$ or $n = 0$, no computation is performed and the subroutine returns after doing some parameter checking.

See references [“63” on page 1256, “9” on page 1253, “78” on page 1257, “61” on page 1256, “62” on page 1256].

Error conditions

Resource Errors
lwork = 0 and unable to allocate work space.

Computational Errors
None.

Input-Argument Errors

1. $m < 0$
2. $n < 0$
3. $lda \leq 0$
4. $lda < m$
5. $lwork \neq 0$, $lwork \neq -1$, and $lwork < \max(1, n)$

Examples

Example 1

This example shows the ***QR*** factorization of a general matrix ***A*** of size 6 × 2.

Note: Because *lwork* = 0, DGEQRF dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
              M    N    A  LDA  TAU   WORK  LWORK  INFO
              |    |    |   |    |     |      |      |
CALL DGEQRF ( 6 ,  2  , A , 6 , TAU , WORK ,  0   , INFO)
```

General matrix ***A*** of size 6 × 2:

```
         ┌                    ┐
         |   .000000  2.000000 |
         |  2.000000 -1.000000 |
A    =   |  2.000000 -1.000000 |
         |   .000000  1.500000 |
         |  2.000000 -1.000000 |
         |  2.000000 -1.000000 |
         └                    ┘
```

Output:

General matrix ***A*** of size 6 × 2.

```
         ┌                    ┐
         | -4.000000  2.000000 |
         |   .500000  2.500000 |
A    =   |   .500000   .285714 |
         |   .000000  -.428571 |
         |   .500000   .285714 |
         |   .500000   .285714 |
         └                    ┘
```

Vector ***τ*** of length 2:

```
           ┌                  ┐
TAU   =    |  1.000000  1.400000 |
                              ┘
INFO =  0
```

Example 2

This example shows the ***QR*** factorization of a general matrix ***A*** of size 4x5.

Note: Because *lwork* = 0, DGEQRF dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
              M    N    A  LDA  TAU   WORK  LWORK  INFO
              |    |    |   |    |     |      |      |
CALL DGEQRF ( 4 ,  5  , A , 4 , TAU , WORK ,  0  , INFO)
```

General matrix ***A*** of size 4 × 5:

```
         ┌                                                   ┐
         |   .500000   .500000  1.207107   .000000  1.707107 |
A    =   |   .500000 -1.500000  -.500000  2.414214   .707107 |
         |   .500000   .500000   .207107   .000000   .292893 |
         |   .500000 -1.500000  -.500000  -.414214  -.707107 |
                                                           ┘
```

Output:

General matrix ***A*** of size 4 × 5:

```
         ┌                                                    ┐
         | -1.000000  1.000000  -.207107 -1.000000  -1.000000 |
A    =   |   .333333  2.000000  1.207107 -1.000000   1.000000 |
         |   .333333  -.200000   .707107   .000000   1.000000 |
         |   .333333   .400000   .071068 -2.000000  -1.000000 |
                                                            ┘
```

Vector ***τ*** of length 4:

```
           ┌                                      ┐
TAU   =    |  1.500000  1.666667  1.989949   .000000 |
                                                  ┘
INFO =  0
```

Example 3

This example shows the ***QR*** factorization of a general matrix ***A*** of size 6 × 2.

Note: Because *lwork* = 0, ZGEQRF dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
              M    N    A  LDA  TAU   WORK  LWORK  INFO
              |    |    |   |    |     |      |      |
CALL ZGEQRF ( 6 ,  2  , A , 6 , TAU , WORK ,  0  , INFO)
```

General matrix ***A*** of size 6 × 2:

```
         ┌                                                 ┐
         |  (-1.800000, -0.900000) ( 1.100000, -0.800000) |
         |  (-1.600000,  1.000000) ( 1.700000,  1.400000) |
A    =   |  (-1.000000, -0.300000) ( 1.200000,  0.300000) |
         |  ( 1.100000, -0.100000) ( 0.700000, -1.900000) |
         |  ( 0.500000,  0.700000) (-0.200000, -1.500000) |
         |  (-1.500000, -0.700000) ( 1.800000, -0.600000) |
                                                        ┘
```

Output:

General matrix ***A*** of size 6 × 2:

```
        ┌                                            ┐
        |  ( 3.660601,  0.000000) (-1.731956, -0.524504) |
        |  ( 0.255874, -0.225302) (-3.865905,  0.000000) |
 A   =  |  ( 0.187102,  0.024101) ( 0.135165, -0.000348) |
        |  (-0.193177,  0.050152) ( 0.057186, -0.451900) |
        |  (-0.109713, -0.110108) (-0.065903, -0.220144) |
        |  ( 0.288000,  0.080724) ( 0.110147, -0.200172) |
        └                                            ┘
```

Vector ***τ*** of length 2:

```
        ┌                                             ┐
TAU  =  |  ( 1.491723,  0.245861) ( 1.268358,  0.545419) |
                                                      ┘

INFO =  0
```

Example 4

This example shows the ***QR*** factorization of a general matrix ***A*** of size 3 × 4.

Note: Because *lwork* = 0, ZGEQRF dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
              M    N    A  LDA  TAU   WORK  LWORK  INFO
              |    |    |   |    |     |      |      |
CALL ZGEQRF ( 3 ,  4  , A , 3 , TAU , WORK ,  0   , INFO)
```

General matrix ***A*** of size 3 × 4:

```
        ┌                                                       ┐
        |  (-1.60,  0.10) ( 0.30,  1.70) ( 0.30,  0.20) (-0.50, -1.80) |
        |  (-1.20,  0.00) (-0.90, -0.50) ( 1.50,  0.80) ( 1.50, -1.10) |
A    =  |  (-0.10,  1.30) (-1.10,  0.50) ( 0.40, -1.30) ( 1.60,  0.70) |
                                                              ┘
```

Output:

General matrix ***A*** of size 3 × 4:

```
        ┌                                                       ┐
        |  ( 2.39,  0.00) ( 0.64, -0.32) (-1.67, -0.71) (-0.18,  0.88) |
 A   =  |  ( 0.30,  0.01) ( 2.23,  0.00) (-0.70,  0.81) (-2.25, -0.01) |
        |  ( 0.03, -0.33) ( 0.48, -0.28) ( 0.65,  0.00) (-1.00, -1.77) |
                                                              ┘
```

Vector ***τ*** of length 3:

```
        ┌                                        ┐
TAU  =  |  ( 1.67, -0.04) ( 1.35,  0.49) ( 1.99,  0.14) |
                                                 ┘

INFO =  0
```

SGELS, DGELS, CGELS, and ZGELS (Linear Least Squares Solution for a General Matrix)

Purpose

SGELS and DGELS compute the linear least squares solution for a real general matrix ***A*** or its transpose using a ***QR*** factorization without column pivoting, where ***A*** is assumed to have full rank.

CGELS and ZGELS compute the linear least squares solution for a complex general matrix ***A*** or its conjugate transpose using a ***QR*** factorization without column pivoting, where ***A*** is assumed to have full rank.

The following options are provided:

- If *transa* = 'N' and $m \geq n$: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\|\boldsymbol{B} - \boldsymbol{AX}\|$
- If *transa* = 'N' and $m < n$: find the minimum norm solution of an underdetermined system; that is, the problem is: $\boldsymbol{AX} = \boldsymbol{B}$
- For SGELS and DGELS:
  - If *transa* = 'T' and $m \geq n$: find the minimum norm solution of an underdetermined system; that is, the problem is $\boldsymbol{A}^T\boldsymbol{X} = \boldsymbol{B}$
  - If *transa* = 'T' and $m < n$: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\|\boldsymbol{B} - \boldsymbol{A}^T\boldsymbol{X}\|$
- For CGELS and ZGELS:
  - If *transa* = 'C' and $m \geq n$: find the minimum norm solution of an underdetermined system; that is, the problem is $\boldsymbol{A}^H\boldsymbol{X} = \boldsymbol{B}$
  - If *transa* = 'C' and $m < n$: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\|\boldsymbol{B} - \boldsymbol{A}^H\boldsymbol{X}\|$

Table 196: Data Types.

A*, *B*, *work	**Subroutine**
Short-precision real	SGELS^
Long-precision real	DGELS^
Short-precision complex	CGELS^
Long-precision complex	ZGELS^
^LAPACK	

Syntax

Table 197: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SGELS | DGELS | CGELS | ZGELS (*transa*, *m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *work*, *lwork*, *info*)
C and C++	sgels | dgels | cgels | zgels (*transa*, *m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *work*, *lwork*, *info*);
LAPACKE	info = LAPACKE_sgels | LAPACKE_dgels | LAPACKE_cgels | LAPACKE_zgels (*matrix_layout*, *transa*, *m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

transa

indicate the form of matrix $\boldsymbol{A}$ to use in the computation, where:

If *transa* = 'N', matrix $\boldsymbol{A}$ is used.

If *transa* = 'T', matrix $\boldsymbol{A}^T$ is used.

If *transa* = 'C', matrix $\boldsymbol{A}^H$ is used.

Specified as: a single character, where:

- For SGELS and DGELS, it must be 'N' or 'T'.
- For CGELS and ZGELS, it must be 'N' or 'C'.

m
is the number of rows in matrix ***A*** used in the computation.

Specified as: an integer; $m \geq 0$.

n
is the number of columns in matrix ***A*** used in the computation.

Specified as: an integer; $n \geq 0$.

nrhs
is the number of right-hand sides; that is, the number of columns in matrix ***B*** used in the computation.

Specified as: an integer; $nrhs \geq 0$.

a
is the *m* by *n* coefficient matrix ***A***.

Note: No data should be moved to form $\boldsymbol{A}^T$ or $\boldsymbol{A}^H$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 196 on page 813.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer; $lda > 0$ and $lda \geq m$.

b
is the matrix ***B*** of right-hand side vectors.

If *transa* = 'N', matrix ***B*** has *m* rows and *nrhs* columns.

For DGELS and SGELS, if *transa* = 'T', matrix ***B*** has *n* rows and *nrhs* columns.

For CGELS and ZGELS, if *transa* = 'C', matrix ***B*** has *n* rows and *nrhs* columns.

Specified as: the *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 196 on page 813.

ldb
is the leading dimension of the array specified for *b*.

Specified as: an integer; $ldb > 0$ and $ldb \geq \max(m,n)$.

work
has the following meaning:

If *lwork* = 0, *work* is ignored.

If $lwork \neq 0$, *work* is the work area used by this subroutine, where:

- If $lwork \neq -1$, its size is (at least) of length *lwork*.
- If *lwork* = -1, its size is (at least) of length 1.

Specified as: an area of storage containing numbers of data type indicated in Table 196 on page 813.

lwork
is the number of elements in array *work*.

Specified as: an integer; where:

- If *lwork* = 0, these subroutines dynamically allocate the work area used by this subroutine. The work area is deallocated before control is returned to the calling program. This option is an extension to the LAPACK standard.
- If *lwork* = -1, these subroutines perform a work area query and return the optimal size of *work* in $work_1$. No computation is performed, and the subroutine returns after error checking is complete.
- Otherwise, it must be:

  $lwork \geq \max(1, mn + \max(mn, nrhs))$

  where $mn = \min(m, n)$.

info

See On Return.

On Return

a

is the updated general matrix ***A***. The matrix ***A*** is overwritten; that is, the original input is not preserved.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 196 on page 813.

b

is the updated general matrix ***B***, containing the results of the computation. ***B*** is overwritten by the solution vectors, stored columnwise:

- If *transa* = 'N' and $m \geq n$, rows 1 to *n* of ***B*** contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements n+1 to *m* in that column.
- If *transa* = 'N' and $m < n$, rows 1 to *n* of ***B*** contain the minimum norm solution vectors.
- For SGELS and DGELS:
  - If *transa* = 'T' and $m \geq n$, rows 1 to *m* of ***B*** contain the minimum norm solution vectors.
  - If *transa* = 'T' 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.
- For CGELS and ZGELS:
  - If *transa* = 'C' and $m \geq n$, rows 1 to *m* of ***B*** contain the minimum norm solution vectors.
  - If *transa* = '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.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 196 on page 813.

work

is the work area used by this subroutine if $lwork \neq 0$, where:

If $lwork \neq 0$ and $lwork \neq -1$, its size is (at least) of length *lwork*.

If *lwork* = -1, its size is (at least) of length 1.

Returned as: an area of storage, where:

If $lwork \geq 1$ or *lwork* = -1, then $work_1$ is set to the optimal *lwork* value and contains numbers of the data type indicated in Table 196 on page 813.

Except for $work_1$, the contents of *work* are overwritten on return.

info

indicates that a successful computation occurred.

Returned as:

- For SGELS, DGELS, CGELS, and ZGELS, returned as: an integer; info ≥ 0.
- For LAPACKE_sgels, LAPACKE_dgels, LAPACKE_cgels, and LAPACKE_zgels, returned as an integer function value; info ≥ 0.

Notes and Coding Rules

1. In your C program, argument *info* must be passed by reference.
2. All subroutines accept lowercase letters for the *transa* argument.
3. The vectors and matrices used in the computation must have no common elements; otherwise, results are unpredictable.
4. For best performance, specify *lwork* = 0.

Function

SGELS and DGELS compute the linear least squares solution for a real general matrix $\boldsymbol{A}$ or its transpose using a $\boldsymbol{QR}$ factorization without column pivoting, where $\boldsymbol{A}$ is assumed to have full rank.

CGELS and ZGELS compute the linear least squares solution for a complex general matrix $\boldsymbol{A}$ or its conjugate transpose using a $\boldsymbol{QR}$ factorization without column pivoting, where $\boldsymbol{A}$ is assumed to have full rank.

The following options are provided:

- If *transa* = 'N' and $m \geq n$: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\|\boldsymbol{B} - \boldsymbol{AX}\|$
- If *transa* = 'N' and $m < n$: find the minimum norm solution of an underdetermined system; that is, the problem is: $\boldsymbol{AX} = \boldsymbol{B}$
- For SGELS and DGELS:
  - If *transa* = 'T' and $m \geq n$: find the minimum norm solution of an underdetermined system; that is, the problem is $\boldsymbol{A}^T\boldsymbol{X} = \boldsymbol{B}$
  - If *transa* = 'T' and $m < n$: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\|\boldsymbol{B} - \boldsymbol{A}^T\boldsymbol{X}\|$
- For CGELS and ZGELS:
  - If *transa* = 'C' and $m \geq n$: find the minimum norm solution of an underdetermined system; that is, the problem is $\boldsymbol{A}^H\boldsymbol{X} = \boldsymbol{B}$
  - If *transa* = 'C' and $m < n$: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\|\boldsymbol{B} - \boldsymbol{A}^H\boldsymbol{X}\|$

If (m = 0 and n = 0) or *nrhs* = 0, then no computation is performed and the subroutine returns after doing some parameter checking.

See reference [“75” on page 1257].

Error conditions

Resource Errors
lwork = 0 and unable to allocate work space.

Computational Errors
None.

Input-Argument Errors

1. For SGELS and DGELS, *transa* ≠ 'N' or 'T'

For CGELS and ZGELS, *transa* ≠ 'N' or 'C'

2. $m < 0$
3. $n < 0$
4. *nrhs* < 0
5. *lda* < *m*
6. *lda* ≤ 0
7. *ldb* < max(*m*, *n*)
8. *ldb* ≤ 0
9. *lwork* ≠ 0, *lwork* ≠ -1, and *lwork* < max(1, *mn* + max(*mn*, *nrhs*)) where *mn* = min(*m*, *n*)

Examples

Example 1

This example finds the least squares solution of an overdetermined real general system; that is, it solves the least squares problem: minimize ‖***B-AX***‖. Matrix ***A*** is size 6 × 2 and matrix ***B*** is size 6 × 3.

Note: Because *lwork* = 0, DGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
          TRANSA   M   N   NRHS   A   LDA  B  LDB  WORK  LWORK  INFO
             |     |   |     |    |    |   |   |     |     |      |
CALL DGELS ( 'N' ,  6 , 2 ,   3 ,  A ,  6 , B,  6,  WORK,   0,   INFO )
```

General matrix ***A*** of size 6 × 2:

```
        ┌                        ┐
        |   .000000000  2.000000000 |
        |  2.000000000 -1.000000000 |
A    =  |  2.000000000 -1.000000000 |
        |   .000000000  1.500000000 |
        |  2.000000000 -1.000000000 |
        |  2.000000000 -1.000000000 |
                                   ┘
```

General matrix ***B*** of size 6 × 3:

```
        ┌                                    ┐
        |  1.000000000  4.000000000  1.000000000 |
        |  1.000000000  1.000000000  2.000000000 |
B    =  |  1.000000000 -1.000000000  1.000000000 |
        |  1.000000000  3.000000000  2.000000000 |
        |  1.000000000  1.000000000  1.000000000 |
        |  1.000000000 -1.000000000  1.000000000 |
                                              ┘
```

Output:

General matrix ***A*** is overwritten.

Solution matrix ***X*** overwrites ***B***:

```
                                              ┐
        |   .780000000  1.000000000  1.025000000 |
        |   .560000000  2.000000000   .800000000 |
B    =  |   .042857143 -1.285714286  -.250000000 |
        |   .185714286   .428571429  1.250000000 |
        |   .042857143   .714285714  -.250000000 |
        |   .042857143 -1.285714286  -.250000000 |
                                              ┘
```

```
INFO  =  0
```

Example 2

This example finds the minimum norm solution of an underdetermined real general system $\boldsymbol{A}^T\boldsymbol{X} = \boldsymbol{B}$. Matrix ***A*** is size 6 × 2. On input, matrix ***B*** is size 2 × 1, stored in array *b* with leading dimension 6.

Note: Because *lwork* = 0, DGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
          TRANSA   M   N   NRHS   A   LDA  B  LDB  WORK  LWORK  INFO
             |     |   |     |    |    |   |   |     |     |      |
CALL DGELS ( 'T' ,  6 , 2 ,   1 ,  A ,  6 , B,  6,  WORK,   0,   INFO )
```

General matrix ***A*** of size 6 × 2:

```
        ┌                          ┐
        |   .000000000  2.000000000 |
        |  2.000000000 -1.000000000 |
A    =  |  2.000000000 -1.000000000 |
        |   .000000000  1.500000000 |
        |  2.000000000 -1.000000000 |
        |  2.000000000 -1.000000000 |
                                   ┘
```

General matrix ***B*** of size 2 × 1:

```
        ┌             ┐
B    =  |  1.000000000 |
        |  1.000000000 |
        |       .      |
        |       .      |
        |       .      |
        |       .      |
                      ┘
```

Output:

General matrix ***A*** is overwritten.

Solution matrix ***X*** overwrites ***B***:

```
                      ┐
        |   .480000000 |
        |   .125000000 |
B    =  |   .125000000 |
        |   .360000000 |
        |   .125000000 |
        |   .125000000 |
                      ┘
```

```
INFO  =  0
```

Example 3

This example finds the minimum norm solution of an underdetermined real general system ***AX*** = ***B***. Matrix ***A*** is size 3 × 4. On input, matrix ***B*** is size 3 × 4, stored in array *b* with leading dimension 4.

Note: Because *lwork* = 0, DGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
          TRANSA   M   N   NRHS   A   LDA  B  LDB  WORK  LWORK  INFO
             |     |   |     |    |    |   |   |     |     |      |
CALL DGELS ( 'N' ,  3 , 4 ,   4 ,  A ,  3 , B,  4,  WORK,   0,   INFO )
```

General matrix ***A*** of size 3 × 4:

```
        ┌                                                      ┐
        |   .500000000   .500000000    .500000000   .500000000 |
A    =  |   .500000000 -1.500000000    .500000000 -1.500000000 |
        |  1.000000000  1.000000000    .000000000  1.000000000 |
                                                               ┘
```

General matrix ***B*** of size 3 × 4:

```
        ┌                                                  ┐
        |  1.000000000  1.000000000   1.000000000   .000000000 |
B    =  |  1.000000000 -1.000000000   2.500000000  1.000000000 |
        |  1.000000000  1.000000000   3.000000000   .000000000 |
        |       .            .             .            .      |
        └                                                  ┘
```

Output:

General matrix ***A*** is overwritten.

Solution matrix ***X*** overwrites ***B***:

```
                                                           ┐
        |  1.000000000   .000000000   3.500000000   .500000000 |
B    =  |   .000000000   .500000000   -.250000000  -.250000000 |
        |  1.000000000  1.000000000  -1.000000000   .000000000 |
        |   .000000000   .500000000   -.250000000  -.250000000 |
                                                           ┘
```

```
INFO  =  0
```

Example 4

This example finds the least squares solution of an overdetermined real general system; that is, it solves the least squares problem: minimize ‖***B***-$\boldsymbol{A}^T$***X***‖. Matrix ***A*** is size 3 × 4. On input, matrix ***B*** is size 4 × 4.

Note: Because *lwork* = 0, DGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
           TRANSA  M   N   NRHS  A  LDA  B  LDB  WORK  LWORK  INFO
              |    |   |    |    |   |   |   |    |      |     |
CALL DGELS ( 'T' , 3 , 4 ,  4 ,  A , 3 , B , 4 , WORK ,  0 ,  INFO )
```

General matrix ***A*** of size 3 × 4:

```
        ┌                                                  ┐
        |   .500000000   .500000000    .500000000   .500000000 |
A    =  |   .500000000 -1.500000000    .500000000 -1.500000000 |
        |  1.207106781  -.500000000    .207106781  -.500000000 |
        └                                                  ┘
```

General matrix ***B*** of size 4 × 4:

```
        ┌                                                  ┐
        |  1.000000000  1.000000000   1.000000000   .000000000 |
B    =  |  1.000000000 -1.000000000   2.000000000  2.414213562 |
        |  1.000000000  1.000000000   3.000000000   .000000000 |
        |  1.000000000 -1.000000000   4.000000000  -.414213562 |
        └                                                  ┘
```

Output:

General matrix ***A*** is overwritten.

Solution matrix ***X*** overwrites ***B***:

```
                                                           ┐
        |  2.000000000  1.000000000   6.121320344   .500000000 |
B    =  |   .000000000  1.000000000    .707106781  -.500000000 |
        |   .000000000   .000000000  -2.000000000   .000000000 |
        |   .000000000   .000000000   1.414213562 -2.000000000 |
                                                           ┘
```

```
INFO  =  0
```

Example 5

This example finds the minimum norm solution of an underdetermined complex general system ***AX* = *B***. Matrix ***A*** is size 3 × 4. Matrix ***B*** is size 3 × 3, stored in array *b* with leading dimension 4.

Note: Because *lwork* = 0, ZGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
           TRANSA  M   N   NRHS  A  LDA  B  LDB  WORK  LWORK  INFO
              |    |   |    |    |   |   |   |    |      |     |
CALL ZGELS ( 'N' , 3 , 4 ,  3 ,  A , 3 , B , 4 , WORK ,  0 ,  INFO )
```

General matrix ***A*** of size 3 × 4:

```
         ┌                                                      ┐
         | (1.00,  0.00) (-2.00, 1.00) (-3.00, -1.00) ( 4.00, -3.00) |
A    =   | (1.00, -1.00) ( 2.00, 2.00) (-3.00,  0.00) (-4.00, -2.00) |
         | (1.00, -2.00) (-2.00, 3.00) (-3.00,  1.00) ( 4.00, -1.00) |
         └                                                      ┘
```

General matrix ***B*** of size 3 × 3:

```
         ┌                                          ┐
         | ( 1.00,  0.00) ( 0.00,  1.00) ( 1.00,  1.00) |
 B   =   | (-1.00,  1.00) ( 1.00, -1.00) ( 0.00,  0.00) |
         | ( 2.00,  1.00) ( 1.00,  2.00) (-1.00, -1.00) |
         |          .              .              .     |
         └                                          ┘
```

Output:

General matrix ***A*** is overwritten.

Solution matrix ***X*** overwrites ***B***:

```
                                                    ┐
         | (-0.16,  0.15) (-0.08,  0.18) ( 0.16, -0.31) |
 B   =   | ( 0.11,  0.02) ( 0.21, -0.50) (-0.38,  0.65) |
         | (-0.13, -0.32) ( 0.16,  0.12) (-0.27, -0.28) |
         | ( 0.37, -0.05) ( 0.04,  0.06) (-0.19,  0.33) |
                                                    ┘
```

```
INFO  =  0
```

Example 6

This example finds the least squares solution of an overdetermined complex general system A; that is, it solves the least squares problem: minimize ‖***B*-*AX***‖. Matrix ***A*** is size 6 × 2. Matrix ***B*** is size 6 × 1.

Note: Because *lwork* = 0, ZGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
           TRANSA  M   N   NRHS  A  LDA  B  LDB  WORK  LWORK  INFO
              |    |   |    |    |   |   |   |    |      |     |
CALL ZGELS ( 'N' , 6 , 2 ,  1 ,  A , 6 , B , 6 , WORK ,  0 ,  INFO )
```

Matrix ***A*** is the same used as input in Example 3 for ZGEQRF.

General matrix ***B*** of 6 × 1:

```
         ┌             ┐
         | ( 6.0, 0.0) |
B    =   | ( 5.0, 0.0) |
         | ( 4.0, 0.0) |
         | ( 3.0, 0.0) |
         | ( 2.0, 0.0) |
         | ( 1.0, 0.0) |
         └             ┘
```

Output:

General matrix ***A*** is overwritten.

Solution matrix ***X*** overwrites ***B***:

```
                                  ┐
B    =  | (-1.135350,  0.520298) |
        | ( 0.944064,  0.624509) |
        | ( 1.062824, -0.899701) |
        | ( 2.570856,  1.687827) |
        | ( 2.556854,  2.835820) |
        | (-3.982815, -0.231572) |
                                  ┘
```

```
INFO  =  0
```

Example 7

This example finds the minimum norm solution of an underdetermined complex general system $\boldsymbol{A^{H}X} = \boldsymbol{B}$. Matrix ***A*** is size 3 × 3. Matrix ***B*** is size 3 × 2.

Note: Because *lwork* = 0, ZGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
           TRANSA  M   N   NRHS  A  LDA  B  LDB  WORK  LWORK  INFO
              |    |   |    |    |   |   |   |    |      |     |
CALL ZGELS ( 'C' , 3 , 3 ,  2 ,  A , 3 , B , 3 , WORK ,  0 ,  INFO )
```

Matrix ***A*** is the same used as input in Example 4 for CPOSV.

Matrix ***B*** is the same used as input in Example 4 for CPOSV.

Output:

General matrix ***A*** is overwritten.

Solution matrix ***X*** overwrites ***B***:

```
                                    ┐
B    =  | ( 2.0, -1.0) (2.0,  0.0) |
        | ( 1.0, -1.0) (0.0,  1.0) |
        | ( 3.0,  0.0) (1.0, -1.0) |
                                    ┘
```

```
INFO  =  0
```

Example 8

This example finds the least squares solution of an overdetermined complex general system; that is, it solves the least squares problem: minimize $\|\boldsymbol{B}\text{-}\boldsymbol{A^{H}X}\|$. Matrix ***A*** is size 2 × 6. Matrix ***B*** is size 6 × 1.

Note: Because *lwork* = 0, ZGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
           TRANSA  M   N   NRHS  A  LDA  B  LDB  WORK  LWORK  INFO
              |    |   |    |    |   |   |   |    |      |     |
CALL ZGELS ( 'C' , 2 , 6 ,  1 ,  A , 2 , B , 6 , WORK ,  0 ,  INFO )
```

General matrix ***A*** of size 2 × 6:

```
        ┌                                                                    ┐
A    =  | (2.0, 0.0) (6.0, 0.0) (10.0, 0.0) (14.0, 0.0) (18.0, 0.0) (22.0, 0.0) |
        | (4.0, 0.0) (8.0, 0.0) (12.0, 0.0) (16.0, 0.0) (20.0, 0.0) (24.0, 0.0) |
                                                                             ┘
```

General matrix ***B*** of size 6 × 1:

```
         ┌              ┐
         | ( 6.0, 0.0)  |
B    =   | ( 5.0, 0.0)  |
         | ( 4.0, 0.0)  |
         | ( 3.0, 0.0)  |
         | ( 2.0, 0.0)  |
         | ( 1.0, 0.0)  |
         └              ┘
```

Output:

General matrix ***A*** is overwritten.

Solution matrix ***X*** overwrites ***B***:

```
                          ┐
         | (-3.50, 0.00)  |
B    =   | ( 3.25, 0.00)  |
         | ( 0.00, 0.00)  |
         | ( 0.00, 0.00)  |
         | ( 0.00, 0.00)  |
         | ( 0.00, 0.00)  |
                          ┘
```

```
INFO  =  0
```

SGELSD, DGELSD, CGELSD, and ZGELSD (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)

Purpose

These subroutines compute the linear least squares solution for a general matrix ***A*** using the singular value decomposition.

The following options are provided:

- If $m \geq n$: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\|\boldsymbol{B} - \boldsymbol{AX}\|$
- If $m < n$: find the minimum norm solution of an undetermined system; that is, the problem is: ***AX***=***B***

Table 198: Data Types.

A*, *B*, *work	***s*, *rcond*, *rwork***	**Subroutine**
Short-precision real	Short-precision real	SGELSD$^{\Delta}$
Long-precision real	Long-precision real	DGELSD$^{\Delta}$
Short-precision complex	Short-precision real	CGELSD$^{\Delta}$
Long-precision complex	Long-precision real	ZGELSD$^{\Delta}$
$^{\Delta}$LAPACK		

Syntax

Table 199: Syntax for supported computing languages.

Language	Syntax
Fortran	CALL SGELSD | DGELSD (*m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *s*, *rcond*, *rank*, *work*, *lwork*, *iwork*, *info*) CALL CGELSD | ZGELSD (*m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *s*, *rcond*, *rank*, *work*, *lwork*, *rwork*, *iwork*, *info*)

C and C++	sgelsd | dgelsd (*m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *s*, *rcond*, *rank*, *work*, *lwork*, *iwork*, *info*);
	 cgelsd | zgelsd (*m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *s*, *rcond*, *rank*, *work*, *lwork*, *rwork*, *iwork*, *info*);

LAPACKE	info = LAPACKE_sgelsd | LAPACKE_dgelsd (*matrix_layout*, *m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *s*, *rcond*, *rank*);
	 info = LAPACKE_cgelsd | LAPACKE_zgelsd (*matrix_layout*, *m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *s*, *rcond*, *rank*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

m

is the number of rows in matrix ***A*** and ***B***.

Specified as: an integer; $m \geq 0$.

n

is the number of columns in matrix ***A***.

Specified as: an integer; $n \geq 0$.

nrhs

is the number of right-hand sides; that is, the number of columns in matrix ***B***.

Specified as: an integer; $nrhs \geq 0$.

a

is the *m* by *n* general matrix ***A***.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 198 on page 822.

lda

is the leading dimension of the array specified for ***A***.

Specified as: an integer; $lda > 0$ and $lda \geq m$.

b

is the general matrix ***B*** containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *m*, reside in the columns of matrix ***B***.

Specified as: the *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 198 on page 822.

ldb

is the leading dimension of the array specified for *b*.

Specified as: an integer; $ldb \geq \max(m,n)$.

rcond

is used to determine the effective rank of matrix ***A***. Singular values of $s_i \leq (rcond)(s_i)$ are treated as zero.

If *rcond* is less than or equal to zero or *rcond* is greater than or equal to one, then an *rcond* value of ε is used, where ε is the machine precision.

Specified as: a number of data type indicated in Table 198 on page 822.

work

is the work area used by this subroutine, where:

If *lwork*=0, *work* is ignored.

If *lwork* ≠ 0, the size of *work* is determined as follows:

- If *lwork* ≠ -1, *work* is (at least) of length *lwork*.
- If *lwork*=-1, *work* is (at least) of length 1.

Specified as: an area of storage containing numbers of data type indicated in Table 198 on page 822.

lwork

is the number of elements in array *work*.

Specified as: an integer; where:

- If *lwork*=0, these subroutines dynamically allocate the work area used by this subroutine. The work area is deallocated before control is returned to the calling program. This option is an extension to the LAPACK standard.
- If *lwork*=-1, these subroutines perform a work area query and return the optimal size of *work* in $work_1$. No computation is performed, and the subroutine returns after error checking is complete.
- Otherwise:

  For SGELSD and DGELSD

  $lwork \geq 12r + 2(r)(smlsiz) + 8(r)(nlvl) + (r)(nrhs) + (smlsiz+1)^2$, where:

  – $r = \min(m,n)$

  – $smlsiz = 25$

  – $nlvl = \max(0, \text{int}(\log_2(r/(smlsiz+1)))+1)$

  For CGELSD and ZGELSD

  $lwork \geq \max(1,m+n+r,2r + (r)(nrhs))$, where $r = \min(m, n)$.

Note: These formulas represent the minimum workspace required. For best performance, specify either *lwork* = -1 (to obtain the optimal size to use) or *lwork* = 0 (to direct the subroutine to dynamically allocate the workspace).

rwork

is a work area of size max(1, *lrwork*), where:

$lrwork \geq 10r + 2(r)(smlsiz) + 8(r)(nlvl) +2(smlsiz)(nrhs) + \max((smlsiz+1)^2, n(1 + nrhs) + 2nrhs)$, where:

- $r = \min(m, n)$
- $smlsiz = 25$
- $nlvl = \max(0, \text{int}(\log_2(r/(smlsiz+1))) + 1)$

Specified as: an area of storage containing numbers of data type indicated in Table 198 on page 822.

iwork

is a work area of size max(1, *liwork*), where:

$liwork \geq 3(r)(nlvl) + 11r$ where:

- $r = \min(m, n)$

- *smlsiz* = 25
- *nlvl* = max(0,int($\log_2$(*r*/(*smlsiz*+1))) + 1)

Specified as: an area of storage containing integers.

info
See On Return.

On Return

a
The matrix ***A*** is overwritten; that is, the original input is not preserved.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 198 on page 822.

b
is the updated general matrix ***B***, containing the results of the computation. ***B*** is overwritten by the *n* by *nrhs* solution matrix ***X***.

- If $m \geq n$ and *rank* = *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 $m < n$, rows 1 to *n* of ***B*** contain the minimum norm solution vectors.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 198 on page 822.

s
is the vector ***s*** containing the singular values of matrix ***A***.

Returned as: a one-dimensional array of (at least) length min(*m*,*n*) containing numbers of the data type indicated in Table 198 on page 822 .

rank
is the effective rank of ***A***; that is the number of singular values that are greater than *rcond*(s_1).

Returned as: an integer.

work
is the work area used by this subroutine if *lwork* ≠ 0, where:

If *lwork* ≠ 0 and *lwork* ≠ -1, its size is (at least) of length *lwork*.

If *lwork*=-1, its size is (at least) of length 1.

Returned as: an area of storage, where:

If *lwork* ≥ 1 or *lwork*=-1, then $work_1$ is set to the optimal *lwork* value and contains numbers of the data type indicated in Table 198 on page 822.

Except for $work_1$, the contents of *work* are overwritten on return.

rwork
is a work area used by these subroutines.

Returned as: an area of storage where, if *info* = 0, $rwork_1$ is set to the minimum size of *rwork*.

iwork
is a work area used by these subroutines.

Returned as: an area of storage where, if *info* = 0, $iwork_1$ is set to the minimum size of *iwork*.

info
has the following meaning:

If *info* = 0, the subroutine completed successfully.

If $0 < info \leq \max(m, n)$, *info* specifies how many superdiagonals of an intermediate bidiagonal form did not converge to zero.

If *info* = max(*m*, *n*) + 1, a singular value failed to converge.

SGELSD, DGELSD, CGELSD, and ZGELSD

Returned as:

- For SGELSD, DGELSD, CGELSD, and ZGELSD, returned as: an integer; info ≥ 0.
- For LAPACKE_sgelsd, LAPACKE_dgelsd, LAPACKE_cgelsd, and LAPACKE_zgelsd, returned as an integer function value; info ≥ 0.

Notes and Coding rules:

1. In your C program, arguments *rank* and *info* must be passed by reference.
2. *a*, *b*, *s*, *work*, *rwork* and *iwork* must have no common elements; otherwise, results are unpredictable.
3. For best performance, specify *lwork* = 0.

Function

These subroutines compute the linear least squares solution for a general matrix ***A*** using the singular value decomposition.

The following options are provided:

- If *m* ≥ *n*: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize ‖***B*** - ***AX***‖
- If *m* < *n*: find the minimum norm solution of an undetermined system; that is, the problem is: ***AX***=***B***

See reference [“35” on page 1255], [“75” on page 1257].

Error conditions

Resource Errors
lwork=0 and unable to allocate work space.

Computational Errors

- Superdiagonals of an intermediate bidiagonal form did not converge to zero.
- A singular value failed to converge.

Input-Argument Errors

1. *m* < 0
2. *n* < 0
3. *nrhs* < 0
4. *lda* ≤ 0
5. *lda* < *m*
6. *ldb* ≤ 0
7. *ldb* < max(*m*,*n*)
8. *lwork* ≠ 0 and *lwork* ≠ -1 and *lwork* < the minimum required value
9. *rwork* ≠ 0 and *rwork* ≠ -1 and *rwork* < the minimum required value
10. *iwork* ≠ 0 and *iwork* ≠ -1 and *iwork* < the minimum required value

Examples

Example 1

This example finds the least squares solution of an overdetermined real general system; that is, it solves the least squares problem: minimize ‖***B***-***AX***‖. Matrix ***A*** is size 6 × 2 and matrix ***B*** is size 6 × 3.

Notes:

- Because *lwork*=0, DGELSD dynamically allocates the work area used by this subroutine.
- *iwork* is an integer work array of size 22.

Call Statements and Input:

```
               M    N    NRHS    A    LDA  B    LDB  S    RCOND    RANK    WORK LWORK IWORK    INFO
               |    |     |      |     |   |     |   |      |        |       |    |     |        |
CALL DGELSD ( 6 , 2 ,    3 ,    A ,   6 , B ,   6 , S , RCOND , RANK , WORK , 0 , IWORK , INFO )
```

A = (same as input A in Example 1 for DGELS)
B = (same as input B in Example 1 for DGELS)

```
RCOND  = .745058D-08
```

Output:

General matrix ***A*** is overwritten.

Solution matrix ***X*** overwrites ***B***:

```
                                                 ┐
         |   0.780000000  1.000000000  1.025000000 |
         |   0.560000000  2.000000000  0.800000000 |
B    =   |   0.042857143 -1.285714286 -0.250000000 |
         |   0.185714286  0.428571429  1.250000000 |
         |   0.042857143  0.714285714 -0.250000000 |
         |   0.042857143 -1.285714286 -0.250000000 |
                                                 ┘
```

```
                       ┐
         |  4.650367627 |
S    =   |  2.150367627 |
                       ┘
```

```
RANK  =  2
```

```
INFO  =  0
```

Example 2

This example finds the minimum norm solution of an underdetermined real general system ***AX*** = ***B***. Matrix ***A*** is size 3 × 4. On input, matrix ***B*** is size 3 × 3, stored in array *b* with leading dimension 4.

Notes:

- Because *lwork*=0, DGELSD dynamically allocates the work area used by this subroutine.
- *iwork* is an integer work array of size 33.

Call Statements and Input:

```
               M    N    NRHS    A    LDA  B    LDB  S    RCOND    RANK    WORK LWORK IWORK    INFO
               |    |     |      |     |   |     |   |      |        |       |    |     |        |
CALL DGELSD ( 3 , 4 ,    4 ,    A ,   3 , B ,   4 , S , RCOND , RANK , WORK , 0 , IWORK , INFO )
```

A = (same as input A in Example 3 for DGELS)
B = (same as input B in Example 3 for DGELS)

```
RCOND  = -1
```

Output:

General matrix ***A*** is overwritten.

Solution matrix ***X*** overwrites ***B***:

```
                                                               ┐
         | 1.000000000  0.000000000  3.500000000  0.500000000 |
B    =   | 0.000000000  0.500000000 -0.250000000 -0.250000000 |
         | 1.000000000  1.000000000 -1.000000000  0.000000000 |
         | 0.000000000  0.500000000 -0.250000000 -0.250000000 |
                                                               ┘
```

```
                      ┐
         |  2.672011881 |
S    =   |  1.297080811 |
         |  0.407712367 |
                      ┘
```

```
RANK  =  3
```

```
INFO  =  0
```

Example 3

This example finds the least squares solution of an overdetermined complex general system; that is, it solves the least squares problem: minimize ‖***B***-***AX***‖. Matrix ***A*** is size 6 × 2 and matrix ***B*** is size 6 × 1.

Notes:

- Because *lwork*=0, ZGELSD dynamically allocates the work area used by this subroutine.
- *rwork* is a real work array of size 871.
- *iwork* is an integer work array of size 22.

Call Statements and Input:

```
              M   N   NRHS   A   LDA  B   LDB  S   RCOND   RANK   WORK LWORK RWORK
IWORK   INFO
              |   |     |    |    |   |    |   |     |       |      |    |     |
|       |
CALL ZGELSD ( 6 , 2 ,   1 ,  A ,  6 , B ,  6 , S , RCOND , RANK , WORK , 0 , RWORK ,
IWORK , INFO )
```

A = (same as input A in Example 6 for ZGELS)
B = (same as input B in Example 6 for ZGELS)

```
RCOND  = .745058D-08
```

Output:

General matrix ***A*** is overwritten.

Solution matrix ***X*** overwrites ***B***:

```
                                ┐
         | (-1.135350,  0.520298) |
         | ( 0.944064,  0.624509) |
B    =   | ( 1.062824, -0.899701) |
         | ( 2.570856,  1.687826) |
         | ( 2.556854,  2.835820) |
         | (-3.982815, -0.231572) |
                                ┘
```

```
                      ┐
        |  4.781121271 |
S    =  |  2.959878261 |
                      ┘
```

```
RANK  =  2
```

```
INFO  =  0
```

Example 4

This example finds the minimum norm solution of an underdetermined complex general system ***AX*** = ***B***. Matrix ***A*** is size 3 × 4. On input, matrix ***B*** is size 3 × 3, stored in array *b* with leading dimension 4.

Notes:

- Because *lwork*=0, ZGELSD dynamically allocates the work area used by this subroutine.
- *rwork* is a real work array of size 1081.
- *iwork* is an integer work array of size 33.

Call Statements and Input:

```
              M   N   NRHS   A   LDA  B   LDB  S   RCOND   RANK   WORK LWORK RWORK
IWORK   INFO
              |   |     |    |    |   |    |   |     |       |      |    |     |
|       |
CALL ZGELSD ( 3 , 4 ,   3 ,  A ,  3 , B ,  4 , S , RCOND , RANK , WORK , 0 , RWORK ,
IWORK , INFO )
```

A = (same as input A in Example 5 for ZGELS)
B = (same as input B in Example 5 for ZGELS)

```
RCOND  = -1
```

Output:

General matrix ***A*** is overwritten.

Solution matrix ***X*** overwrites ***B***:

```
                                                    ┐
         | (-0.16, 0.15) (-0.08, 0.18) ( 0.16,-0.31) |
B    =   | ( 0.11, 0.02) ( 0.21,-0.50) (-0.38, 0.65) |
         | (-0.13,-0.32) ( 0.16, 0.12) (-0.27,-0.28) |
         | ( 0.37,-0.05) ( 0.04, 0.06) (-0.19, 0.33) |
                                                    ┘
```

```
                      ┐
        |  9.895527537 |
S    =  |  4.876518979 |
        |  1.816066467 |
                      ┘
```

```
RANK  =  3
```

```
INFO  =  0
```

SGESVF and DGESVF (Singular Value Decomposition for a General Matrix)

Purpose

These subroutines compute the singular value decomposition of general matrix ***A*** in preparation for solving linear least squares problems. To compute the minimal norm linear least squares solution of ***AX***≅***B***, follow the call to these subroutines with a call to SGESVS or DGESVS, respectively.

Table 200: Data Types.	
Data Types	
A, ***B***, ***s***, *aux*	**Subroutine**
Short-precision real	SGESVF
Long-precision real	DGESVF

Syntax

Language	Syntax
Fortran	CALL SGESVF | DGESVF (*iopt*, *a*, *lda*, *b*, *ldb*, *nb*, *s*, *m*, *n*, *aux*, *naux*)
C and C++	sgesvf | dgesvf (*iopt*, *a*, *lda*, *b*, *ldb*, *nb*, *s*, *m*, *n*, *aux*, *naux*);

On Entry

iopt

indicates the type of computation to be performed, where:

If *iopt* = 0 or 10, singular values are computed.

If *iopt* = 1 or 11, singular values and ***V*** are computed.

If *iopt* = 2 or 12, singular values, ***V***, and $\boldsymbol{U}^T\boldsymbol{B}$ are computed.

Specified as: an integer; *iopt* = 0, 1, 2, 10, 11, or 12.

If *iopt* < 10, singular values are unordered.

If *iopt* ≥ 10, singular values are sorted in descending order and, if applicable, the columns of ***V*** and the rows of $\boldsymbol{U}^T\boldsymbol{B}$ are swapped to correspond to the sorted singular values.

a

is the *m* by *n* general matrix ***A***, whose singular value decomposition is to be computed.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 200 on page 830.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and *lda* ≥ max(*m*, *n*).

b

has the following meaning, where:

If *iopt* = 0, 1, 10, or 11, this argument is not used in the computation.

If *iopt* = 2 or 12, it is the *m* by *nb* matrix ***B***.

Specified as: an *ldb* by (at least) *nb* array, containing numbers of the data type indicated in Table 200 on page 830.

If this subroutine is followed by a call to SGESVS or DGESVS, ***B*** should contain the right-hand side of the linear least squares problem, $\boldsymbol{AX} \cong \boldsymbol{B}$. (The *nb* column vectors of ***B*** contain right-hand sides for *nb* distinct linear least squares problems.) However, if the matrix $\boldsymbol{U}^T$ is desired on output, ***B*** should be equal to the identity matrix of order *m*.

ldb

has the following meaning, where:

If *iopt* = 0, 1, 10, or 11, this argument is not used in the computation.

If *iopt* = 2 or 12, it is the leading dimension of the array specified for *b*.

Specified as: an integer. It must have the following values, where:

If *iopt* = 0, 1, 10, or 11, $ldb > 0$.

If *iopt* = 2 or 12, $ldb > 0$ and $ldb \geq \max(m, n)$.

nb

has the following meaning, where:

If *iopt* = 0, 1, 10, or 11, this argument is not used in the computation.

If *iopt* = 2 or 12, it is the number of columns in matrix ***B***.

Specified as: an integer; if *iopt* = 2 or 12, $nb > 0$.

s

See On Return.

m

is the number of rows in matrices ***A*** and ***B***.

Specified as: an integer; $m \geq 0$.

n

is the number of columns in matrix ***A*** and the number of elements in vector ***s***.

Specified as: an integer; $n \geq 0$.

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 200 on page 830.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SGESVF and DGESVF dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, It must have the following value, where:

If *iopt* = 0 or 10, $naux \geq n+\max(m, n)$.

If *iopt* = 1 or 11, $naux \geq 2n+\max(m, n)$.

If *iopt* = 2 or 12, $naux \geq 2n+\max(m, n, nb)$.

On Return

a

has the following meaning, where:

If *iopt* = 0, or 10, ***A*** is overwritten; that is, the original input is not preserved.

If *iopt* = 1, 2, 11, or 12, ***A*** contains the real orthogonal matrix ***V***, of order *n*, in its first *n* rows and *n* columns. If *iopt* = 11 or 12, the columns of ***V*** are swapped to correspond to the sorted singular values. If *m* > *n*, rows *n*+1, *n*+2, ..., *m* of array A are overwritten; that is, the original input is not preserved.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 200 on page 830.

b

has the following meaning, where:

If *iopt* = 0, 1, 10, or 11, ***B*** is not used in the computation.

If *iopt* = 2 or 12, ***B*** is overwritten by the *n* by *nb* matrix $\boldsymbol{U}^{\mathrm{T}}\boldsymbol{B}$.

If *iopt* = 12, the rows of $\boldsymbol{U}^{\mathrm{T}}\boldsymbol{B}$ are swapped to correspond to the sorted singular values. If *m* > *n*, rows *n* +1, *n*+2, ..., *m* of array B are overwritten; that is, the original input is not preserved.

Returned as: an *ldb* by (at least) *nb* array, containing numbers of the data type indicated in Table 200 on page 830.

s

is a the vector ***s*** of length *n*, containing the singular values of matrix ***A***. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 200 on page 830; $s_i \geq 0$, where:

If *iopt* < 10, the singular values are unordered in ***s***.

If *iopt* ≥ 10, the singular values are sorted in descending order in ***s***; that is, $s_1 \geq s_2 \geq \ldots \geq s_n \geq 0$. If applicable, the columns of ***V*** and the rows of $\boldsymbol{U}^{\mathrm{T}}\boldsymbol{B}$ are swapped to correspond to the sorted singular values.

Notes

1. The following items must have no common elements; otherwise, results are unpredictable: matrices ***A*** and ***B***, vector ***s***, and the data area specified for *aux*.
2. When you specify *iopt* = 0, 1, 10, or 11, you must also specify:
   - A dummy argument for *b*
   - A positive value for *ldb*

   See Example.
3. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

The singular value decomposition of a real general matrix is computed as follows:

$\boldsymbol{A} = \boldsymbol{U}\Sigma\boldsymbol{V}^{\mathrm{T}}$

where:

$\boldsymbol{U}^{\mathrm{T}}\boldsymbol{U} = \boldsymbol{V}^{\mathrm{T}}\boldsymbol{V} = \boldsymbol{V}\boldsymbol{V}^{\mathrm{T}} = \boldsymbol{I}$

A is an *m* by *n* real general matrix.

V is a real general orthogonal matrix of order *n*. On output, ***V*** overwrites the first *n* rows and *n* columns of ***A***.

$\boldsymbol{U}^T\boldsymbol{B}$ is an *n* by *nb* real general matrix. On output, $\boldsymbol{U}^T\boldsymbol{B}$ overwrites the first *n* rows and *nb* columns of ***B***.

Σ is an *n* by *n* real diagonal matrix. The diagonal elements of Σ are the singular values of ***A***, returned in the vector ***s***.

If *m* or *n* is equal to 0, no computation is performed.

One of the following algorithms is used:

1. Golub-Reinsch Algorithm (See pages 134 to 151 in reference [“120” on page 1259].)
   a. Reduce the real general matrix ***A*** to bidiagonal form using Householder transformations.
   b. Iteratively reduce the bidiagonal form to diagonal form using a variant of the QR algorithm.
2. Chan Algorithm (See reference [“21” on page 1254].)
   a. Compute the QR decomposition of matrix ***A*** using Householder transformations; that is, ***A*** = ***QR***.
   b. Apply the Golub-Reinsch Algorithm to the matrix ***R***.

      If $\boldsymbol{R} = \boldsymbol{XWY}^T$ is the singular value decomposition of ***R***, the singular value decomposition of matrix ***A*** is given by:

      $$\boldsymbol{A} = \boldsymbol{Q}\begin{bmatrix}\boldsymbol{X}\\0\end{bmatrix}\boldsymbol{WY}^{\mathrm{T}}$$

      where:

      $$\boldsymbol{U} = \boldsymbol{Q}\begin{bmatrix}\boldsymbol{X}\\0\end{bmatrix}$$

      $$\Sigma = \boldsymbol{W}$$

      $$\boldsymbol{V} = \boldsymbol{Y}$$

Also, see references [“21” on page 1254], [“71” on page 1257], [“94” on page 1258], and pages 134 to 151 in reference [“120” on page 1259]. These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.

Error conditions

Resource Errors
   Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors
   Singular value (*i*) failed to converge after (*x*) iterations.

- The singular values (s_j, $j = n, n\text{-}1, ..., i\text{+}1$) are correct. If *iopt* < 10, they are unordered. Otherwise, they are ordered.
- *a* has been modified.
- If *iopt* = 2 or 12, then *b* has been modified.
- The return code is set to 1.

- *i* and *x* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2107 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. See "What Can You Do about ESSL Computational Errors?" on page 65.

Input-Argument Errors

1. $iopt \neq 0, 1, 2, 10, 11$, or 12
2. $lda \leq 0$
3. $\max(m, n) > lda$
4. $ldb \leq 0$ and $iopt = 2, 12$
5. $\max(m, n) > ldb$ and $iopt = 2, 12$
6. $nb \leq 0$ and $iopt = 2, 12$
7. $m < 0$
8. $n < 0$
9. Error 2015 is recoverable or $naux \neq 0$, and *naux* is too small—that is, less than the minimum required value. Return code 2 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows how to find only the singular values, ***s***, of a real long-precision general matrix ***A***, where:

- `M` is greater than `N`.
- `NAUX` is greater than or equal to `N`+max(`M`, `N`) = 7.
- `LDB` has been set to 1 to avoid a Fortran error message.
- `DUMMY` is a placeholder for argument *b*, which is not used in the computation.
- The singular values are returned in S.
- On output, matrix ***A*** is overwritten; that is, the original input is not preserved.

Call Statement and Input:

```
             IOPT  A  LDA    B    LDB  NB  S   M   N   AUX  NAUX
              |    |   |     |     |   |   |   |   |    |    |
CALL DGESVF( 0  , A , 4 , DUMMY , 1 , 0 , S , 4 , 3 , AUX , 7  )


        |  1.0   2.0   3.0 |
A    =  |  4.0   5.0   6.0 |
        |  7.0   8.0   9.0 |
        | 10.0  11.0  12.0 |
```

Output:

```
S        =   (25.462, 1.291, 0.000)
```

Example 2

This example computes the singular values, ***s***, of a real long-precision general matrix ***A*** and the matrix ***V***, where:

- `M` is equal to `N`.
- `NAUX` is greater than or equal to 2N+max(`M`, `N`) = 9.
- `LDB` has been set to 1 to avoid a Fortran error message.
- `DUMMY` is a placeholder for argument *b*, which is not used in the computation.
- The singular values are returned in S.

- The matrix ***V*** is returned in A.

Call Statement and Input:

```
             IOPT  A  LDA    B    LDB  NB  S   M   N   AUX  NAUX
              |    |   |     |     |   |   |   |   |    |    |
CALL DGESVF( 1  , A , 3 , DUMMY , 1 , 0 , S , 3 , 3 , AUX , 9  )


A    =  |  2.0  1.0  1.0 |
        |  4.0  1.0  0.0 |
        | -2.0  2.0  1.0 |
```

Output:

```
        | -0.994   0.105  -0.041 |
A    =  | -0.112  -0.870   0.480 |
        | -0.015  -0.482  -0.876 |


S        =    (4.922, 2.724, 0.597)
```

Example 3

This example computes the singular values, ***s***, and computes matrices ***V*** and $\boldsymbol{U}^T\boldsymbol{B}$ in preparation for solving the underdetermined system ***AX≅B***, where:

- M is less than N.
- NAUX is greater than or equal to 2N+max(M, N, NB) = 9.
- The singular values are returned in S.
- The matrix ***V*** is returned in A.
- The matrix $\boldsymbol{U}^T\boldsymbol{B}$ is returned in B.

Call Statement and Input:

```
             IOPT  A  LDA  B  LDB  NB  S   M   N   AUX  NAUX
              |    |   |   |   |   |   |   |   |    |    |
CALL DGESVF( 2  , A , 3 , B , 3 , 1 , S , 2 , 3 , AUX , 9  )
```

```
        | 1.0  2.0  2.0 |
A    =  | 2.0  4.0  5.0 |
        |  .    .    .  |
```

```
        | 1.0 |
B    =  | 4.0 |
        |  .  |
```

Output:

```
        | -0.304  -0.894   0.328 |
A    =  | -0.608   0.447   0.656 |
        | -0.733   0.000  -0.680 |


        | -4.061 |
B    =  |  0.000 |
        | -0.714 |


S        =   (7.342, 0.000, 0.305)
```

Example 4

This example computes the singular values, ***s***, and matrices ***V*** and $\boldsymbol{U}^T\boldsymbol{B}$ in preparation for solving the overdetermined system ***AX***≅***B***, where:

- M is greater than N.
- NAUX is greater than or equal to 2N+max(M, N, NB) = 7.
- The singular values are returned in S.
- The matrix ***V*** is returned in A.
- The matrix $\boldsymbol{U}^T\boldsymbol{B}$ is returned in B.

Call Statement and Input:

```
            IOPT  A  LDA  B  LDB  NB  S   M   N   AUX  NAUX
             |    |   |   |   |   |   |   |   |    |    |
CALL DGESVF( 2  , A , 3 , B , 3 , 2 , S , 3 , 2 , AUX , 7  )
```

```
        | 1.0  4.0 |
A    =  | 2.0  5.0 |
        | 3.0  6.0 |
```

```
        | 7.0  10.0 |
B    =  | 8.0  11.0 |
        | 9.0  12.0 |
```

Output:

```
        |  0.922  -0.386 |
A    =  | -0.386  -0.922 |
        |   .       .    |
```

```
        |  -1.310   -2.321 |
B    =  | -13.867  -18.963 |
        |    .        .    |


X        =   (0.773, 9.508)
```

Example 5

This example computes the singular values, ***s***, and matrices ***V*** and $\boldsymbol{U}^T\boldsymbol{B}$ in preparation for solving the overdetermined system ***AX***≅***B***. The singular values are sorted in descending order, and the columns of ***V*** and the rows of $\boldsymbol{U}^T\boldsymbol{B}$ are swapped to correspond to the sorted singular values.

- M is greater than N.
- NAUX is greater than or equal to 2N+max(M, N, NB) = 7.
- The singular values are returned in S.
- The matrix ***V*** is returned in A.
- The matrix $\boldsymbol{U}^T\boldsymbol{B}$ is returned in B.

Call Statement and Input:

```
            IOPT  A  LDA  B  LDB  NB  S   M   N   AUX  NAUX
             |    |   |   |   |   |   |   |   |    |    |
CALL DGESVF( 12 , A , 3 , B , 3 , 2 , S , 3 , 2 , AUX , 7  )
```

```
        | 1.0  4.0 |
A    =  | 2.0  5.0 |
         | 3.0  6.0 |
```

```
         | 7.0  10.0 |
B    =   | 8.0  11.0 |
         | 9.0  12.0 |
```

Output:

```
         | -0.386   0.922 |
A    =   | -0.922  -0.386 |
         |   .       .    |
```

```
         | -13.867  -18.963 |
B    =   |  -1.310   -2.321 |
         |   .         .    |


S         =   (9.508, 0.773)
```

SGESVS and DGESVS (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)

Purpose

These subroutines compute the minimal norm linear least squares solution of ***AX*≅*B***, where ***A*** is a general matrix, using the singular value decomposition computed by SGESVF or DGESVF.

Table 201: Data Types.

Data Types	
V, ***UB***, ***X***, ***s***, τ	**Subroutine**
Short-precision real	SGESVS
Long-precision real	DGESVS

Syntax

Language	**Syntax**
Fortran	CALL SGESVS | DGESVS (*v*, *ldv*, *ub*, *ldub*, *nb*, *s*, *x*, *ldx*, *m*, *n*, *tau*)
C and C++	sgesvs | dgesvs (*v*, *ldv*, *ub*, *ldub*, *nb*, *s*, *x*, *ldx*, *m*, *n*, *tau*);

On Entry

v

is the orthogonal matrix ***V*** of order *n* in the singular value decomposition of matrix ***A***. It is produced by a preceding call to SGESVF or DGESVF, where it corresponds to output argument *a*.

Specified as: an *ldv* by (at least) *n* array, containing numbers of the data type indicated in Table 201 on page 837.

ldv

is the leading dimension of the array specified for *v*.

Specified as: an integer; *ldv* > 0 and *ldv* ≥ *n*.

ub
is an *n* by *nb* matrix, containing $\boldsymbol{U}^T\boldsymbol{B}$. It is produced by a preceding call to SGESVF or DGESVF, where it corresponds to output argument *b*. On output, $\boldsymbol{U}^T\boldsymbol{B}$ is overwritten; that is, the original input is not preserved.

Specified as: an *ldub* by (at least) *nb* array, containing numbers of the data type indicated in Table 201 on page 837.

ldub
is the leading dimension of the array specified for *ub*.

Specified as: an integer; $ldub > 0$ and $ldub \geq n$.

nb
is the number of columns in matrices ***X*** and $\boldsymbol{U}^T\boldsymbol{B}$.

Specified as: an integer; $nb > 0$.

s
is the vector ***s*** of length *n*, containing the singular values of matrix ***A***. It is produced by a preceding call to SGESVF or DGESVF, where it corresponds to output argument *s*.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 201 on page 837; $s_i \geq 0$.

x
See On Return.

ldx
is the leading dimension of the array specified for *x*.

Specified as: an integer; $ldx > 0$ and $ldx \geq n$.

m
is the number of rows in matrix ***A***.

Specified as: an integer; $m \geq 0$.

n
is the number of columns in matrix ***A***, the order of matrix ***V***, the number of elements in vector ***s***, the number of rows in matrix ***UB***, and the number of rows in matrix ***X***. Specified as: an integer; $n \geq 0$.

tau
is the error tolerance τ. Any singular values in vector ***s*** that are less than τ are treated as zeros when computing matrix ***X***.

Specified as: a number of the data type indicated in Table 201 on page 837; $\tau \geq 0$. For more information on the values for τ, see "Notes" on page 838.

On Return

x
is an *n* by *nb* matrix, containing the minimal norm linear least solutions of $\boldsymbol{AX} \cong \boldsymbol{B}$. The *nb* column vectors of ***X*** contain minimal norm solution vectors for *nb* distinct linear least squares problems.

Returned as: an *ldx* by (at least) *nb* array, containing numbers of the data type indicated in Table 201 on page 837.

Notes

1. ***V***, ***X***, ***s***, and $\boldsymbol{U}^T\boldsymbol{B}$ can have no common elements; otherwise the results are unpredictable.
2. In problems involving experimental data, τ should reflect the absolute accuracy of the matrix elements:

   $\tau \geq \max(|\Delta_{ij}|)$

where Δ_{ij} are the errors in a_{ij}. In problems where the matrix elements are known exactly or are only affected by roundoff errors:

$$\left[\tau \geq \varepsilon\left(\sqrt{mn}\right)\right] \max\left(s_j\right) \qquad \text{for} \quad j = \left(1, \ldots, n\right)$$

where:

ε is equal to 0.11920E-06 for SGESVS and 0.22204D-15 for DGESVS. ***s*** is a vector containing the singular values of matrix ***A***.

For more information, see references [“21” on page 1254], [“71” on page 1257], [“94” on page 1258], and pages 134 to 151 in reference [“120” on page 1259].

Function

The minimal norm linear least squares solution of ***AX≅B***, where ***A*** is a real general matrix, is computed using the singular value decomposition, produced by a preceding call to SGESVF or DGESVF. From SGESVF or DGESVF, the singular value decomposition of ***A*** is given by the following:

$\boldsymbol{A} = \boldsymbol{U}\Sigma\boldsymbol{V}^{T}$

The linear least squares of solution ***X***, for ***AX≅B***, is given by the following formula:

$\boldsymbol{X} = \boldsymbol{V}\Sigma^{+}\boldsymbol{U}^{T}\boldsymbol{B}$

where:

$$\Sigma + \text{ is the diagonal matrix with elements } \sigma_j^+\text{, where:}$$
$$\sigma_j^+ = 1.0/\sigma_j \quad \text{if } \sigma_j \geq \tau \text{ and } \sigma_j \neq 0$$
$$\sigma_j^+ = 0 \quad \text{for all other cases}$$

If *m* or *n* is equal to 0, no computation is performed. See references [“21” on page 1254], [“71” on page 1257], [“94” on page 1258], and pages 134 to 151 in reference [“120” on page 1259]. These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.

Error conditions

Computational Errors
None

Input-Argument Errors

1. *ldv* ≤ 0
2. *n* > *ldv*
3. *ldub* ≤ 0
4. *n* > *ldub*
5. *ldx* ≤ 0
6. *n* > *ldx*
7. *nb* ≤ 0
8. *m* < 0
9. *n* < 0

10. τ < 0

Examples

Example 1

This example finds the linear least squares solution for the underdetermined system ***AX≅B***, using the singular value decomposition computed by DGESVF. Matrix ***A*** is:

```
                        | 1.0  2.0  2.0 |
                        | 2.0  4.0  5.0 |
```

and matrix ***B*** is:

```
                            | 1.0 |
                            | 4.0 |
                            └
```

On output, matrix $\boldsymbol{U}^T\boldsymbol{B}$ is overwritten.

Note: This example corresponds to DGESVF Example 3.

Call Statement and Input:

```
              V  LDV  UB  LDUB  NB  S   X  LDX  M   N   TAU
              |   |   |    |    |   |   |   |   |   |    |
CALL DGESVS( V , 3 , UB , 3  , 1 , S , X , 3 , 2 , 3 , TAU )
```

```
        | -0.304  -0.894   0.328 |
V    =  | -0.608   0.447   0.656 |
        | -0.733   0.000  -0.680 |
```

```
        | -4.061 |
UB   =  |  0.000 |
        | -0.714 |


S        =  (7.342, 0.000, 0.305)
TAU      =  0.3993D-14
```

Output:

```
        | -0.600 |
X    =  | -1.200 |
        |  2.000 |
```

Example 2

This example finds the linear least squares solution for the overdetermined system ***AX≅B***, using the singular value decomposition computed by DGESVF. Matrix ***A*** is:

```
                           | 1.0  4.0 |
                           | 2.0  5.0 |
                           | 3.0  6.0 |
```

and where ***B*** is:

```
                           | 7.0  10.0 |
                           | 8.0  11.0 |
                            | 9.0  12.0 |
```

On output, matrix $\boldsymbol{U}^T\boldsymbol{B}$ is overwritten.

Note: This example corresponds to DGESVF Example 4.

Call Statement:

```
             V  LDV  UB  LDUB  NB  S   X  LDX  M   N   TAU
             |   |   |    |    |   |   |   |   |   |    |
CALL DGESVS( V , 3 , UB , 3  , 2 , S , X , 2 , 3 , 2 , TAU )
```

Input:

```
        |  0.922  -0.386 |
V    =  | -0.386  -0.922 |
        |    .       .   |
```

```
        |  -1.310   -2.321 |
UB   =  | -13.867  -18.963 |
        |     .        .   |


S        =  (0.773, 9.508)
TAU      =  0.5171D-14
```

Output:

```
X    =  | -1.000  -2.000 |
        |  2.000   3.000 |
```

SGELLS and DGELLS (Linear Least Squares Solution for a General Matrix with Column Pivoting)

Purpose

These subroutines compute the minimal norm linear least squares solution of $\boldsymbol{AX} \cong \boldsymbol{B}$, using a QR decomposition with column pivoting.

Table 202: Data Types.

Data Types	
A, B, X, rn, τ, *aux*	**Subroutine**
Short-precision real	SGELLS
Long-precision real	DGELLS

Syntax

Language	Syntax
Fortran	CALL SGELLS | DGELLS (*iopt*, *a*, *lda*, *b*, *ldb*, *x*, *ldx*, *rn*, *tau*, *m*, *n*, *nb*, *k*, *aux*, *naux*)
C and C++	sgells | dgells (*iopt*, *a*, *lda*, *b*, *ldb*, *x*, *ldx*, *rn*, *tau*, *m*, *n*, *nb*, *k*, *aux*, *naux*);

On Entry

iopt

indicates the type of computation to be performed, where:

If *iopt* = 0, ***X*** is computed.

If *iopt* = 1, ***X*** and the Euclidean Norm of the residual vectors are computed.

Specified as: an integer; *iopt* = 0 or 1.

a

is the *m* by *n* coefficient matrix ***A***. On output, ***A*** is overwritten; that is, the original input is not preserved.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 202 on page 841.

lda

is the leading dimension of the array specified for *a*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *m*.

b

is the *m* by *nb* matrix ***B***, containing the right-hand sides of the linear systems. The *nb* column vectors of ***B*** contain right-hand sides for *nb* distinct linear least squares problems. On output, ***B*** is overwritten; that is, the original input is not preserved.

Specified as: an *ldb* by (at least) *nb* array, containing numbers of the data type indicated in Table 202 on page 841.

ldb

is the leading dimension of the array specified for *b*.

Specified as: an integer; *ldb* > 0 and *ldb* ≥ *m*.

x

See On Return.

ldx

is the leading dimension of the array specified for *x*.

Specified as: an integer; *ldx* > 0 and *ldx* ≥ *n*.

rn

See On Return.

tau

is the tolerance τ, used to determine the subset of the columns of ***A*** used in the solution.

Specified as: a number of the data type indicated in Table 202 on page 841; τ ≥ 0. For more information on how to select a value for τ, see "Notes" on page 843.

m

is the number of rows in matrices ***A*** and ***B***.

Specified as: an integer; *m* ≥ 0.

n

is the number of columns in matrix ***A*** and the number of rows in matrix ***X***.

Specified as: an integer; *n* ≥ 0.

nb

is the number of columns in matrices ***B*** and ***X*** and the number of elements in vector ***rn***.

Specified as: an integer; *nb* > 0.

k

See On Return.

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 202 on page 841. On output, the contents of *aux* are overwritten.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SGELLS and DGELLS dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, it must have the following values:

For SGELLS

For SGELLS, it must have the following values:

For 32-bit integer arguments

$naux \geq 3n + \max(n, nb)$

For 64-bit integer arguments

$naux \geq 4n + \max(n, nb) + 3$

For DGELLS

For DGELLS, it must have the following values:

For 32-bit integer arguments

$naux \geq [\text{ceiling}(2.5n) + \max(n, nb)]$

For 64-bit integer arguments

$naux \geq 3n + \max(n, nb)$

On Return

x

is the solution matrix ***X***, with *n* rows and *nb* columns, where:

If $k \neq 0$, the *nb* column vectors of ***X*** contain minimal norm least squares solutions for *nb* distinct linear least squares problems. The elements in each solution vector correspond to the original columns of ***A***.

If $k = 0$, the *nb* column vectors of ***X*** are set to 0.

Returned as: an *ldx* by (at least) *nb* array, containing numbers of the data type indicated in Table 202 on page 841.

rn

is the vector ***rn*** of length *nb*, where:

If *iopt* = 0 or *k* = 0, ***rn*** is not used in the computation.

If *iopt* = 1, rn_i is the Euclidean Norm of the residual vector for the linear least squares problem defined by the *i*-th column vector of ***B***.

Returned as: a one-dimensional array of (at least) *nb*, containing numbers of the data type indicated in Table 202 on page 841.

k

is the number of columns of matrix ***A*** used in the solution. Returned as: an integer; *k* = (the number of diagonal elements of matrix ***R*** exceeding τ in magnitude).

Notes

1. In your C program, argument *k* must be passed by reference.
2. If $ldb \geq \max(m, n)$, matrix ***X*** and matrix ***B*** can be the same; otherwise, matrix ***X*** and matrix ***B*** can have no common elements, or the results are unpredictable.

3. The following items must have no common elements; otherwise, results are unpredictable:
   - Matrices ***A*** and ***X***, vector ***rn***, and the data area specified for *aux*
   - Matrices ***A*** and ***B***, vector ***rn***, and the data area specified for *aux*.
4. If the relative uncertainty in the matrix ***B*** is ρ, then:

   $\tau \geq \rho \| \boldsymbol{A} \|_F$

   See references [“54” on page 1256], [“75” on page 1257], and [“94” on page 1258] for additional guidance on determining suitable values for τ.
5. When you specify *iopt* = 0, you must also specify a dummy argument for *rn*. For more details, see Example 1.
6. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 49.

Function

The minimal norm linear least squares solution of ***AX≅B*** is computed using a QR decomposition with column pivoting, where:

A is an *m* by *n* real general matrix.
B is an *m* by *nb* real general matrix.
X is an *n* by *nb* real general matrix.

Optionally, the Euclidean Norms of the residual vectors can be computed. Following are the steps involved in finding the minimal norm linear least squares solution of ***AX≅B***. ***A*** is decomposed, using Householder transformations and column pivoting, into the following form:

AP = ***QR***

where:

P is a permutation matrix.
Q is an orthogonal matrix.
R is an upper triangular matrix.

k is the first index, where:

$|r_{k+1,k+1}| \leq \tau$

If $k = n$, the minimal norm linear least squares solution is obtained by solving $\boldsymbol{RX} = \boldsymbol{Q}^T\boldsymbol{B}$ and reordering ***X*** to correspond to the original columns of ***A.***

If $k < n$, ***R*** has the following form:

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ 0 & 0 \end{bmatrix}$$

To find the minimal norm linear least squares solution, it is necessary to zero the submatrix $\boldsymbol{R}_{12}$ using Householder transformations. See references [“54” on page 1256], [“75” on page 1257], and [“94” on page 1258]. If *m* or *n* is equal to 0, no computation is performed. These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.

Error conditions

Resource Errors

Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors

None

Input-Argument Errors

1. *iopt* ≠ 0 or 1
2. *lda* ≤ 0
3. *m* > *lda*
4. *ldb* ≤ 0
5. *m* > *ldb*
6. *ldx* ≤ 0
7. *n* > *ldx*
8. *m* < 0
9. *n* < 0
10. *nb* ≤ 0
11. τ < 0
12. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example solves the underdetermined system ***AX***≅***B***. On output, ***A*** and ***B*** are overwritten. DUMMY is used as a placeholder for argument *rn*, which is not used in the computation.

Call Statement and Input:

```
            IOPT  A  LDA  B  LDB  X  LDX   RN     TAU   M   N   NB  K   AUX  NAUX
             |    |   |   |   |   |   |     |       |    |   |   |   |    |    |
CALL DGELLS( 0  , A , 2 , B , 2 , X , 3 , DUMMY , TAU , 2 , 3 , 1 , K , AUX , 11 )
```

```

A    =  | 1.0  2.0  2.0 |
        | 2.0  4.0  5.0 |



B    =  | 1.0 |
        | 4.0 |


TAU      =   0.0
```

Output:

```

        | -0.600 |
X    =  | -1.200 |
        |  2.000 |


K        =   2
```

Example 2

This example solves the overdetermined system ***AX***≅***B***. On output, ***A*** and ***B*** are overwritten. DUMMY is used as a placeholder for argument *rn*, which is not used in the computation.

Call Statement and Input:

```
            IOPT  A  LDA  B  LDB  X  LDX   RN      TAU   M   N   NB  K   AUX  NAUX
             |    |   |   |   |   |   |     |       |    |   |   |   |    |    |
CALL DGELLS( 0  , A , 3 , B , 3 , X , 2 , DUMMY , TAU , 3 , 2 , 2 , K , AUX , 7  )
```

```
        | 1.0  4.0 |
A    =  | 2.0  5.0 |
        | 3.0  6.0 |
```

```
        | 7.0  10.0 |
B    =  | 8.0  11.0 |
        | 9.0  12.0 |


TAU      =   0.0
```

Output:

```
X    =  | -1.000  -2.000 |
        |  2.000   3.000 |


K        =   2
```

Example 3

This example solves the overdetermined system ***AX***≅***B*** and computes the Euclidean Norms of the residual vectors. On output, ***A*** and ***B*** are overwritten.

Call Statement and Input:

```
            IOPT  A  LDA  B  LDB  X  LDX  RN   TAU   M   N   NB  K   AUX  NAUX
             |    |   |   |   |   |   |   |     |    |   |   |   |    |    |
CALL DGELLS( 1  , A , 3 , B , 3 , X , 2 , RN , TAU , 3 , 2 , 2 , K , AUX , 7  )
```

```
        | 1.1  -4.3 |
A    =  | 2.0  -5.0 |
        | 3.0  -6.0 |
```

```
        | -7.0  10.0 |
B    =  | -8.0  11.0 |
        | -9.0  12.0 |


TAU      =   0.0
```

Output:

```
X    =  | 0.543  -1.360 |
        | 1.785  -2.699 |
```

```
RN   =  | 0.196 |
        | 0.275 |


K        =   2
```

Chapter 11. Eigensystem analysis subroutines

The eigensystem analysis subroutines are described here.

Overview of the Eigensystem Analysis Subroutines

The eigensystem analysis subroutines provide solutions to the algebraic eigensystem analysis problem and the generalized eigensystem analysis problem. These subroutines correspond to the LAPACK routines described in reference [“9” on page 1253].

Table 203: List of LAPACK Eigensystem Analysis Subroutines.

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SGEEV$^{\Delta}$ CGEEV$^{\Delta}$ SGEEVX$^{\Delta}$ CGEEVX$^{\Delta}$ LAPACKE_sgeev$^{\Delta}$ LAPACKE_cgeev$^{\Delta}$ LAPACKE_sgeevx Δ LAPACKE_cgeevx Δ	DGEEV$^{\Delta}$ ZGEEV$^{\Delta}$ DGEEVX$^{\Delta}$ ZGEEVX$^{\Delta}$ LAPACKE_dgeev$^{\Delta}$ LAPACKE_zgeev$^{\Delta}$ LAPACKE_dgeevx Δ LAPACKE_zgeevx Δ	“SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)” on page 850

Table 203: List of LAPACK Eigensystem Analysis Subroutines. (continued)

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SSYEV$^{\Delta}$ CHEEV$^{\Delta}$ SSPEVX$^{\Delta}$ CHPEVX$^{\Delta}$ SSYEVX$^{\Delta}$ CHEEVX$^{\Delta}$ >|SSYEVR$^{\Delta}$|< >|CHEEVR$^{\Delta}$|< LAPACKE_ssyev$^{\Delta}$ LAPACKE_cheev$^{\Delta}$ LAPACKE_sspevx $^{\Delta}$ LAPACKE_chpevx $^{\Delta}$ LAPACKE_ssyevx $^{\Delta}$ LAPACKE_cheevx $^{\Delta}$ >| LAPACKE_ssyevr$^{\Delta}$ |< >| LAPACKE_cheevr $^{\Delta}$|<	DSYEV$^{\Delta}$ ZHEEV$^{\Delta}$ DSPEVX$^{\Delta}$ ZHPEVX$^{\Delta}$ DSYEVX$^{\Delta}$ ZHEEVX$^{\Delta}$ >|DSYEVR$^{\Delta}$|< >|ZHEEVR$^{\Delta}$|< LAPACKE_dsyev$^{\Delta}$ LAPACKE_zheev$^{\Delta}$ LAPACKE_dspevx $^{\Delta}$ LAPACKE_zhpevx $^{\Delta}$ LAPACKE_dsyevx $^{\Delta}$ LAPACKE_zheevx $^{\Delta}$ >| LAPACKE_dsyevr$^{\Delta}$ |< >| LAPACKE_zheevr $^{\Delta}$|<	"SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, ZHEEVX, SSYEVR, DSYEVR, CHEEVR, and ZHEEVR (Eigenvalues, and Optionally, the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix) " on page 867
SSPEVD$^{\Delta}$ CHPEVD$^{\Delta}$ SSYEVD$^{\Delta}$ CHEEVD$^{\Delta}$ LAPACKE_sspevd $^{\Delta}$ LAPACKE_chpevd $^{\Delta}$ LAPACKE_ssyevd $^{\Delta}$ LAPACKE_cheevd $^{\Delta}$	DSPEVD$^{\Delta}$ ZHPEVD$^{\Delta}$ DSYEVD$^{\Delta}$ ZHEEVD$^{\Delta}$ LAPACKE_dspevd $^{\Delta}$ LAPACKE_zhpevd $^{\Delta}$ LAPACKE_dsyevd $^{\Delta}$ LAPACKE_zheevd $^{\Delta}$	"SSPEVD, DSPEVD, CHPEVD, ZHPEVD, SSYEVD, DSYEVD, CHEEVD, and ZHEEVD (Eigenvalues and, Optionally the Eigenvectors, of a Real Symmetric or Complex Hermitian Matrix Using a Divide-and-Conquer Algorithm) " on page 893
SGGEV$^{\Delta}$ CGGEV$^{\Delta}$ SGGEVX$^{\Delta}$ CGGEVX$^{\Delta}$ LAPACKE_sggev$^{\Delta}$ LAPACKE_cggev$^{\Delta}$ LAPACKE_sggevx $^{\Delta}$ LAPACKE_cggevx $^{\Delta}$	DGGEV$^{\Delta}$ ZGGEV$^{\Delta}$ DGGEVX$^{\Delta}$ ZGGEVX$^{\Delta}$ LAPACKE_dggev$^{\Delta}$ LAPACKE_zggev$^{\Delta}$ LAPACKE_dggevx $^{\Delta}$ LAPACKE_zggevx $^{\Delta}$	"SGGEV, DGGEV, CGGEV, ZGGEV, SGGEVX, DGGEVX, CGGEVX, and ZGGEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix Generalized Eigenproblem)" on page 905

Table 203: List of LAPACK Eigensystem Analysis Subroutines. (continued)

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
>| SSYGVD$^{\Delta}$ CHEGVD$^{\Delta}$ LAPACKE_ssygvd $^{\Delta}$ LAPACKE_chegvd $^{\Delta}$	DSYGVD$^{\Delta}$ ZHEGVD$^{\Delta}$ LAPACKE_dsygvd $^{\Delta}$ LAPACKE_zhegvd $^{\Delta}$	"SSYGVD, DSYGVD, CHEGVD, ZHEGVD (Eigenvalues, and Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem Using a Divide-and-Conquer Algorithm)" on page 922|<
>|SSYGV$^{\Delta}$|< >|CHEGV$^{\Delta}$|< SSPGVX$^{\Delta}$ CHPGVX$^{\Delta}$ SSYGVX$^{\Delta}$ CHEGVX$^{\Delta}$ >| LAPACKE_ssygv$^{\Delta}$ |< >| LAPACKE_chegv$^{\Delta}$ |< LAPACKE_sspgvx $^{\Delta}$ LAPACKE_chpgvx $^{\Delta}$ LAPACKE_ssygvx $^{\Delta}$ LAPACKE_chegvx $^{\Delta}$	>|DSYGV$^{\Delta}$|< >|ZHEGV$^{\Delta}$|< DSPGVX$^{\Delta}$ ZHPGVX$^{\Delta}$ DSYGVX$^{\Delta}$ ZHEGVX$^{\Delta}$ >| LAPACKE_dsygv$^{\Delta}$ |< >| LAPACKE_zhegv$^{\Delta}$ |< LAPACKE_dspgvx $^{\Delta}$ LAPACKE_zhpgvx $^{\Delta}$ LAPACKE_dsygvx $^{\Delta}$ LAPACKE_zhegvx $^{\Delta}$	"SSYGV, DSYGV, CHEGV, ZHEGV, SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX (Eigenvalues, and Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem)" on page 931
$^{\Delta}$LAPACK		

Performance and Accuracy Considerations

1. The short precision subroutines provide increased accuracy by accumulating intermediate results in long precision when the VSX unit is not used. Occasionally, for performance reasons, these intermediate results are stored.
2. There are some ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 61.

Eigensystem Analysis Subroutines

This contains the eigensystem analysis subroutine descriptions.

SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)

Purpose

- SGEEV, DGEEV, CGEEV, and ZGEEV compute the eigenvalues and, optionally, right eigenvectors and/or left eigenvectors of a general matrix.
- SGEEVX, DGEEVX, CGEEVX, and ZGEEVX compute the eigenvalues and, optionally, right eigenvectors, left eigenvectors, reciprocal condition numbers for eigenvalues, and reciprocal condition numbers for right eigenvectors of a general matrix.

For a right eigenvector ***v*** of ***A***:

$\boldsymbol{Av} = \boldsymbol{\lambda v}$

For a left eigenvector ***u*** of ***A***:

$\boldsymbol{u^H A} = \boldsymbol{\lambda u^H}$

The computed eigenvectors are normalized to have the Euclidean norm equal to one and the largest component real.

Table 204: Data Types.

A*, *vl*, *vr*, *work*, *wr*, *wi*, *w	***scale*, *abnrm*, *rconde*, *rcondv*, *rwork***	**Subroutine**
Short-precision real	Short-precision real	SGEEV$^{\Delta}$ SGEEVX$^{\Delta}$
Long-precision real	Long-precision real	DGEEV$^{\Delta}$ DGEEVX$^{\Delta}$
Short-precision complex	Short-precision real	CGEEV$^{\Delta}$ CGEEVX$^{\Delta}$
Long-precision complex	Long-precision real	ZGEEV$^{\Delta}$ ZGEEVX$^{\Delta}$

$^{\Delta}$ LAPACK

Syntax

Language	Syntax
Fortran	CALL SGEEV | DGEEV (*jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, work, lwork, info*) CALL CGEEV | ZGEEV (*jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work, lwork, rwork, info*) CALL SGEEVX | 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 | ZGEEVX (*balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, rwork, info*)
C and C++	sgeev | dgeev (*jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, work, lwork, info*); cgeev | zgeev (*jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work, lwork, rwork, info*); sgeevx | dgeevx (*balanc, jobvl, jobvr, sense, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, iwork, info*); cgeevx | zgeevx (*balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, rwork, info*);

LAPACKE	info = LAPACKE_sgeev | LAPACKE_dgeev (*matrix_layout, jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr*);
	info = LAPACKE_cgeev | LAPACKE_zgeev (*matrix_layout, jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr*);
	info = LAPACKE_sgeevx | LAPACKE_dgeevx (*matrix_layout, balanc, jobvl, jobvr, sense, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrm, rconde, rcondv*);
	info = LAPACKE_cgeevx | LAPACKE_zgeevx (*matrix_layout, balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrm, rconde, rcondv*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

balanc

indicates whether or not to scale ***A*** diagonally and whether or not to permute its rows and columns to improve the conditioning of its eigenvalues, where *balanc* can have any of the following values:

N
Neither diagonally scale nor permute ***A***.

P
Permute ***A***, but do not diagonally scale it.

S
Diagonally scale ***A***, but do not permute it.

B
Both diagonally scale and permute ***A***.

When diagonal scaling is specified, the subroutine replaces ***A*** with ***DAD***$^{-1}$ where ***D*** is a diagonal matrix chosen to make the rows and columns of ***A*** more equal in norm and the condition numbers of its eigenvalues and eigenvectors smaller.

When permuting is specified, the subroutine makes ***A*** more nearly upper triangular.

The computed reciprocal condition numbers correspond to the balanced matrix. In exact arithmetic, permuting rows and columns does not change the condition numbers, but diagonal scaling does change the condition numbers.

Specified as: a single character. It must be 'N', 'P', 'S', or 'B'.

jobvl
indicates whether or not to compute the left eigenvectors of ***A***, where *jobvl* can have either of the following values:

N
Do not compute the left eigenvectors of ***A***.

V
Compute the left eigenvectors of ***A***.

Note: If *sense* = 'E' or 'B', *jobvl* must = 'V'.

Specified as: a single character. It must be 'N' or 'V'.

jobvr
indicates whether or not to compute the right eigenvectors of ***A***, where *jobvr* can have either of the following values:

N
Do not compute the right eigenvectors of ***A***.

V
Compute the right eigenvectors of ***A***.

Note: If *sense* = 'E' or 'B', *jobvr* must = 'V'.

Specified as: a single character. It must be 'N' or 'V'.

sense
indicates which reciprocal numbers to compute (if any), where *sense* can have any of the following values:

N
Do not compute reciprocal condition numbers.

E
Compute reciprocal condition numbers for eigenvalues only.

V
Compute reciprocal condition numbers for right eigenvectors only.

B
Compute reciprocal condition numbers for eigenvalues and right eigenvectors.

Note: If *sense* = 'E' or 'B', both *jobvl* and *jobvr* must equal 'V' (so that both left and right eigenvectors are also computed).

Specified as: a single character. It must be 'N', 'E', 'V', or 'B'.

n
is the order of the general matrix ***A***.

Specified as: an integer; $n \geq 0$.

a
is the general matrix ***A*** of order *n*.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 204 on page 850.

lda
is the leading dimension of the array specified for *a*.

Specified as: an integer, where $lda > 0$ and $lda \ge n$

wr
See On Return.

wi
See On Return.

w
See On Return.

ldvl
is the leading dimension of the array specified for *vl*.

Specified as: an integer; $ldvl > 0$; if *jobvl* = 'V', $ldvl \ge n$.

ldvr
is the leading dimension of the array specified for *vr*.

Specified as: an integer; $ldvr > 0$; if *jobvr* = 'V', $ldvr \ge n$.

work
is the storage work area used by this subroutine. Its size is specified by *lwork*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 204 on page 850.

lwork
is the number of elements in array `WORK`.

Specified as an integer, where:

- If *lwork* = 0, the subroutine dynamically allocates the workspace needed for use during this computation. The dynamically allocated workspace will be freed prior to returning control to the calling program.
- If *lwork* = -1, a workspace query is assumed. The subroutine only calculates the optimal size of the `WORK` array and returns this value as the first entry of the `WORK` array.

Otherwise:

- For SGEEV and DGEEV:
  - If *jobvl* = 'N' or *jobvr* = 'N', $lwork \ge \max(1, 3n)$.
  - If *jobvl* = 'V' or *jobvr* = 'V', $lwork \ge \max(1, 4n)$.
- For CGEEV and ZGEEV, $lwork \ge \max(1, 2n)$.
- For SGEEVX and DGEEVX:
  - If *sense* = 'N' or 'E':
    - If *jobvl* = 'N' and *jobvr* = 'N', $lwork \ge \max(1, 2n)$.
    - If *jobvl* = 'V' or *jobvr* = 'V', $lwork \ge \max(1, 3n)$.
  - If *sense* = 'V' or 'B', $lwork \ge \max(1, n(n + 6))$.
- For CGEEVX and ZGEEVX:
  - If *sense* = 'N' or 'E', $lwork \ge \max(1, 2n)$.
  - If *sense* = 'V' or 'B', $lwork \ge \max(1, n^2 + 6n^2)$.

Note: These formulas represent the minimum workspace required. For best performance, specify either *lwork* = -1 (to obtain the optimal size to use) or *lwork* = 0 (to direct the subroutine to dynamically allocate the workspace).

rwork
is a storage work area of size $2n$.

Specified as: an area of storage containing numbers of the data type indicated in Table 204 on page 850.

iwork

is a storage work area of size max(1, 2*n*-2).

If *sense* = 'N' or 'E', *iwork* is not referenced by the subroutine.

Specified as: an integer array.

On Return

a

is the updated general matrix ***A*** of order *n*. On output, A is overwritten; that is, the original input is not preserved. If *jobvl* = 'V' or *jobvr* = 'V', A contains the Schur form of the balanced matrix.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 204 on page 850.

wr

contains the real part of the computed eigenvalues.

Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 204 on page 850.

wi

contains the imaginary part of the computed eigenvalues.

Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 204 on page 850.

w

contains the computed eigenvalues.

Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 204 on page 850.

vl

contains the left eigenvectors.

- If *jobvl* = 'V', the left eigenvectors are stored one after another in the columns of *vl*, in the same order as their eigenvalues.
  - For SGEEV, DGEEV, SGEEVX and DGEEVX:
    - If the *j*th eigenvalue is real, then the *j*th column of *vl* contains its eigenvector.
    - If the *j*th and (*j*+1)st eigenvalues form a complex conjugate pair, then the *j*th and (*j*+1)st columns of *vl* contain the real and imaginary parts of the eigenvector corresponding to the *j*th eigenvalue. The conjugate of this eigenvector is the eigenvector for the (*j*+1)st eigenvalue.
- If *jobvl* = 'N', *vl* is not referenced.

Returned as: an array of size (*ldvl*, *n*) containing numbers of the data type indicated in Table 204 on page 850.

vr

contains the right eigenvectors.

- If *jobvr* = 'V', the left eigenvectors are stored one after another in the columns of *vr*, in the same order as their eigenvalues.
  - For SGEEV, DGEEV, SGEEVX and DGEEVX:
    - If the *j*th eigenvalue is real, then the *j*th column of *vr* contains its eigenvector.
    - If the *j*th and (*j*+1)st eigenvalues form a complex conjugate pair, then the *j*th and (*j*+1)st columns of *vr* contain the real and imaginary parts of the eigenvector corresponding to the *j*th eigenvalue. The conjugate of this eigenvector is the eigenvector for the (*j*+1)st eigenvalue.
- If *jobvr* = 'N', *vr* is not referenced.

Returned as: an array of size (*ldvr*, *n*) containing numbers of the data type indicated in Table 204 on page 850.

ilo

has the following meaning:

If *balanc* = 'N', *ilo* = 1.

Otherwise, the value of *ilo* is determined when ***A*** is balanced.

The balanced $\boldsymbol{a}_{ij}$ = 0 if $i > j$ and j = 1, ..., (*ilo*-1) or i = (*ihi*+1), ..., n.

Returned as: an integer; 1 ≤ *ilo* ≤ n.

ihi

has the following meaning:

If *balanc* = 'N', *ihi* = n.

Otherwise, the value of *ihi* is determined when ***A*** is balanced.

The balanced $\boldsymbol{a}_{ij}$ = 0 if $i > j$ and j = 1, ..., (*ilo*-1) or i = (*ihi*+1), ..., n.

Returned as: an integer; 1 ≤ *ihi* ≤ n.

scale

contains the 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)
- $scale_j = d_j$, for j = *ilo*, ..., *ihi*
- $scale_j = p_j$, for j = (*ihi*+1), ..., n

Returned as: a one-dimensional array of (at least) length n containing numbers of the data type indicated in Table 204 on page 850.

abnrm

is the one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).

Returned as: a number of the data type indicated in Table 204 on page 850; *abnrm* ≥ 0.

rconde

contains the computed reciprocal condition numbers of the eigenvalues, where $rconde_j$ is the reciprocal condition number of the jth eigenvalue.

Returned as: an array of dimension n containing numbers of the data type indicated in Table 204 on page 850.

rcondv

contains the computed reciprocal condition numbers of the eigenvectors, where $rcondv_j$ is the reciprocal condition number of the jth right eigenvector.

Returned as: an array of dimension n containing numbers of the data type indicated in Table 204 on page 850.

work

is the work area used by this subroutine if *lwork* ≠ 0, where:

If *lwork* ≠ 0 and *lwork* ≠ -1, its size is (at least) of length *lwork*.

If *lwork* = -1, its size is (at least) of length 1.

Returned as: an area of storage, where:

If *lwork* ≥ 1 or *lwork* = -1, then $work_1$ is set to the optimal *lwork* value and contains numbers of the data type indicated in Table 204 on page 850.

Except for $work_1$, the contents of *work* are overwritten on return.

rwork

is a storage work area of size 2*n*.

Returned as: an area of storage containing numbers of the data type indicated in Table 204 on page 850.

iwork

is a storage work area of size 2*n*-2.

If *sense* = 'N' or 'E', *iwork* is not referenced by the subroutine.

Returned as: an integer array.

info

has the following meaning:

If *info* = 0, the subroutine completed successfully.

If *info* > 0:

- For SGEEV, DGEEV, CGEEV, ZGEEV, LAPACKE_sgeev, LAPACKE_dgeev, LAPACKE_cgeev, and LAPACKE_zgeev, if *info* > 0, the QR algorithm failed to compute all the eigenvalues, and no eigenvectors were computed; elements (*i*+1):*n* of the eigenvalue arrays contain eigenvalues that have converged.
- For SGEEVX, DGEEVX, CGEEVX, ZGEEVX, LAPACKE_sgeevx, LAPACKE_dgeevx, LAPACKE_cgeevx, and LAPACKE_zgeevx, if *info* > 0, the QR algorithm failed to compute all the eigenvalues, and no eigenvectors or reciprocal condition numbers were computed; elements 1:(*ilo*-1) and (*i*+1):*n* of the eigenvalue arrays contain eigenvalues that have converged.

Returned as:

- For SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX returned as: an integer; info ≥ 0.
- For LAPACKE_sgeev, LAPACKE_dgeev, LAPACKE_cgeev, LAPACKE_zgeev, LAPACKE_sgeevx, LAPACKE_dgeevx, LAPACKE_cgeevx, and LAPACKE_zgeevx, returned as an integer function value; info ≥ 0.

Notes

1. ***A***, *wr*, *wi*, *vl*, *vr*, *work*, *rwork*, and *iwork* must have no common elements; otherwise, results are unpredictable.
2. When you specify *jobvl* = 'N', you must specify a dummy argument for *vl*.
3. When you specify *jobvr* = 'N', you must specify a dummy argument for *vr*.
4. When you specify *sense* = 'N', you must specify a dummy argument for *rconde*.
5. When you specify *sense* = 'N' or 'E', you must specify dummy arguments for *rcondv* and *iwork*.
6. In your C program, the *ilo*, *ihi*, *abnrm*, *info* arguments must be passed by reference.
7. These subroutines accept lowercase letters for the *balanc*, *jobvl*, *jobvr*, and *sense* arguments.
8. The vectors and matrices used in the computation must have no common elements; otherwise, results are unpredictable.
9. For best performance, specify *lwork* = 0.

Function

These subroutines compute the following for a general matrix ***A*** :

- SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX compute:
  - eigenvalues
  - optionally, the right eigenvectors, left eigenvectors, or both

- SGEEVX, DGEEVX, CGEEVX, and ZGEEVX also compute:
  - optionally, the reciprocal condition numbers for the eigenvalues
  - optionally, the reciprocal condition numbers for the right eigenvectors

Computing eigenvalues only

The eigenvalues (only) of general matrix ***A*** are computed as follows:

1. If necessary, scale the general matrix ***A***.
2. Balance the general matrix ***A***.
3. Reduce the balanced matrix to an upper Hessenberg matrix using the following types of transformations:

   SGEEV, DGEEV, SGEEVX and DGEEVX
   Orthogonal similarity transformations

   CGEEV, DGEEV, CGEEVX and ZGEEVX
   Unitary similarity transformations
4. Compute the eigenvalues of the upper Hessenberg matrix using the multi-shift QR algorithm or the implicit double-shift QR algorithm.
5. If specified for SGEEVX, DGEEVX, CGEEVX, and ZGEEVX, compute reciprocal condition numbers.
6. If necessary, undo scaling.

Computing eigenvalues and right eigenvectors or left eigenvectors or both

The eigenvalues and right eigenvectors or left eigenvectors, or both, of general matrix ***A*** are computed as follows:

1. If necessary, scale the general matrix ***A***.
2. Balance the general matrix ***A***.
3. Reduce the balanced matrix to an upper Hessenberg matrix using the following types of transformations:

   SGEEV, DGEEV, SGEEVX and DGEEVX
   Orthogonal similarity transformations

   CGEEV, ZGEEV, CGEEVX and ZGEEVX
   Unitary similarity transformations
4. Accumulate the transformations.
5. Compute the eigenvalues of the upper Hessenberg matrix, and the appropriate eigenvectors of the corresponding balanced matrix, using the multi-shift QR algorithm or the implicit double-shift QR algorithm.
6. If specified, compute reciprocal condition numbers.
7. Undo balancing the eigenvectors; normalize the eigenvectors; and make the largest component real.
8. If necessary, undo scaling.

If n = 0, no computation is performed and the subroutine returns after doing some parameter checking.

For more information, see references [“15” on page 1254], [“16” on page 1254], [“67” on page 1257], [“68” on page 1257], and [“76” on page 1257].

Error conditions

Resource Errors

lwork = 0, and unable to allocate work area.

Computational Errors

1. Eigenvalue (*i*) failed to converge.

- For SGEEV, DGEEV, CGEEV and ZGEEV, elements (i+1):n of *wr* and *wi* contain eigenvalues that have converged. No eigenvectors have been computed.
- For SGEEVX, DGEEVX, CGEEVX, and ZGEEVX, elements 1:(*ilo*-1) and (i+1):n contain eigenvalues that have converged. No eigenvectors or condition numbers have been computed.
- The computational error message may occur multiple times with processing continuing after each error because the default for the number of allowable errors for error code 2153 is set to be unlimited in the ESSL error option table.

2. The subroutine computed the eigenvalues using multiple algorithms.
   - Performance may be degraded.
   - The computational error message may occur multiple times with processing continuing after each error because the default for the number of allowable errors for error code 2613 is set to be unlimited in the ESSL error option table.

Input-Argument Errors

1. *balanc* ≠ 'N', 'S', 'P', or 'B'
2. *jobvl* ≠ 'N', or 'V'
3. *jobvr* ≠ 'N', or 'V'
4. *sense* ≠ 'N', 'E', 'V', or 'B'
5. (*sense* = 'E' or *sense* = 'B') and (*jobvl* ≠ 'V' or *jobvr* ≠ 'V')
6. $n < 0$
7. $lda \leq 0$
8. $n > lda$
9. $ldvl \leq 0$
10. $ldvr \leq 0$
11. *jobvl* ≠ 'V' and $ldvl < n$
12. *jobvr* ≠ 'V' and $ldvr < n$
13. For SGEEV and DGEEV, if $lwork \neq 0$:
    - If *jobvl* = 'N' and *jobvr* = 'N', $lwork < \max(1, 3n)$.
    - If *jobvl* = 'V' or *jobvr* = 'V', $lwork < 4n$.

    For CGEEV and ZGEEV, if $lwork \neq 0$ and $lwork < \max(1,2n)$.
14. For SGEEVX and DGEEVX:
    - If *sense* = 'N' or 'E':
      – If *jobvl* = 'N' and *jobvr* = 'N', $lwork < \max(1, 2n)$.
      – If *jobvl* = 'V' or *jobvr* = 'V', $lwork < 3n$.
    - If *sense* = 'V' or 'B', $lwork < n(n + 6)$.

    For CGEEVX and ZGEEVX:
    - If *sense* = 'N' or 'E', $lwork < \max(1, 2n)$.
    - If *sense* = 'V' or 'B', $lwork < n^2 + 2n$.
15. $lwork \neq 0$ and $lwork \neq -1$ and *lwork* < the minimum required value.
16. The size of a work array is greater than 2147483647 when 32-bit integers are used.

Examples

Example 1

This example shows how to find the eigenvalues only of a long-precision real general matrix ***A*** of order 4, where:

- LDVL and LDVR are set to 1 to avoid an error condition.
- DUMMY1 is a placeholder for VL. VL is not used.
- DUMMY2 is a placeholder for VR. VR is not used.
- DUMMY3 is a placeholder for RCONDE. RCONDE is not used.
- DUMMY4 is a placeholder for RCONDV. RCONDV is not used.
- IDUMMY is a placeholder for IWORK. IWORK is not used.

Note:

1. This matrix is used in Example 5.5 in referenced text [“76” on page 1257].
2. Because *lwork* = 0, the subroutine dynamically allocates WORK.
3. On output, A has been overwritten.

Call Statement and Input:

```
          JOBVL JOBVR N   A LDA WR  WI    VL    LDVL  VR    LDVR WORK LWORK INFO
            |     |   |   |  |  |   |      |     |    |      |   |     |   |
CALL DGEEV( 'N',  'N', 4, A, 4, WR, WI, DUMMY1, 1, DUMMY2, 1, WORK,  0, INFO )
```

–or–

```
          BALANC JOBVL JOBVR SENSE  N  A  LDA WR  WI   VL   LDVL   VR  LDVR ILO  IHI
             |     |     |     |    |  |   |   |   |    |     |     |    |   |    |
CALL DGEEVX( 'N',  'N',  'N',  'N',  4, A,  4, WR, WI, DUMMY1, 1, DUMMY2, 1, ILO, IHI,

             SCALE  ABNRM  RCONDE  RCONDV  WORK LWORK IWORK  INFO
               |      |      |       |      |    |     |      |
             SCALE, ABNRM, DUMMY3, DUMMY4, WORK,  0, IDUMMY, INFO )
```

```
         | -2.0  2.0  2.0  2.0 |
A    =   | -3.0  3.0  2.0  2.0 |
         | -2.0  0.0  4.0  2.0 |
         | -1.0  0.0  0.0  5.0 |
```

Output:

```
         | 1.000000 |
WR   =   | 2.000000 |
         | 3.000000 |
         | 4.000000 |
```

```
         | 0.000000 |
WI   =   | 0.000000 |
         | 0.000000 |
         | 0.000000 |
```

```
         | 1.000000 |
SCALE =  | 1.000000 |
         | 1.000000 |
         | 1.000000 |
```

```
ILO     =   1
IHI     =   4
ABNRM   =  11.0
INFO    =   0
```

Example 2

For a long precision real general matrix ***A*** of order 4, this example shows the following, depending on the subroutine you are using:

- For DGEEV, this example shows how to find the eigenvalues and left and right eigenvectors of a general matrix ***A***.
- for DGEEVX, this example shows the eigenvalues, left and right eigenvectors, and reciprocal condition numbers for the eigenvalues and right eigenvectors of a balanced general matrix ***A*** of order 4.

Note:

1. This matrix is used in Example 5.5 in referenced text [“76” on page 1257].
2. `IWORK` is an integer work array of size 6.
3. On output, A has been overwritten by the Schur form of the balanced matrix.

Call Statement and Input:

```
           JOBVL JOBVR N  A LDA WR  WI  VL LDVL VR LDVR WORK LWORK INFO
             |     |   |  |  |   |   |   |   |   |   |    |    |     |
CALL DGEEV( 'V',  'V', 4, A, 4, WR, WI, VL,  1, VR,  1, WORK,  0,  INFO )
```

–or–

```
           BALANC JOBVL JOBVR SENSE N  A LDA WR  WI  VL LDVL VR LDVR ILO  IHI
             |      |     |     |   |  |  |   |   |   |   |   |   |   |    |
CALL DGEEVX( 'B',  'V',  'V',  'B', 4, A, 4, WR, WI, VL,  4, VR,  4, ILO, IHI,

             SCALE  ABNRM  RCONDE  RCONDV  WORK LWORK IWORK  INFO
               |      |      |       |      |    |      |     |
             SCALE, ABNRM, RCONDE, RCONDV, WORK,  0,  IWORK, INFO )
```

```
          | -2.0  2.0  2.0  2.0 |
A      =  | -3.0  3.0  2.0  2.0 |
          | -2.0  0.0  4.0  2.0 |
          | -1.0  0.0  0.0  5.0 |
```

Output:

- For DGEEV:

```
          | 1.000000  -6.949732   2.539925 -0.707107 |
A      =  | 0.000000   2.000000   1.060660 -0.717805 |
          | 0.000000   0.000000   3.000000 -0.780869 |
          | 0.000000   0.000000   0.000000  4.000000 |
```

- For DGEEVX:

```
          | 1.000000  -6.949732  -1.320184  0.103510 |
A      =  | 0.000000   2.000000  -2.415229  1.002262 |
          | 0.000000   0.000000   3.000000 -0.780869 |
          | 0.000000   0.000000   0.000000  4.000000 |
```

```
          | 1.000000 |
WR     =  | 2.000000 |
          | 3.000000 |
          | 4.000000 |
```

```
          | 0.000000 |
WI     =  | 0.000000 |
          | 0.000000 |
          | 0.000000 |
```

```
          | -0.707107  -0.408248   0.000000   0.000000 |
VL     =  |  0.707107   0.816497   0.408248   0.000000 |
          |  0.000000  -0.408248  -0.816497  -0.447214 |
          |  0.000000   0.000000   0.408248   0.894427 |
```

```
          | -0.730297   0.625543  -0.554700   0.500000 |
VR     =  | -0.547723   0.625543  -0.554700   0.500000 |
          | -0.365148   0.417029  -0.554700   0.500000 |
          | -0.182574   0.208514  -0.277350   0.500000 |
```

```
          |  0.087287 |
RCONDE =  |  0.053722 |
          |  0.096561 |
          |  0.282843 |
```

```
          |  0.448959 |
RCONDV =  |  0.244976 |
          |  0.289148 |
          |  0.508520 |
```

```
         |  1.000000 |
SCALE  = |  2.000000 |
         |  1.000000 |
         |  0.500000 |
```

```
ILO     =   1
IHI     =   4
ABNRM   =   7.5
INFO    =   0
```

Example 3

This example shows how to find the eigenvalues, left and right eigenvectors, and reciprocal condition numbers of a balanced long-precision real general matrix ***A*** of order 3.

Note:

1. This matrix is used in Example 5.4 in referenced text ["76" on page 1257].
2. IWORK is an integer work array of size 4.
3. On output, A has been overwritten by the Schur form of the balanced matrix.

Call Statement and Input:

```
           BALANC JOBVL JOBVR SENSE N  A  LDA  WR  WI  VL LDVL VR LDVR ILO  IHI
             |     |     |     |    |  |   |    |   |   |   |   |   |    |    |
CALL DGEEVX( 'B',  'V',  'V',  'B', 3, A,  3,  WR, WI, VL,  3, VR,  3, ILO, IHI,

             SCALE  ABNRM  RCONDE  RCONDV  WORK LWORK IWORK  INFO
               |      |       |       |      |    |     |      |
            SCALE, ABNRM, RCONDE, RCONDV, WORK,  0,  IWORK, INFO )
```

```
          |   8.0  -1.0  -5.0 |
A      =  |  -4.0   4.0  -2.0 |
          |  18.0  -5.0  -7.0 |
```

Output:

```
          | 2.000000  -6.928203  -13.435029 |
A      =  | 2.309401   2.000000  -10.206207 |
          | 0.000000   0.000000    1.000000 |
```

```
          | 2.000000 |
WR     =  | 2.000000 |
          | 1.000000 |
```

```
          |  4.000000 |
WI     =  | -4.000000 |
          |  0.000000 |
```

```
          | -0.877058   0.000000  -0.816497 |
VL     =  |  0.263117  -0.087706   0.408248 |
          |  0.350823   0.175412   0.408248 |
```

```
          |  0.316228  -0.316228   0.408248 |
VR     =  |  0.632456   0.000000   0.816497 |
          |  0.000000  -0.632456   0.408248 |
```

```
          |  0.301511 |
RCONDE =  |  0.301511 |
          |  0.192450 |
```

```
          |  1.671856 |
RCONDV =  |  1.671856 |
          |  1.174058 |
```

```
         |  0.500000 |
SCALE  = |  1.000000 |
         |  1.000000 |
```

```
ILO     =   1
IHI     =   3
ABNRM   =  19.0
INFO    =   0
```

Example 4

This example shows how to find the eigenvalues and right eigenvectors of a long-precision complex general matrix ***A*** of order 4, where:

- LDVL is set to 1 to avoid an error condition.

- DUMMY1 is a placeholder for VL. VL is not used.
- DUMMY2 is a placeholder for RCONDE. RCONDE is not used.
- DUMMY3 is a placeholder for RCONDV. RCONDV is not used.

Note:

1. This matrix is used in Example 6.5 in referenced text [“76” on page 1257].
2. On output, A has been overwritten by the Schur form of the balanced matrix.
3. RWORK is a real array of length 8.

Call Statement and Input:

```
             JOBVL JOBVR N   A LDA W    VL   LDVL VR LDVR WORK LWORK RWORK
INFO
               |     |   |   |  |  |     |    |   |   |    |    |      |
|
CALL ZGEEV( 'N',  'V', 4, A, 4, W, DUMMY1, 1, VR,  4, WORK, 0,   RWORK,
INFO )
```

–or–

```
           BALANC JOBVL JOBVR SENSE N  A LDA W   VL    LDVL VR LDVR ILO  IHI
              |     |     |     |   |  |  |  |    |       |   |   |    |    |
CALL ZGEEVX( 'N',  'N',  'V',  'N', 4, A, 4, W, DUMMY1,  1, VR,  4, ILO, IHI,

            SCALE  ABNRM  RCONDE  RCONDV  WORK LWORK RWORK  INFO
              |      |      |       |      |    |     |      |
            SCALE, ABNRM, DUMMY2, DUMMY3, WORK,  0,  RWORK, INFO )
```

```
        | ( 5.0, 9.0) ( 5.0,  5.0) (-6.0, -6.0) (-7.0, -7.0) |
A    =  | ( 3.0, 3.0) ( 6.0, 10.0) (-5.0, -5.0) (-6.0, -6.0) |
        | ( 2.0, 2.0) ( 3.0,  3.0) (-1.0,  3.0) (-5.0, -5.0) |
        | ( 1.0, 1.0) ( 2.0,  2.0) (-3.0, -3.0) ( 0.0,  4.0) |
```

Output:

- For ZGEEV:

```
A      =  | ( 2.0000, 6.0000) (-4.8694, -0.6574) (-6.7837, -7.6341) (17.1461, -0.1118) |
          | ( 0.0000, 0.0000) ( 4.0000,  8.0000) ( 0.1351,  1.7912) (-2.7907, -2.0982) |
          | ( 0.0000, 0.0000) ( 0.0000,  0.0000) ( 3.0000,  7.0000) (-3.0172,  0.1977) |
          | ( 0.0000, 0.0000) ( 0.0000,  0.0000) ( 0.0000,  0.0000) ( 1.0000,  5.0000) |
```

- For ZGEEVX:

```
          | ( 2.0000, 6.0000) (-1.1081, 4.9368) (-3.3663, 3.6542) (-19.9524, 4.0936) |
A      =  | ( 0.0000, 0.0000) ( 4.0000, 8.0000) ( 0.1597, 0.5962) ( -2.1519, 5.6785) |
          | ( 0.0000, 0.0000) ( 0.0000, 0.0000) ( 3.0000, 7.0000) (  0.8130, 4.9939) |
          | ( 0.0000, 0.0000) ( 0.0000, 0.0000) ( 0.0000, 0.0000) (  1.0000, 5.0000) |
```

```
          | ( 2.0000, 6.0000) |
W      =  | ( 4.0000, 8.0000) |
          | ( 3.0000, 7.0000) |
          | ( 1.0000, 5.0000) |
```

```
          | ( 0.3780, 0.0000) ( 0.5774, 0.0000) ( 0.5774, 0.0000) ( 0.7559, 0.0000) |
VR     =  | ( 0.7559, 0.0000) ( 0.5774, 0.0000) ( 0.5774, 0.0000) ( 0.3780, 0.0000) |
          | ( 0.3780, 0.0000) ( 0.5774, 0.0000) ( 0.0000, 0.0000) ( 0.3780, 0.0000) |
          | ( 0.3780, 0.0000) ( 0.0000, 0.0000) ( 0.5774, 0.0000) ( 0.3780, 0.0000) |
```

```
         |  1.000000 |
SCALE  = |  1.000000 |
         |  1.000000 |
         |  1.000000 |
```

```
ILO     =   1
IHI     =   4
ABNRM   =  29.5
INFO    =   0
```

Example 5

For a long precision complex general matrix ***A*** of order 4, this example shows the following, depending on the subroutine you are using:

- For ZGEEV, this example shows how to find the eigenvalues and left and right eigenvectors of a complex general matrix ***A***.
- for ZGEEVX, this example shows the eigenvalues, left and right eigenvectors, and reciprocal condition numbers for the eigenvalues and right eigenvectors of a balanced complex general matrix ***A*** of order 4.

Note:

1. This matrix is used in Example 6.5 in referenced text [“76” on page 1257].
2. RWORK is a real array of length 8.
3. On output, A has been overwritten by the Schur form of the balanced matrix.

Call Statement and Input:

```
              JOBVL  JOBVR N   A LDA W    VL   LDVL  VR LDVR WORK LWORK RWORK
INFO
                |      |   |   |  |  |      |    |    |   |    |    |       |
|
CALL ZGEEV( 'V',  'V', 4, A, 4, W, DUMMY1, 1, VR,  4, WORK, 0,   RWORK,
INFO )
```

–or–

```
           BALANC JOBVL JOBVR SENSE N  A LDA  W  VL LDVL VR LDVR ILO  IHI
              |     |     |     |   |  |  |   |   |   |   |   |   |    |
CALL ZGEEVX( 'P',  'V',  'V',  'B', 4, A, 4,  W, VL,  4, VR,  4, ILO, IHI,

            SCALE  ABNRM  RCONDE  RCONDV  WORK LWORK RWORK  INFO
              |      |       |       |      |    |     |     |
            SCALE, ABNRM, RCONDE, RCONDV, WORK,  0,  RWORK, INFO )
```

```
        | ( 5.0, 9.0) ( 5.0,  5.0) (-6.0, -6.0) (-7.0, -7.0) |
A    =  | ( 3.0, 3.0) ( 6.0, 10.0) (-5.0, -5.0) (-6.0, -6.0) |
        | ( 2.0, 2.0) ( 3.0,  3.0) (-1.0,  3.0) (-5.0, -5.0) |
        | ( 1.0, 1.0) ( 2.0,  2.0) (-3.0, -3.0) ( 0.0,  4.0) |
```

Output:

- For ZGEEV:

```
A      =  | ( 2.0000, 6.0000) (-4.8694, -0.6574) (-6.7837, -7.6341) (17.1461, -0.1118) |
          | ( 0.0000, 0.0000) ( 4.0000,  8.0000) ( 0.1351,  1.7912) (-2.7907, -2.0982) |
          | ( 0.0000, 0.0000) ( 0.0000,  0.0000) ( 3.0000,  7.0000) (-3.0172,  0.1977) |
          | ( 0.0000, 0.0000) ( 0.0000,  0.0000) ( 0.0000,  0.0000) ( 1.0000,  5.0000) |
```

- For ZGEEVX:

```
A      =  | ( 2.0000, 6.0000) (-1.1081, 4.9368) (-3.3663, 3.6542) (-19.9524, 4.0936) |
          | ( 0.0000, 0.0000) ( 4.0000, 8.0000) ( 0.1597, 0.5962) ( -2.1519, 5.6785) |
          | ( 0.0000, 0.0000) ( 0.0000, 0.0000) ( 3.0000, 7.0000) (  0.8130, 4.9939) |
          | ( 0.0000, 0.0000) ( 0.0000, 0.0000) ( 0.0000, 0.0000) (  1.0000, 5.0000) |
```

```
W      =  | ( 2.0000, 6.0000) |
          | ( 4.0000, 8.0000) |
          | ( 3.0000, 7.0000) |
          | ( 1.0000, 5.0000) |
```

```
VR     =  | ( 0.3780, 0.0000) ( 0.5774, 0.0000) ( 0.5774, 0.0000) ( 0.7559, 0.0000) |
          | ( 0.7559, 0.0000) ( 0.5774, 0.0000) ( 0.5774, 0.0000) ( 0.3780, 0.0000) |
          | ( 0.3780, 0.0000) ( 0.5774, 0.0000) ( 0.0000, 0.0000) ( 0.3780, 0.0000) |
          | ( 0.3780, 0.0000) ( 0.0000, 0.0000) ( 0.5774, 0.0000) ( 0.3780, 0.0000) |
```

```
VL     =  | (-0.5774, 0.0000) (-0.3780, 0.0000) (-0.3780, 0.0000) ( 0.0000, 0.0000) |
          | ( 0.0000, 0.0000) (-0.3780, 0.0000) (-0.3780, 0.0000) (-0.5774, 0.0000) |
          | ( 0.5774, 0.0000) ( 0.3780, 0.0000) ( 0.7559, 0.0000) ( 0.5774, 0.0000) |
          | ( 0.5774, 0.0000) ( 0.7559, 0.0000) ( 0.3780, 0.0000) ( 0.5774, 0.0000) |
```

```
RCONDE =  |   0.2182  |
          |   0.2182  |
          |   0.2182  |
          |   0.2182  |
```

```
RCONDV =  |   0.3089  |
          |   0.6450  |
          |   0.1770  |
          |   0.5504  |
```

```
SCALE  =  |  1.000000  |
          |  1.000000  |
          |  1.000000  |
          |  1.000000  |
```

```
ILO     =   1
IHI     =   4
ABNRM   =  29.5
INFO    =   0
```

Example 6

This example shows how to find the eigenvalues, left and right eigenvectors, and reciprocal condition numbers for the eigenvalues and right eigenvectors of a balanced long-precision complex general matrix ***A*** of order 4.

Note:

1. This matrix is used in Example 6.5 in referenced text [“76” on page 1257].
2. RWORK is a real array of length 8.
3. On output, A has been overwritten by the Schur form of the balanced matrix.

Call Statement and Input:

```
          BALANC JOBVL JOBVR SENSE N  A  LDA  W  VL LDVL VR LDVR ILO  IHI
             |     |     |     |   |  |   |   |   |   |   |   |   |    |
CALL ZGEEVX( 'B',  'V',  'V',  'B', 4, A, LDA, W, VL,  4, VR,  4, ILO, IHI,

           SCALE  ABNRM  RCONDE  RCONDV  WORK LWORK RWORK  INFO
             |      |      |       |      |     |     |      |
           SCALE, ABNRM, RCONDE, RCONDV, WORK,  0,  RWORK, INFO )
```

```
        | ( 5.0, 9.0) ( 5.0,  5.0) (-6.0, -6.0) (-7.0, -7.0) |
A    =  | ( 3.0, 3.0) ( 6.0, 10.0) (-5.0, -5.0) (-6.0, -6.0) |
        | ( 2.0, 2.0) ( 3.0,  3.0) (-1.0,  3.0) (-5.0, -5.0) |
        | ( 1.0, 1.0) ( 2.0,  2.0) (-3.0, -3.0) ( 0.0,  4.0) |
```

Output:

```
          | ( 2.0000, 6.0000) ( 0.2165, -4.9088) ( 6.7861, -7.6319) (-16.4572,  4.8125) |
A      =  | ( 0.0000, 0.0000) ( 4.0000,  8.0000) ( 0.1841,  1.7868) (  1.5401, -3.1335) |
          | ( 0.0000, 0.0000) ( 0.0000,  0.0000) ( 3.0000,  7.0000) ( -0.6773, -2.9469) |
          | ( 0.0000, 0.0000) ( 0.0000,  0.0000) ( 0.0000,  0.0000) (  1.0000,  5.0000) |
```

```
          | ( 2.0000, 6.0000) |
W      =  | ( 4.0000, 8.0000) |
          | ( 3.0000, 7.0000) |
          | ( 1.0000, 5.0000) |
```

```
          | ( 0.3780, 0.0000) ( 0.5774, 0.0000) ( 0.5774, 0.0000) ( 0.7559, 0.0000) |
VR     =  | ( 0.7559, 0.0000) ( 0.5774, 0.0000) ( 0.5774, 0.0000) ( 0.3780, 0.0000) |
          | ( 0.3780, 0.0000) ( 0.5774, 0.0000) ( 0.0000, 0.0000) ( 0.3780, 0.0000) |
          | ( 0.3780, 0.0000) ( 0.0000, 0.0000) ( 0.5774, 0.0000) ( 0.3780, 0.0000) |
```

```
          | (-0.5774, 0.0000) (-0.3780, 0.0000) (-0.3780, 0.0000) ( 0.0000, 0.0000) |
VL     =  | ( 0.0000, 0.0000) (-0.3780, 0.0000) (-0.3780, 0.0000) (-0.5774, 0.0000) |
          | ( 0.5774, 0.0000) ( 0.3780, 0.0000) ( 0.7559, 0.0000) ( 0.5774, 0.0000) |
          | ( 0.5774, 0.0000) ( 0.7559, 0.0000) ( 0.3780, 0.0000) ( 0.5774, 0.0000) |
```

```
          |   0.1633  |
RCONDE =  |   0.2108  |
          |   0.2108  |
          |   0.2887  |
```

```
          |   0.4507  |
RCONDV =  |   0.4293  |
          |   0.1317  |
          |   0.5114  |
```

```
          |  2.000000  |
SCALE  =  |  1.000000  |
          |  1.000000  |
          |  1.000000  |
```

```
ILO     =   1
IHI     =   4
ABNRM   =  27.3
INFO    =   0
```

SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, ZHEEVX, SSYEVR, DSYEVR, CHEEVR, and ZHEEVR (Eigenvalues, and Optionally, the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix)

Purpose

These subroutines compute eigenvalues, and optionally, the eigenvectors of a real symmetric matrix or a complex Hermitian matrix:

- SSYEV and DSYEV compute all eigenvalues, and optionally, the eigenvectors of real symmetric matrix ***A***, stored in lower or upper storage mode.
- CHEEV and ZHEEV compute all eigenvalues, and optionally, the eigenvectors of complex Hermitian matrix ***A***, stored in lower or upper storage mode.
- SSPEVX and DSPEVX compute selected eigenvalues, and optionally, the eigenvectors of real symmetric matrix ***A***, stored in lower- or upper-packed storage mode.
- CHPEVX and ZHPEVX compute selected eigenvalues, and optionally, the eigenvectors of complex Hermitian matrix ***A***, stored in lower- or upper-packed storage mode.
- SSYEVX and DSYEVX compute selected eigenvalues, and optionally, the eigenvectors of real symmetric matrix ***A***, stored in lower or upper storage mode.
- CHEEVX and ZHEEVX compute selected eigenvalues, and optionally, the eigenvectors of complex Hermitian matrix ***A***, stored in lower or upper storage mode.
- >|SSYEVR and DSYEVR compute selected eigenvalues, and optionally, the eigenvectors of real symmetric matrix ***A***, stored in lower or upper storage mode. |<
- >|CHEEVR and ZHEEVR compute selected eigenvalues, and optionally, the eigenvectors of complex Hermitian matrix ***A***, stored in lower or upper storage mode. |<

For SSYEV, DSYEV, CHEEV, and ZHEEV, eigenvalues are returned in vector ***w***, and eigenvectors are returned in matrix ***A***.

For SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, ZHEEVX, >|SSYEVR, DSYEVR, CHEEVR, and ZHEEVR,|< eigenvalues are returned in vector ***w***, and eigenvectors are returned in matrix ***Z***.

The formula is shown below:

$\boldsymbol{A}\boldsymbol{z} = w\boldsymbol{z}$

where $\boldsymbol{A} = \boldsymbol{A}^T$ or $\boldsymbol{A} = \boldsymbol{A}^H$.

Table 205: Data Types.

vl, vu, abstol, w, rwork	***A, Z, work***	**Subroutine**
Short-precision real	Short-precision real	SSYEV$^{\Delta}$ SSPEVX$^{\Delta}$ SSYEVX$^{\Delta}$ >|SSYEVR$^{\Delta}$|<
Long-precision real	Long-precision real	DSYEV$^{\Delta}$ DSPEVX$^{\Delta}$ DSYEVX$^{\Delta}$ >|DSYEVR$^{\Delta}$|<
Short-precision real	Short-precision complex	CHEEV$^{\Delta}$ CHPEVX$^{\Delta}$ CHEEVX$^{\Delta}$ >|CHEEVR$^{\Delta}$|<
Long-precision real	Long-precision complex	ZHEEV$^{\Delta}$ ZHPEVX$^{\Delta}$ ZHEEVX$^{\Delta}$ >|ZHEEVR$^{\Delta}$|<

$^{\Delta}$ LAPACK

Syntax

Table 206: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SSYEV | DSYEV (*jobz, uplo, n, a, lda, w, work, lwork, info*) CALL CHEEV | ZHEEV (*jobz, uplo, n, a, lda, w, work, lwork, rwork, info*) CALL SSPEVX | DSPEVX (*jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info*) CALL CHPEVX | ZHPEVX (*jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info*) CALL SSYEVX | DSYEVX (*jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work, lwork, iwork, ifail, info*) CALL CHEEVX | ZHEEVX (*jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work, lwork, rwork, iwork, ifail, info*) >|CALL SSYEVR | DSYEVR (*jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info*) |< >|CALL CHEEVR | ZHEEVR (*jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work, lwork, rwork, lrwork, iwork, liwork, info*)|<

Table 206: Syntax for supported computing languages. (continued)	
Language	**Syntax**

C and C++	ssyev | dsyev (*jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *info*);
	cheev | zheev (*jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *rwork*, *info*);
	sspevx | dspevx (*jobz*, *range*, *uplo*, *n*, *ap*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *iwork*, *ifail*, *info*);
	chpevx | zhpevx (*jobz*, *range*, *uplo*, *n*, *ap*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *rwork*, *iwork*, *ifail*, *info*);
	ssyevx | dsyevx (*jobz*, *range*, *uplo*, *n*, *a*, *lda*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *lwork*, *iwork*, *ifail*, *info*);
	cheevx | zheevx (*jobz*, *range*, *uplo*, *n*, *a*, *lda*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *lwork*, *rwork*, *iwork*, *ifail*, *info*);
	>|ssyevr | dsyevr (*jobz*, *range*, *uplo*, *n*, *a*, *lda*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *isuppz*, *work*, *lwork*, *iwork*, *liwork*, *info*); |<
	>|cheevr | zheevr (*jobz*, *range*, *uplo*, *n*, *a*, *lda*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *isuppz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*);|<

LAPACKE	info = LAPACKE_ssyev | LAPACKE_dsyev (*matrix_layout*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*);
	info = LAPACKE_cheev | LAPACKE_zheev (*matrix_layout*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*);
	info = LAPACKE_sspevx | LAPACKE_dspevx (*matrix_layout*, *jobz*, *range*, *uplo*, *n*, *ap*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *ifail*);
	info = LAPACKE_chpevx | LAPACKE_zhpevx (*matrix_layout*, *jobz*, *range*, *uplo*, *n*, *ap*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *ifail*);
	info = LAPACKE_ssyevx | LAPACKE_dsyevx (*matrix_layout*, *jobz*, *range*, *uplo*, *n*, *a*, *lda*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *ifail*);
	info = LAPACKE_cheevx | LAPACKE_zheevx (*matrix_layout*, *jobz*, *range*, *uplo*, *n*, *a*, *lda*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *ifail*);
	>|info = LAPACKE_ssyevr | LAPACKE_dsyevr (*matrix_layout*, *jobz*, *range*, *uplo*, *n*, *a*, *lda*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *isuppz*);|<
	>|info = LAPACKE_cheevr | LAPACKE_zheevr (*matrix_layout*, *jobz*, *range*, *uplo*, *n*, *a*, *lda*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *isuppz*); |<

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

jobz

indicates the type of computation to be performed, where:

If *jobz* = 'N', eigenvalues only are computed.

If *jobz* = 'V', eigenvalues and eigenvectors are computed.

Specified as: a single character; *jobz* = 'N' or 'V'.

range

indicates the type of computation to be performed, where:

If *range* = 'A', all eigenvalues are to be found.

If *range* = 'V', all eigenvalues in the interval [*vl*, *vu*] are to be found.

If *range* = 'I', the *il*-th through *iu*-th eigenvalues are to be found.

Specified as: a single character; *range* = 'A', 'V', or 'I'.

uplo

indicates whether the upper or lower triangular part of the matrix ***A*** is referenced, where:

If *uplo* = 'U', the upper triangular part is referenced.

If *uplo* = 'L', the lower triangular part is referenced.

Specified as: a single character; *uplo* = 'U' or 'L'.

n

is the order of matrix ***A*** used in the computation.

Specified as: an integer; $n \geq 0$.

ap

is the real symmetric or complex Hermitian matrix ***A*** of order *n*. It is stored in an array, referred to as ***AP***, where:

If *uplo* = 'U', it is stored in upper-packed storage mode.

If *uplo* = 'L', it is stored in lower-packed storage mode.

Specified as: one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 205 on page 868.

a

is the real symmetric or complex Hermitian matrix ***A*** of order *n*.

If *uplo* = 'U', it is stored in upper storage mode.

If *uplo* = 'L', it is stored in lower storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 205 on page 868.

lda

is the leading dimension of the array specified for ***A***.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

vl

has the following meaning:

If *range* = 'V', it is the lower bound of the interval to be searched for eigenvalues.

If *range* ≠ 'V', this argument is ignored.

Specified as: a number of the data type indicated in Table 205 on page 868. If *range* = 'V', $vl < vu$.

vu

has the following meaning:

If *range* = 'V', it is the upper bound of the interval to be searched for eigenvalues.

If *range* ≠ 'V', this argument is ignored.

Specified as: a number of the data type indicated in Table 205 on page 868. If *range* = 'V', $vl < vu$.

il

has the following meaning:

If *range* = 'I', it is the index (from smallest to largest) of the smallest eigenvalue to be returned.

If *range* ≠ 'I', this argument is ignored.

Specified as: an integer; if *range* = 'I', $1 \le il \le \max(1, n)$

iu

has the following meaning:

If *range* = 'I', it is the index (from smallest to largest) of the largest eigenvalue to be returned.

If *range* ≠ 'I', this argument is ignored.

Specified as: an integer; if range = 'I', $\min(il, n) \le iu \le n$.

abstol

is the absolute tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to:

$abstol + \varepsilon(\max(|a|, |b|))$

where ε is the machine precision. If *abstol* is less than or equal to zero, then ε(norm(***T***)) is used in its place, where norm(***T***) is the one-norm of the tridiagonal matrix obtained by reducing ***A*** to tridiagonal form. For most problems, this is the appropriate level of accuracy to request.

For certain strongly graded matrices, greater accuracy can be obtained in very small eigenvalues by setting *abstol* to a very small positive number. However, if *abstol* is less than:

$$\sqrt{unfl}$$

where *unfl* is the underflow threshold, then:

$$\sqrt{unfl}$$

is used in its place.

Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold—that is, (2**unfl*).

If *jobz* = 'V', setting *abstol* to *unfl* (the underflow threshold) yields the most orthogonal eigenvectors.

Notes:

1. The approximate values of the constants used for *abstol* are listed below:

   For SSPEVX, CHPEVX, SSYEVX, CHEEVX, SSYEVR, and CHEEVR

   ε
   0.119209289550781250E-06

   unfl
   0.1175494351E-37

   $\sqrt{unfl}$
   0.1084202172E-18

   For DSPEVX, ZHPEVX, DSYEVX, ZHEEVX, DSYEVR, and ZHEEVR

   ε
   0.222044604925031308E-15

   unfl
   0.222507385850720138E-307

$\sqrt{unfl}$

0.149166814624004135E-153

2. The value of *abstol* can affect which algorithm is used to compute the eigenvalues and eigenvectors. See Function.
3. >|For SSYEVR, CHEEVR, DSYEVR, and ZHEEVR, if high relative accuracy is important, set the *abstol* argument to the safe minimum value of `unfl`.|<

Specified as: a number of the data type indicated in Table 205 on page 868.

m
See On Return.

w
See On Return.

z
See On Return.

ldz
is the leading dimension of the array specified for ***Z***.

Specified as: an integer; *ldz* > 0 and, if *jobz* = 'V', *ldz* ≥ *n*.

isuppz
See On Return.

work
is a work area used by these subroutines.

For SSYEV, DSYEV, CHEEV, ZHEEV, SSYEVX, DSYEVX, CHEEVX, ZHEEVX, SSYEVR, DSYEVR, CHEEVR, and ZHEEVR
The size of the work area is specified by the *lwork* argument.

For SSPEVX and DSPEVX
The size of the work area is 8*n*.

For CHPEVX and ZHPEVX
The size of the work area is 2*n*.

Specified as: an area of storage; which contains numbers of the data type indicated in Table 205 on page 868.

lwork
is used to determine the size of the `WORK` array.

Specified as: an integer, where:

- If *lwork* = 0, the subroutine dynamically allocates the workspace needed for use during this computation. The dynamically allocated workspace will be freed prior to returning control to the calling program.
- If *lwork* = -1, a workspace query is assumed. The subroutine only calculates the optimal size of the `WORK` array and returns this value as the first entry of the `WORK` array.

Otherwise:

For SSYEV and DSYEV
lwork ≥ max(1, 3*n*-1).

For CHEEV and ZHEEV
lwork ≥ max(1, 2*n*-1).

For SSYEVX and DSYEVX
lwork ≥ max(1, 8*n*).

For CHEEVX and ZHEEVX
lwork ≥ max(1, 2*n*).

For SSYEVR and DSYEVR
lwork ≥ max(1, 26*n*).

For CHEEVR and ZHEEVR
lwork ≥ max(1, 2*n*).

Note: These formulas represent the minimum workspace required. For best performance, specify either *lwork* = -1 (to obtain the optimal size to use) or *lwork* = 0 (to direct the subroutine to dynamically allocate the workspace).

rwork
is a work area:

For CHEEV and ZHEEV
rwork is a storage work area of size max(1, 3*n*-2) .

For CHPEVX, ZHPEVX, CHEEVX, and ZHEEVX
rwork is a storage work area of size 7*n*.

For SSYEVR, DSYEVR, CHEEVR, and ZHEEVR
The size of the *rwork* storage work area is specified by the *lrwork* argument.

Specified as: an area of storage, which contains real numbers of the data type indicated in Table 205 on page 868.

lrwork
is used to determine the size of the RWORK array.

Specified as: an integer, where:

- If *lrwork* = 0, the subroutine dynamically allocates the workspace required during this computation. The dynamically allocated workspace is freed before returning the control to the calling program.
- If *lrwork* = -1, a workspace query is assumed. The subroutine calculates only the optimal size of the RWORK array and returns this value as the first entry of the RWORK array.
- Otherwise, for CHEEVR and ZHEEVR:

  lrwork ≥ max(1, 24*n*)

Note: These formulas represent the minimum workspace required. For best performance, specify either *lrwork* = -1 (to obtain the optimal size of workspace) or *lrwork* = 0 (to direct the subroutine to dynamically allocate the workspace).

iwork
is a work area.

For SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSYPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, and ZHEEVX
iwork is a storage work area of size 5*n*.

For SSYEVR, DSYEVR, CHEEVR, and ZHEEVR
The size of the *iwork* storage area is specified by the *liwork* argument.

Specified as: an area of storage containing integers.

liwork
is used to determine the size of the IWORK array.

Specified as: an integer, where:

- If *liwork* = 0, the subroutine dynamically allocates the workspace that is required during this computation. The dynamically allocated workspace is freed before returning the control to the calling program.
- If *liwork* = -1, a workspace query is assumed. The subroutine only calculates the optimal size of the IWORK array and returns this value as the first entry of the IWORK array.

Otherwise, for SSYEVR, DSYEVR, CHEEVR, and ZHEEVR:

If *liwork* > 0, *liwork* ≥ max(1, 10*n*).

Note: These formulas represent the minimum workspace required. For best performance, specify either *liwork* = -1 (to obtain the optimal size of workspace) or *liwork* = 0 (to direct the subroutine to dynamically allocate the workspace).

ifail

See On Return.

On Return

ap

On exit, the matrix ***A*** is overwritten by values generated during the reduction to tridiagonal form.

If *uplo* = 'U', the diagonal and first superdiagonal of the tridiagonal matrix ***T*** overwrite the corresponding elements of ***A***.

If *uplo* = 'L', the diagonal and first subdiagonal of ***T*** overwrite the corresponding elements of ***A***.

Returned as: a one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 205 on page 868.

a

For SSYEV, DSYEV, CHEEV, and ZHEEV

On exit:

If *jobz* = 'V', the columns of *a* contain the orthonormal eigenvectors of the matrix ***A*** corresponding to the computed eigenvalues, with the *i*-th column of ***A*** holding the eigenvector associated with *w*(*i*).

If *jobz* = 'N':

- If *uplo* = 'U', the leading *n* by *n* upper triangular part of ***A*** is overwritten.
- If *uplo* = 'L', the leading *n* by *n* lower triangular part of ***A*** is overwritten.

For SSYEVX, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, DSYEVX, CHEEVX, and ZHEEVX

On exit, the matrix ***A*** is overwritten by values generated during the reduction to tridiagonal form.

If *uplo* = 'U', the diagonal and first superdiagonal of the tridiagonal matrix ***T*** overwrite the corresponding elements of ***A***.

If *uplo* = 'L', the diagonal and first subdiagonal of ***T*** overwrite the corresponding elements of ***A***.

For SSYEVR, DSYEVR, CHEEVR, and ZHEEVR

On exit, the matrix ***A*** is destroyed.

Returned as: an array of dimension *lda* by (at least) *n*, containing numbers of the data type indicated in Table 205 on page 868.

m

is the number of eigenvalues found.

Returned as: an integer; $0 \le m \le n$.

w

For SSYEV, DSYEV, CHEEV, and ZHEEV

is the vector ***w*** of length *n*, which contains the computed eigenvalues in ascending order.

For SSYEVX, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, DSYEVX, CHEEVX, ZHEEVX, SSYEVR, DSYEVR, CHEEVR, and ZHEEVR

is the vector ***w***, which contains the computed eigenvalues in ascending order in the first *m* elements of *w*.

Returned as: a one-dimensional array of (at least) length *n*, which contains numbers of the data type indicated in Table 205 on page 868.

z

has the following meaning, where:

If *jobz* = 'N', *z* is ignored.

If *jobz* = 'V', the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix ***A*** corresponding to the computed 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*.

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. Supplying *n* as upper bound is always safe.

Returned as: an *ldz* by (at least) max(1, *m*) array, containing numbers of the data type indicated in Table 205 on page 868.

isuppz

is a work area used by these subroutines.

If *jobz* = 'V' and *range* = 'A' or 'I' and *iu*-*il* = *n*-1, then the *isuppz* array contains the indices that indicate the nonzero elements in the matrix ***Z***. The i^{th} eigenvector is nonzero only in elements *isuppz*(2**i*-1) through *isuppz*(2**i*).

If *jobz* = 'N' or *range* = 'V' or 'I' and *iu*-*il* ≠ *n*-1, then the *isuppz* array is not referenced.

Returned as: an integer array of dimension 2*max(1, *m*).

Notes:

1. The *isuppz* array is set only when you want to compute all eigenvalues and eigenvectors and conform to IEEE 754 standard.
2. The *isuppz* array contains nonzero elements in the matrix ***Z*** before applying orthogonal matrix, which is used in reduction of the matrix to tridiagonal form. It might not be consistent with the nonzero elements in the final output of the eigenvectors in matrix ***Z***.
3. Ensure that at least 2*max(1, *m*) columns are specified in the *isuppz* array. The exact value of *m* is not known in advance and an upper bound must be used. Supplying 2*max(1, *n*) columns is always safe.

work

is a work area used by these subroutines.

Returned as: an area of storage where:

If *lwork* = -1, *work* is (at least) of length 1 and $work_1$ contains the calculated optimal size of the `WORK` array.

If *lwork* ≠ -1 and *lwork* ≠ 0, *work* is (at least) of length *lwork* and $work_1$ contains the value specified for *lwork*.

Except for $work_1$, the contents of *work* are overwritten on return.

rwork

is a work area used by these subroutines.

For CHEEVR and ZHEEVR

Returned as: an area of storage where:

- If *lrwork* = -1, the *rwork* array is (at least) of length 1 and the $rwork_1$ entry contains the calculated optimal size of the `RWORK` array.
- If *lrwork* ≠ -1 and *lrwork* ≠ 0, then the *rwork* array is (at least) of length *lrwork* and the $rwork_1$ entry contains the value specified for the *lrwork* argument.

Except for the $rwork_1$ value, the contents of the *rwork* array are overwritten on return.

iwork

For SSYEVR, DSYEVR, CHEEVR, and ZHEEVR

Returned as: an area of storage where:

- If *liwork* = -1, then the *iwork* array is (at least) of length 1 and the $iwork_1$ entry contains the calculated optimal size of the `IWORK` array.
- If *liwork* ≠ -1 and *liwork* ≠ 0, then the *iwork* array is (at least) of length *liwork* and the $iwork_1$ entry contains the value specified for the *liwork* argument.

Except for the $iwork_1$ value, the contents of the *iwork* array are overwritten on return.

ifail

has the following meaning:

If *jobz* = 'N', *ifail* is ignored.

If *jobz* = 'V':

- If *info* = 0, the first *m* elements of *ifail* are zero.
- If *info* > 0, *ifail* contains the indices of the eigenvectors that failed to converge.

Returned as: an array of length *n*, containing integers.

info

has the following meaning:

For SSYEV, DSYEV, CHEEV, ZHEEV, LAPACKE_ssyev, LAPACKE_dsyev, LAPACKE_cheev, and LAPACKE_zheev

If *info* = 0, then the algorithm converged. This indicates a normal exit.

If *info* = *i*, the algorithm failed to converge. *i* off-diagnoal elements of an intermediate tridiagonal form did not converge to zero.

For SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, ZHEEVX, LAPACKE_sspevx, LAPACKE_dspevx, LAPACKE_chpevx, LAPACKE_zhpevx, LAPACKE_ssyevx, LAPACKE_dsyevx, LAPACKE_cheevx, and LAPACKE_zheevx

If *info* = 0, then all eigenvectors converged. This indicates a normal exit.

If *info* = *i*, then *i* eigenvectors failed to converge.

For SSYEVR, DSYEVR, CHEEVR, ZHEEVR, LAPACKE_ssyevr, LAPACKE_dsyevr, LAPACKE_cheevr, and LAPACKE_zheevr

If *info* = 0, all eigenvectors converged. This value indicates a normal exit.

If *info* > 0, the algorithm failed to converge.

- If *jobz* = 'N' and *info* = 1, bisection failed to converge.
- If *jobz* = 'N' and *info* = 2, not all of the eigenvalues were found.
- If *jobz* = 'N' and *info* = 3, bisection failed to converge and not all of the eigenvalues were found.
- If *jobz* = 'N' and *info* = 4, Gershgorin interval that was initially used was too small. No eigenvalues were computed.
- If *jobz* = 'V' and if *info* = *i*, *i* eigenvectors failed to converge.

Returned as:

- For SSYEV, DSYEV, CHEEV, ZHEEV, SSYEVX, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, DSYEVX, CHEEVX, ZHEEVX, >|SSYEVR, DSYEVR, CHEEVR, and ZHEEVR:|<

  An integer; *info* ≥ 0.
- For LAPACKE_ssyev, LAPACKE_dsyev, LAPACKE_cheev, LAPACKE_zheev, LAPACKE_ssyevx, LAPACKE_sspevx, LAPACKE_dspevx, LAPACKE_chpevx, LAPACKE_zhpevx, LAPACKE_dsyevx, LAPACKE_cheevx, LAPACKE_zheevx, LAPACKE_ssyevr, LAPACKE_dsyevr, LAPACKE_cheevr, and LAPACKE_zheevr:

  An integer function value; *info* ≥ 0.

Notes:

1. This subroutine accepts lowercase letters for the *jobz*, *range*, and *uplo* arguments.

2. In your C program, the arguments *info* and *m* must be passed by reference.
3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.
4. ***A***, ***Z***, *w*, >|*isuppz*|<, *ifail*, *work*, *rwork*, *iwork* must have no common elements; otherwise, results are unpredictable.
5. For a description of how real symmetric matrices are stored in lower- or upper-packed storage mode, see "Lower-Packed Storage Mode" on page 80 or "Upper-Packed Storage Mode" on page 82, respectively.

   For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see "Complex Hermitian Matrix" on page 84.
6. >|For best performance, specify *lwork* = 0, *lrwork* = 0, and *liwork* = 0.|<

Function

For SSYEV, DSYEV, CHEEV, and ZHEEV

These subroutines compute all eigenvalues and, optionally, the eigenvectors of a real symmetric or complex Hermitian matrix ***A***, stored in lower or upper storage mode. (If *n* = 0, the subroutine returns after completing parameter checking.)

The computation involves the following steps:

1. If necessary, scale the matrix ***A***.
2. Reduce the matrix to real symmetric tridiagonal form.
3. Compute all eigenvalues and, optionally, the eigenvectors.
   - If *jobz* = 'N', compute all the eigenvalues of the symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm.
   - Otherwise, for *jobz* = 'V', compute all eigenvalues and eigenvectors of the symmetric tridiagonal matrix using the implicit QL or QR method.
4. Rescale eigenvalues appropriately if matrix ***A*** was scaled.

For SSYEVX, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, DSYEVX, CHEEVX, and ZHEEVX

These subroutines compute selected eigenvalues and, optionally, the eigenvectors of a real symmetric or complex Hermitian matrix ***A***, stored in lower-packed or upper-packed storage mode or in lower or upper storage mode. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues. (If *n* = 0, the subroutine returns after completing parameter checking.)

The computation involves the following steps:

1. If necessary, scale the matrix ***A***.
2. Reduce the matrix to real symmetric tridiagonal form.
3. Compute the selected eigenvalues and, optionally, the eigenvectors. The algorithm used depends on the value specified for *abstol* and whether or not all eigenvalues are requested.
   a. If *abstol* ≤ 0 and if all eigenvalues were requested (that is, *range* = 'A' or *range* = 'I' with *il* = 1 and *iu* = *n*), do the following:
      - If *jobz* = 'N', compute all the eigenvalues of the symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm.
      - Otherwise, for *jobz* = 'V', compute all eigenvalues and eigenvectors of the symmetric tridiagonal matrix using the implicit QL or QR method.

   b. Otherwise, if *abstol* > 0, if a subset of the eigenvalues was requested via *range* = 'I' or *range* = 'V', or if the previous step failed to compute all eigenvalues, do the following:

        1) Compute the requested eigenvalues using bisection. If *abstol* ≤ 0, then ε(norm(T)) is used in its place, where norm(T) is the one-norm of the tridiagonal matrix obtained by reducing ***A*** to tridiagonal form.
        2) If the eigenvectors were also requested, compute the eigenvectors using inverse iteration.

4. Rescale eigenvalues appropriately if matrix ***A*** was scaled.

For SSYEVR, DSYEVR, CHEEVR, and ZHEEVR

These subroutines compute selected eigenvalues, and optionally, the eigenvectors of a real symmetric or complex Hermitian matrix ***A***, stored in lower or upper storage mode. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues. (If *n* = 0, the subroutine returns after checking the parameters.)

The computation involves the following steps:

1. If necessary, scale the matrix ***A***.
2. Reduce the matrix to real symmetric tridiagonal form.
3. Compute the selected eigenvalues, and optionally, the eigenvectors. The algorithm used depends on the value specified for *abstol* and whether or not all eigenvalues are requested.
    a. If *abstol* ≤ 0 and if all eigenvalues were requested (that is, *range* = 'A' or *range* = 'I', *il* = 1, and *iu* = *n*), complete one of the following steps:
        - If *jobz* = 'N', compute all the eigenvalues of the symmetric tridiagonal matrix by using the Pal-Walker-Kahan variant of the QL or QR algorithm.
        - If *jobz* = 'V', compute all eigenvalues and eigenvectors of the symmetric tridiagonal matrix by using the Multiple Relatively Robust Representations method.

    b. Otherwise, if *abstol* > 0, if a subset of the eigenvalues was requested by using *range* = 'I' or *range* = 'V', or if the previous step failed to compute all eigenvalues, complete the following steps:
        1) Compute the requested eigenvalues using bisection. If *abstol* ≤ 0, then ε(norm(T)) is used in the place of *abstol*, where norm(T) is the one-norm of the tridiagonal matrix obtained by reducing matrix ***A*** to tridiagonal form.
        2) If the eigenvectors were also requested, compute the eigenvectors by using inverse iteration.
4. Rescale eigenvalues appropriately if matrix ***A*** was scaled.

For more information on these methods, see references [“9” on page 1253], [“35” on page 1255], [“37” on page 1255], and [“76” on page 1257].

Error conditions

Resource Errors

1. *lwork* = 0, and unable to allocate work area.
2. >|*lrwork* = 0, and unable to allocate work area.|<
3. >|*liwork* = 0, and unable to allocate work area.|<

Computational Errors

1. Bisection failed to converge for some eigenvalues. The eigenvalues may not be as accurate as the absolute and relative tolerances.
2. The number of eigenvalues computed does not match the number of eigenvalues requested.
3. No eigenvalues were computed because the Gershgorin interval initially used was incorrect.
4. Some eigenvectors failed to converge. The indices are stored in ***ifail***.

5. The subroutine computed the eigenvalues using multiple algorithms. Performance may be degraded.
6. If *info* = *i*, the algorithm failed to converge. *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Note: The default for the number of allowable errors for error conditions 2154, 2155, 2156, 2157, 2162, and 2613 is set to be unlimited in the ESSL error option table; therefore, each computational error message may occur multiple times with processing continuing after each error.

Input-Argument Errors

1. *jobz* ≠ 'N' or 'V'
2. *range* ≠ 'A', 'V', or 'I'
3. *uplo* ≠ 'U' or 'L'
4. *n* < 0
5. *range* = 'V', *n* > 0, and *vu* ≤ *vl*
6. *range* = 'I' and (*il* < 1 or *il* > max(1, *n*))
7. *range* = 'I' and (*iu* < min(*n*, *il*) or *iu* > *n*)
8. *lda* ≤ 0
9. *lda* < *n*
10. *ldz* ≤ 0
11. *jobz* = 'V' and *ldz* < *n*
12. *lwork* ≠ 0 and *lwork* ≠ -1 and *lwork* < the minimum required value
13. *lrwork* ≠ 0, *lrwork* ≠ -1, and *lrwork* < the minimum required value
14. *liwork* ≠ 0, *liwork* ≠ -1, and *liwork* < the minimum required value

Examples

Example 1

This example shows how to find the eigenvalues only of a real symmetric matrix ***A*** of order 4, stored in lower-packed storage mode.

Note: This matrix is Example 4.1 in referenced text [“76” on page 1257].

Matrix ***A*** is:

```
                   ┌                     ┐
                   |  5.0 4.0 1.0 1.0 |
                   |  4.0 5.0 1.0 1.0 |
                   |  1.0 1.0 4.0 2.0 |
                   |  1.0 1.0 2.0 4.0 |
                   └                     ┘
```

Call Statement and Input:

```
            JOBZ RANGE UPLO  N  AP  VL  VU  IL  IU ABSTOL M  W  Z  LDZ  WORK  IWORK  IFAIL  INFO
              |    |     |   |   |   |   |    |   |    |    |  |  |   |     |      |      |      |
CALL DSPEVX ('N', 'A',  'L', 4, AP, VL, VU, IL, IU, -1.0, M, W, Z,  4,  WORK, IWORK, IFAIL, INFO)

AP       =  (5.0, 4.0, 1.0, 1.0, 5.0, 1.0, 1.0, 4.0, 2.0, 4.0)
```

Output:

```
M    =    4
```

```
AP   =  (5.000000, -4.242641, 0.121320, 0.121320, 6.0000000, 1.414214, 0.414214, 5.000000, 0.000000,
2.000000)
```

```
             |  1.000000 |
W    =       |  2.000000 |
             |  5.000000 |
             | 10.000000 |
```

```
Z is not used when JOBZ = 'N'.
```

```
IFAIL is not used when JOBZ = 'N'.
```

```
INFO = 0
```

Example 2

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix ***A*** of order 4, stored in upper-packed storage mode. This example also illustrates the use of the *il* and *iu* arguments when *range* = 'I'.

Note: This matrix is Example 4.1 in referenced text [“76” on page 1257].

Matrix ***A*** is the same matrix used for DSPEVX in Example 1.

Call Statement and Input:

```
            JOBZ RANGE UPLO  N  AP  VL  VU  IL  IU ABSTOL M  W  Z  LDZ  WORK  IWORK  IFAIL  INFO
              |    |    |    |   |   |   |   |   |    |   |  |  |   |     |     |      |      |
CALL DSPEVX ('V', 'I',  'U', 4, AP, VL, VU,  1,  3, -1.0, M, W, Z,  4,  WORK, IWORK, IFAIL, INFO)

AP      =  (5.0, 4.0, 5.0, 1.0, 1.0, 4.0, 1.0, 1.0, 2.0, 4.0)
```

Output:

```
M    =    3
```

```
AP   =   (1.000000, 0.000000, 6.000000, 0.414214, 2.828427, 7.000000, 0.224745, 0.224725, -2.449490, 4.000000)
```

```
             |  1.000000 |
W    =       |  2.000000 |
             |  5.000000 |
             |      .    |
```

```
             |  0.707107  0.000000 -0.316228 |
Z    =       | -0.707107  0.000000 -0.316228 |
             |  0.000000 -0.707107  0.632456 |
             |  0.000000  0.707107  0.632456 |
```

```
IFAIL = (0,0,0,.)
INFO  = 0
```

Example 3

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix ***A*** of order 4, stored in upper-packed storage mode. This example also illustrates the use of the *vl* and *vu* arguments when *range* = 'V'.

Note: This matrix is Example 4.1 in Reference [“76” on page 1257].

Matrix ***A*** is the same matrix used for DSPEVX in Example 1.

Call Statement and Input:

```
            JOBZ RANGE UPLO  N  AP  VL    VU   IL IU ABSTOL M  W  Z  LDZ  WORK  IWORK  IFAIL  INFO
              |    |     |   |   |   |     |    |  |    |   |  |  |   |     |      |      |     |
CALL DSPEVX ('V', 'V',  'U', 4, AP, 3.0, 11.0, 0, 0, -1.0, M, W, Z,  4,  WORK, IWORK, IFAIL, INFO)

AP       =  (5.0, 4.0, 5.0, 1.0, 1.0, 4.0, 1.0, 1.0, 2.0, 4.0)
```

Output:

```
M    =    2
```

```
AP   =   (1.000000, 0.000000, 6.000000, 0.414214, 2.828427, 7.000000, 0.224745, 0.224725, -2.449490, 4.000000)
```

```
         |  5.000000 |
W    =   | 10.000000 |
         |      .    |
         |      .    |
```

```
         | -0.316228 -0.632456 |
Z    =   | -0.316228 -0.632456 |
         |  0.632456 -0.316228 |
         |  0.632456 -0.316228 |
```

```
IFAIL = (0,0,.,.)
INFO  = 0
```

Example 4

This example shows how to find the eigenvalues only of a complex Hermitian matrix ***A*** of order 3, stored in lower-packed storage mode.

Note: This matrix is Example 6.3 in referenced text [“76” on page 1257].

Matrix ***A*** is:

```
                 |  (2.0,  0.0) (0.0, 1.0) (0.0, 0.0) |
                 |  (0.0, -1.0) (2.0, 0.0) (0.0, 0.0) |
                 |  (0.0,  0.0) (0.0, 0.0) (3.0, 0.0) |
```

Call Statement and Input:

```
            JOBZ RANGE UPLO  N  AP  VL  VU  IL  IU ABSTOL M  W  Z  LDZ  WORK  RWORK  IWORK  IFAIL  INFO
              |    |     |   |   |   |   |    |   |    |   |  |  |   |     |     |      |      |      |
CALL ZHPEVX ('N', 'A',  'L', 3, AP, VL, VU, IL, IU, -1.0, M, W, Z,  3,  WORK, RWORK, IWORK, IFAIL, INFO)

AP   =  ((2.0, . ), (0.0, -1.0), (0.0, 0.0), (2.0, . ), (0.0, 0.0), (3.0, . ))
```

Output:

```
M    =    3
```

```
AP   =  ((2.0, 0.0), (-1.0, 0.0), (0.0, 0.0), (2.0, 0.0), (0.0, 0.0), (3.0, 0.0))
```

```
         |  1.000000 |
W    =   |  3.000000 |
         |  3.000000 |
```

```
Z is not used when JOBZ = 'N'.
```

```
IFAIL is not used when JOBZ = 'N'.
```

```
INFO = 0
```

Example 5

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix ***A*** of order 3, stored in upper-packed storage mode. This example also illustrates the use of the *il* and *iu* arguments when *range* = 'I'.

Note: This matrix is Example 6.3 in referenced text [“76” on page 1257].

Matrix ***A*** is the same matrix used for ZHPEVX in Example 4.

Call Statement and Input:

```
             JOBZ RANGE UPLO  N  AP  VL  VU  IL  IU ABSTOL M  W  Z  LDZ  WORK  RWORK  IWORK  IFAIL  INFO
               |    |     |   |   |   |   |   |   |     |   |  |  |   |     |     |      |      |      |
CALL ZHPEVX ('V', 'I',  'U', 3, AP, VL, VU,  1,  2, -1.0, M, W, Z,  3,  WORK, RWORK, IWORK, IFAIL, INFO)

AP   =  ((2.0, . ), (0.0, 1.0), (2.0, . ), (0.0, 0.0), (0.0, 0.0), (3.0, . ))
```

Output:

```
M    =    2
```

```
AP   =  ((2.0, 0.0), (-1.0, 0.0), (2.0, 0.0), (0.0, 0.0), (0.0, 0.0), (3.0, 0.0))
```

```
         |  1.000000 |
W    =   |  3.000000 |
         |      .    |
```

```
         | (0.0000, -0.7071), (0.0000, 0.1591) |
Z    =   | (0.7071,  0.0000), (0.1591, 0.0000) |
         | (0.0000,  0.0000), (0.9744, 0.0000) |
```

```
IFAIL = (0,0,.)
INFO = 0
```

Example 6

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix ***A*** of order 3, stored in upper-packed storage mode. This example also illustrates the use of the *vl* and *vu* arguments when *range* = 'V'.

Note: This matrix is Example 6.3 in referenced text [“76” on page 1257].

Matrix ***A*** is the same matrix used for ZHPEVX in Example 4.

Call Statement and Input:

```
             JOBZ RANGE UPLO  N  AP   VL   VU  IL  IU  ABSTOL M  W  Z  LDZ  WORK  RWORK  IWORK  IFAIL  INFO
               |    |     |   |   |    |    |   |   |     |   |  |  |   |    |     |      |      |      |
CALL ZHPEVX ('V', 'V',  'U', 3, AP, 2.0, 4.0, IL, IU,  -1.0, M, W, Z,  3,  WORK, RWORK, IWORK, IFAIL, INFO)

AP      =  ((2.0, . ), (0.0, 1.0), (2.0, . ), (0.0, 0.0), (0.0, 0.0), (3.0, . ))
```

Output:

```
M    =    2
```

```
AP   =  ((2.0, 0.0), (-1.0, 0.0), (2.0, 0.0), (0.0, 0.0), (0.0, 0.0), (3.0, 0.0))
```

```
        |  3.000000 |
W    =  |  3.000000 |
        |      .    |
```

```
        | ( 0.0000, -0.6634), ( 0.0000, -0.2447) |
Z    =  | (-0.6634,  0.0000), (-0.2447,  0.0000) |
        | (-0.3460,  0.0000), ( 0.9382,  0.0000) |
```

```
IFAIL = (0,0,.)
INFO = 0
```

Example 7

This example shows how to find the eigenvalues only of a symmetric matrix ***A*** of order 4, where:>|

- DUMMY1 is a placeholder for VL. VL is not used.
- DUMMY2 is a placeholder for VU. VU is not used.
- DUMMY3 is a placeholder for IL. IL is not used.
- DUMMY4 is a placeholder for IU. IU is not used.

|<

Notes:

1. This matrix is Example 4.1 in referenced text [“76” on page 1257].
2. Because *lwork* = 0, the subroutine dynamically allocates WORK.
3. >|Because *lrwork* = 0, the subroutine dynamically allocates RWORK.|<
4. >|Because *liwork* = 0, the subroutine dynamically allocates IWORK.|<

Matrix ***A*** is the same matrix used for DSPEVX in Example 1.

Call Statement and Input:

```
           JOBZ UPLO  N  A  LDA W  WORK LWORK INFO
             |    |   |  |   |  |   |     |     |
CALL DSYEV ('N', 'L', 4, A,  4, W, WORK,  0,  INFO)
```

–or–

```
            JOBZ RANGE UPLO  N  A  LDA  VL       VU      IL     IU    ABSTOL M  W  Z LDZ WORK LWORK IWORK  IFAIL  INFO
              |    |     |   |  |   |   |        |        |      |       |   |  |  |  |    |    |     |      |      |
CALL DSYEVX ('N', 'A',  'L', 4, A,  4, DUMMY1, DUMMY2, DUMMY3, DUMMY4, -1.0, M, W, Z, 4, WORK,  0,  IWORK, IFAIL, INFO)
```

–or–

>|

```
            JOBZ RANGE UPLO N  A  LDA  VL      VU      IL      IU    ABSTOL  M  W  Z LDZ ISSUPZ WORK LWORK IWORK LIWORK INFO
              |    |    |   |  |   |   |        |       |       |      |     |  |  |  |    |     |      |     |     |     |
CALL DSYEVR ('N', 'A', 'L', 4, A,  4, DUMMY1, DUMMY2, DUMMY3, DUMMY4, -1.0,  M, W, Z, 4, ISSUPZ, WORK,  0,  IWORK, 0, INFO)
```

|<

```
A     =  |  5.0   .    .    .  |
         |  4.0  5.0   .    .  |
         |  1.0  1.0  4.0   .  |
         |  1.0  1.0  2.0  4.0 |
         └
```

Output:

```
M     =    4
```

- For DSYEV, A is overwritten on output.
- For DSYEVX:

  ```
  A     =  |  5.000000     .        .        .     |
           | -4.242641 6.000000     .        .     |
           |  0.121320 1.414214 5.000000     .     |
           |  0.121320 1.414214 0.000000 2.000000 |
           └
  ```

- >|For DSYEVR, A is overwritten on output. |<

```
W     =  |  1.000000 |
         |  2.000000 |
         |  5.000000 |
         | 10.000000 |
         └
```

Z is not used when `JOBZ = 'N'`.

For DSYEVX, `IFAIL` is not used when `JOBZ = 'N'`.

>|For DSYEVR, `ISUPPZ` is not used when `JOBZ = 'N'`.|<

```
INFO = 0
```

Example 8

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix ***A*** of order 4 stored in upper storage mode.

Notes:

1. This matrix is Example 4.1 in referenced text ["76" on page 1257].
2. Because *lwork* = 0, the subroutine dynamically allocates `WORK`.

Matrix ***A*** is the same matrix used for DSPEVX in Example 1.

Call Statement and Input:

```
              JOBZ UPLO   N   A   LDA W   WORK LWORK INFO
               |    |     |   |    |  |    |     |     |
CALL DSYEV ('V', 'U', 4, A,   4, W, WORK,  0,   INFO)
```

```
A     =  | 5.0  4.0  1.0  1.0 |
         |  .   5.0  1.0  1.0 |
         |  .    .   4.0  2.0 |
         |  .    .    .   4.0 |
         └
```

Output:

```
         |  0.707107  0.000000  0.316228  0.632456 |
```

```
A    =  | -0.707107  0.000000  0.316228  0.632456 |
        |  0.000000  0.707107 -0.632456  0.316228 |
        |  0.000000 -0.707107 -0.632456  0.316228 |
        └                                         ┘
```

```
        |  1.000000 |
W    =  |  2.000000 |
        |  5.000000 |
        | 10.000000 |
        └
```

```
INFO = 0
```

Example 9

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix ***A*** of order 4, where:>|

- DUMMY1 is a placeholder for VL. VL is not used.
- DUMMY2 is a placeholder for VU. VU is not used.

|<

This example also illustrates the use of the *il* and *iu* arguments when *range* = 'I'.

Notes:

1. This matrix is Example 4.1 in referenced text [“76” on page 1257].
2. Because *lwork* = 0, the subroutine dynamically allocates WORK.

Matrix ***A*** is the same matrix used for DSPEVX in Example 1.

Call Statement and Input:

```
             JOBZ RANGE UPLO   N   A   LDA    VL      VU      IL   IU ABSTOL M   W   Z LDZ WORK LWORK IWORK   IFAIL   INFO
              |     |     |    |   |    |     |        |       |    |    |    |   |   |  |    |     |     |       |       |
CALL DSYEVX ('V', 'I',  'U', 4, A,  4, DUMMY1, DUMMY2,  1,  3, -1.0, M, W, Z, 4, WORK,  0,  IWORK, IFAIL, INFO)
```

Output:

```
M    =    3
```

```
        | 1.000000 0.000000 0.414214  0.224745 |
A    =  |     .    6.000000 2.828427  0.224745 |
        |     .        .    7.000000 -2.449490 |
        |     .        .        .     4.000000 |
```

```
        |  1.000000 |
W    =  |  2.000000 |
        |  5.000000 |
        |      .    |
```

```
        |  0.707107  0.000000 -0.316228 |
Z    =  | -0.707107  0.000000 -0.316228 |
        |  0.000000 -0.707107  0.632456 |
        |  0.000000  0.707107  0.632456 |
```

```
IFAIL = (0,0,0,.)
INFO = 0
```

Example 10

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix ***A*** of order 4, where:>|

- DUMMY1 is a placeholder for VL. VL is not used.
- DUMMY2 is a placeholder for VU. VU is not used.

|<

This example also illustrates the use of the *il* and *iu* arguments when *range* = 'I'.

Notes:

1. Because *lwork* = 0, the subroutine dynamically allocates WORK.
2. Because *lrwork* = 0, the subroutine dynamically allocates RWORK.
3. Because *liwork* = 0, the subroutine dynamically allocates IWORK.

Matrix ***A*** is:

```
                 |  5.0 4.0 0.0 0.0 |
                 |  4.0 5.0 1.0 0.0 |
                 |  0.0 1.0 4.0 1.0 |
                 |  0.0 0.0 1.0 4.0 |
                 └
```

Call Statement and Input:

```
           JOBZ RANGE UPLO  N  A  LDA   VL      VU    IL  IU ABSTOL M  W  Z LDZ ISSUPZ WORK  LWORK IWORK LIWORK INFO
            |    |     |    |  |   |     |       |     |   |   |     |  |  |  |    |     |      |     |       |     |
CALL DSYEVR ('V', 'I',  'U', 4, A,  4, DUMMY1, DUMMY2, 1,  4, -1.0, M, W, Z, 4, ISSUPZ, WORK, 0,  IWORK,   0,   INFO)
```

Output:

```
M     =    4
```

Matrix A is overwritten on output.

```
        |  0.821597 |
W     = |  3.075215 |
        |  5.000000 |
        |  9.103188 |
        └
```

```
        |  -0.668539   0.214534  -0.174078   0.690455 |
Z     = |   0.698357  -0.103233   0.000000   0.708266 |
        |  -0.243858  -0.659434   0.696311   0.144331 |
        |   0.076724   0.713067   0.696311   0.028283 |
        └
```

```
          |  1 |
          |  4 |
          |  1 |
          |  4 |
ISUPPZ  = |  1 |
          |  4 |
          |  1 |
          |  4 |
          └
```

```
INFO = 0
```

Example 11

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix ***A*** of order 4, where:>|

- DUMMY3 is a placeholder for IL. IL is not used.
- DUMMY4 is a placeholder for IU. IU is not used.

|<

This example also illustrates the use of the *vl* and *vu* arguments when *range* = 'V'.

Notes:

1. This matrix is Example 4.1 in referenced text [“76” on page 1257].
2. Because *lwork* = 0, the subroutine dynamically allocates WORK.
3. >|Because *lrwork* = 0, the subroutine dynamically allocates RWORK.|<
4. >|Because *liwork* = 0, the subroutine dynamically allocates IWORK.|<

Matrix ***A*** is the same matrix used for DSPEVX in Example 1.

Call Statement and Input:

```
             JOBZ RANGE UPLO  N  A  LDA VL   VU     IL      IU    ABSTOL  M  W  Z LDZ WORK LWORK IWORK  IFAIL  INFO
               |    |     |   |  |   |   |    |      |       |      |     |  |  |  |   |     |     |      |      |
CALL DSYEVX ('V', 'V',  'U', 4, A,  4, 3.0, 11.0, DUMMY3, DUMMY4, -1.0, M, W, Z, 4, WORK,  0,  IWORK, IFAIL, INFO)
```

>|***–or–***|<

>|

```
             JOBZ RANGE UPLO  N  A  LDA  VL  VU     IL       IU  ABSTOL  M  W  Z LDZ WORK LWORK IWORK  ISUPPZ INFO
               |    |     |   |  |   |   |    |      |       |      |     |  |  |  |   |     |     |      |      |
CALL DSYEVR ('V', 'V',  'U', 4, A,  4, 3.0, 11.0, DUMMY3, DUMMY4, -1.0, M, W, Z, 4, WORK,  0,  IWORK, ISUPPZ, INFO)
```

|<

Output:

```
M    =    2
```

- For DSYEVX,

```
A    =    | 1.000000 0.000000 0.414214  0.224745 |
          |     .    6.000000 2.828427  0.224745 |
          |     .        .    7.000000 -2.449490 |
          |     .        .        .     4.000000 |
          └
```

- >|For DSYEVR, A is overwritten on output. |<

```
W    =    |  5.000000 |
          | 10.000000 |
          |      .    |
          |      .    |
          └
```

```
Z    =    | -0.316228 -0.632456 |
          | -0.316228 -0.632456 |
          |  0.632456 -0.316228 |
          |  0.632456 -0.316228 |
          └
```

For DSYEVX, `IFAIL = (0,0,.,.)`

>|For DSYEVR, `ISUPPZ` is not used when `RANGE = 'V'`.|<

```
INFO = 0
```

Example 12

This example shows how to find the eigenvalues only of a complex Hermitian matrix ***A*** of order 3, where:>|

- `DUMMY1` is a placeholder for `VL`. `VL` is not used.
- `DUMMY2` is a placeholder for `VU`. `VU` is not used.
- `DUMMY3` is a placeholder for `IL`. `IL` is not used.
- `DUMMY4` is a placeholder for `IU`. `IU` is not used.

|<

Notes:

1. This matrix is Example 6.3 in referenced text ["76" on page 1257].
2. Because *lwork* = 0, the subroutine dynamically allocates `WORK`.
3. >|Because *lrwork* = 0, the subroutine dynamically allocates `RWORK`.|<
4. >|Because *liwork* = 0, the subroutine dynamically allocates `IWORK`.|<
5. For ZHEEVX, matrix ***A*** is the same matrix used for ZHPEVX in Example 4.

Call Statement and Input:

```
           JOBZ UPLO  N  A  LDA W  WORK LWORK RWORK INFO
             |    |   |  |   |  |    |    |     |     |
CALL ZHEEV ('N', 'L', 3, A,  3, W, WORK,  0,  RWORK INFO)
```

–or–

```
            JOBZ RANGE UPLO  N  A  LDA     VL     VU     IL      IU    ABSTOL M  W  Z  LDZ  WORK LWORK RWORK IWORK  IFAIL  INFO
              |    |     |   |  |   |       |      |      |       |      |    |  |  |   |     |    |     |     |      |      |
CALL ZHEEVX ('N', 'A',  'L', 3, A, LDA, DUMMY1, DUMMY2, DUMMY3, DUMMY4, -1.0, M, W, Z,  3,  WORK,  0,  RWORK, IWORK, IFAIL, INFO)
```

>|***–or–***|<

>|

```
            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 ('N', 'A',  'L', 3, A, LDA, DUMMY1, DUMMY2, DUMMY3, DUMMY4, -1.0,  M, W, Z,  3, ISUPPZ, WORK, 0,   RWORK,  0,   IWORK, 0,   INFO)
```

|<

```
A     =  | ( 2.0, 0.0)       .           .      |
         | ( 0.0,-1.0) (2.0,  0.0)       .      |
         | ( 0.0, 0.0) (0.0,  0.0) (3.0, 0.0)  |
         L
```

Output:

```
M = 3
```

- For ZHEEV, A is overwritten on output.

- For ZHEEVX,

```
A    =  | ( 2.0, 0.0)       .            .      |
        | (-1.0, 0.0) (2.0,  0.0)        .      |
        | ( 0.0, 0.0) (0.0,  0.0) (3.0, 0.0) |
        └
```

- >|For ZHEEVR, A is overwritten on output.|<

```
W    =  |  1.000000 |
        |  3.000000 |
        |  3.000000 |
        └
```

Z is not used when `JOBZ = 'N'`.

>|For ZHEEVR, `ISUPPZ` is not used when `JOBZ = 'N'`.|<

For ZHEEVX, `IFAIL` is not used when `JOBZ = 'N'`.

```
INFO = 0
```

Example 13

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix ***A*** of order 3 stored in upper storage mode.

Notes:

1. This matrix is Example 6.3 in referenced text [“76” on page 1257].
2. Because *lwork* = 0, the subroutine dynamically allocates `WORK`.
3. >|Because *lrwork* = 0, the subroutine dynamically allocates `RWORK`.|<
4. >|Because *liwork* = 0, the subroutine dynamically allocates `IWORK`.|<

Matrix ***A*** is the same matrix used for ZHPEVX in Example 4.

Call Statement and Input:

```
           JOBZ UPLO   N   A   LDA   W   WORK LWORK RWORK   INFO
             |     |    |   |    |    |     |     |     |       |
CALL ZHEEV ('V', 'U', 3, A,   3,   W, WORK,   0,   RWORK, INFO)
```

```
A    =  | (2.0,  0.0) ( 0.0, 1.0) (0.0, 0.0) |
        |      .      ( 2.0, 0.0) (0.0, 0.0) |
        |      .            .     (3.0, 0.0) |
        └
```

Output:

```
A    =  | ( 0.0000, 0.7071) (0.0000, 0.7071) (0.0000, 0.0000) |
        | (-0.7071, 0.0000) (0.7071, 0.0000) (0.0000, 0.0000) |
        | ( 0.0000, 0.0000) (0.0000, 0.0000) (1.0000, 0.0000) |
        └
```

```
W    =  |  1.000000 |
        |  3.000000 |
        |  3.000000 |
        └
```

```
INFO  = 0
```

Example 14

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix ***A*** of order 3, where:>|

- DUMMY1 is a placeholder for VL. VL is not used.
- DUMMY2 is a placeholder for VU. VU is not used.

|<

This example also illustrates the use of the *il* and *iu* arguments when *range* = 'I'.

Notes:

1. This matrix is Example 6.3 in referenced text [“76” on page 1257].
2. Because *lwork* = 0, the subroutine dynamically allocates WORK.
3. >|Because *lrwork* = 0, the subroutine dynamically allocates RWORK.|<
4. >|Because *liwork* = 0, the subroutine dynamically allocates IWORK.|<

Matrix ***A*** is the same matrix used for ZHPEVX in Example 4.

Call Statement and Input:

```
             JOBZ RANGE UPLO  N  A  LDA    VL     VU     IL  IU  ABSTOL M  W  Z  LDZ  WORK  LWORK RWORK  IWORK  IFAIL  INFO
              |    |     |    |  |   |     |      |      |   |     |    |  |  |   |    |      |    |      |      |      |
CALL ZHEEVX ('V', 'I',  'U', 3, A, LDA, DUMMY1, DUMMY2,  1,  2, -1.0,  M, W, Z,  3,  WORK,   0,  RWORK, IWORK, IFAIL, INFO)
```

Output:

```
M    =    2
```

```
A    =   | (2.0,  0.0) (-1.0, 0.0) (0.0, 0.0) |
         |       .      ( 2.0, 0.0) (0.0, 0.0) |
         |       .             .    (3.0, 0.0) |
```

```
W    =   |  1.000000 |
         |  3.000000 |
         |      .    |
```

```
Z    =   | (0.0000, -0.7071), (0.0000, 0.1591) |
         | (0.7071,  0.0000), (0.1591, 0.0000) |
         | (0.0000,  0.0000), (0.9744, 0.0000) |
```

```
IFAIL = (0,0,.)
INFO  = 0
```

Example 15

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix ***A*** of order 3, where:>|

- DUMMY1 is a placeholder for VL. VL is not used.
- DUMMY2 is a placeholder for VU. VU is not used.

|<

This example also illustrates the use of the *il* and *iu* arguments when *range* = 'I'.

Notes:

1. This matrix is Example 6.3 in referenced text [“76” on page 1257].
2. Because *lwork* = 0, the subroutine dynamically allocates WORK.
3. Because *lrwork* = 0, the subroutine dynamically allocates RWORK.
4. Because *liwork* = 0, the subroutine dynamically allocates IWORK.

Matrix ***A*** is the same matrix used in Example 4.

Call Statement and Input:

```
            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 ('V', 'I',  'U', 3, A, LDA, DUMMY1, DUMMY2, 1, 3, -1.0,   M, W, Z,  3, ISUPPZ, WORK,  0,  RWORK,  0,   IWORK,    0,    INFO)
```

Output:

```
M     =    3
```

Matrix A is overwritten on output.

```
W     =  |  1.000000 |
         |  3.000000 |
         |  3.000000 |
         └
```

```
Z     =  | (0.0000, -0.7071), (0.0000, 0.7071), ( 0.0000, 0.0000) |
         | (0.7071,  0.0000), (0.7071, 0.0000), ( 0.0000, 0.0000) |
         | (0.0000,  0.0000), (0.0000, 0.0000), ( 1.0000, 0.0000) |
         └
```

```
              |  1 |
              |  2 |
              |  1 |
ISUPPZ =      |  2 |
              |  3 |
              |  3 |
              └
```

```
INFO  = 0
```

Example 16

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix ***A*** of order 3, where:>|

- DUMMY3 is a placeholder for IL. IL is not used.
- DUMMY4 is a placeholder for IU. IU is not used.

|<

This example also illustrates the use of the *vl* and *vu* arguments when *range* = 'V'.

Notes:

1. This matrix is Example 6.3 in referenced text [“76” on page 1257].
2. Because *lwork* = 0, the subroutine dynamically allocates WORK.
3. >|Because *lrwork* = 0, the subroutine dynamically allocates RWORK.|<
4. >|Because *liwork* = 0, the subroutine dynamically allocates IWORK.|<

Matrix ***A*** is the same matrix used for ZHPEVX in Example 4.

Call Statement and Input:

```
             JOBZ RANGE UPLO  N  A  LDA  VL   VU    IL      IU    ABSTOL  M  W  Z  LDZ WORK LWORK RWORK  IWORK  IFAIL  INFO
              |    |     |    |  |   |    |    |     |       |      |     |  |  |   |    |    |     |      |      |      |
CALL ZHEEVX ('V', 'V',  'U', 3, A, LDA, 2.0, 4.0, DUMMY3, DUMMY4, -1.0,  M, W, Z,  3, WORK,  0,  RWORK, IWORK, IFAIL, INFO)
```

–or–

>|

```
             JOBZ RANGE UPLO N  A  LDA  VL   VU    IL     IU    ABSTOL M  W  Z  LDZ ISUPPZ WORK LWORK RWORK LRWORK IWORK  LRWORK INFO
              |    |     |   |  |   |    |    |     |      |       |   |  |  |   |    |     |     |     |      |     |      |      |
CALL ZHEEVR ('V', 'V', 'U', 3, A, LDA, 2.0, 4.0, DUMMY3, DUMMY4, -1.0, M, W, Z,  3, ISUPPZ,WORK,  0,  RWORK,   0,   IWORK,  0,   INFO)
```

|<

Output:

```
M    =    2
```

- For ZHEEVX,

```
A    =  | (2.0,  0.0) (-1.0, 0.0) (0.0, 0.0) |
        |      .      ( 2.0, 0.0) (0.0, 0.0) |
        |      .             .    (3.0, 0.0) |
        └
```

- For ZHEEVR, A is overwritten on output.

```
W    =  |  3.000000 |
        |  3.000000 |
        |      .    |
        └
```

- For ZHEEVX,

```
Z    =  | ( 0.0000, 0.6634), ( 0.0000, 0.2447) |
        | (-0.6634, 0.0000), (-0.2447, 0.0000) |
        | (-0.3460, 0.0000), ( 0.9382, 0.0000) |
        └
```

- >|For ZHEEVR

```
Z    =  | ( 0.0000,-0.6634),  ( 0.0000, -0.2447) |
        | (-0.6634, 0.0000),  (-0.2447,  0.0000) |
        | (-0.3460, 0.0000),  ( 0.9382,  0.0000) |
        └
```

|<

For ZHEEVX, `IFAIL = (0,0,.)`.

>|For ZHEEVR, `ISUPPZ` is not used when `RANGE = 'V'`.|<

```
INFO  = 0
```

SSPEVD, DSPEVD, CHPEVD, ZHPEVD, SSYEVD, DSYEVD, CHEEVD, and ZHEEVD (Eigenvalues and, Optionally the Eigenvectors, of a Real Symmetric or Complex Hermitian Matrix Using a Divide-and-Conquer Algorithm)

Purpose

These subroutines compute eigenvalues and, optionally, the eigenvectors of a real symmetric matrix or a complex Hermitian matrix.

If eigenvalues only are computed, these subroutines compute the eigenvalues using the Pal-Walker-Kahan variant of the QL or QR algorithm.

If eigenvectors are computed, the subroutine uses a divide-and-conquer method to compute them:

- SSPEVD and DSPEVD compute eigenvalues and, optionally, the eigenvectors of real symmetric matrix ***A***, stored in lower- or upper-packed storage mode.
- CHPEVD and ZHPEVD compute eigenvalues and, optionally, the eigenvectors of complex Hermitian matrix ***A***, stored in lower- or upper-packed storage mode.
- SSYEVD and DSYEVD compute eigenvalues and, optionally, the eigenvectors of real symmetric matrix ***A***, stored in lower or upper storage mode.
- CHEEVD and ZHEEVD compute eigenvalues and, optionally, the eigenvectors of complex Hermitian matrix ***A***, stored in lower or upper storage mode.

Eigenvalues are returned in vector w and eigenvectors are returned in matrix ***Z*** (for subroutines SSPEVD, DSPEVD, CHPEVD, ZHPEVD) or in matrix ***A*** (for subroutines SSYEVD, DSYEVD, CHEEVD, ZHEEVD):

$\boldsymbol{Az} = w\boldsymbol{z}$

where $\boldsymbol{A} = \boldsymbol{A}^T$ or $\boldsymbol{A} = \boldsymbol{A}^H$.

Table 207: Data Types.

w, rwork	***A, Z, work***	**Subroutine**
Short-precision real	Short-precision real	SSPEVD$^{\Delta}$ SSYEVD$^{\Delta}$
Long-precision real	Long-precision real	DSPEVD$^{\Delta}$ DSYEVD$^{\Delta}$
Short-precision real	Short-precision complex	CHPEVD$^{\Delta}$ CHEEVD$^{\Delta}$
Long-precision real	Long-precision complex	ZHPEVD$^{\Delta}$ ZHEEVD$^{\Delta}$
$^{\Delta}$LAPACK		

Syntax

Table 208: Syntax for supported computing languages.

Language	Syntax
Fortran	CALL SSPEVD | DSPEVD (*jobz*, *uplo*, *n*, *ap*, *w*, *z*, *ldz*, *work*, *lwork*, *iwork*, *liwork*, *info*) CALL CHPEVD | ZHPEVD (*jobz*, *uplo*, *n*, *ap*, *w*, *z*, *ldz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*) CALL SSYEVD | DSYEVD (*jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *iwork*, *liwork*, *info*) CALL CHEEVD | ZHEEVD (*jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*)

C and C++	sspevd | dspevd (*jobz*, *uplo*, *n*, *ap*, *w*, *z*, *ldz*, *work*, *lwork*, *iwork*, *liwork*, *info*);
	chpevd | zhpevd (*jobz*, *uplo*, *n*, *ap*, *w*, *z*, *ldz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*);
	ssyevd | dsyevd (*jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *iwork*, *liwork*, *info*);
	cheevd | zheevd (*jobz*, *uplo*, *n*, *a*, *lda*, *w*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*);

LAPACKE	info = LAPACKE_sspevd | LAPACKE_dspevd (*matrix_layout*, *jobz*, *uplo*, *n*, *ap*, *w*, *z*, *ldz*);
	info = LAPACKE_chpevd | LAPACKE_zhpevd (*matrix_layout*, *jobz*, *uplo*, *n*, *ap*, *w*, *z*, *ldz*);
	info = LAPACKE_ssyevd | LAPACKE_dsyevd (*matrix_layout*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*);
	info = LAPACKE_cheevd | LAPACKE_zheevd (*matrix_layout*, *jobz*, *uplo*, *n*, *a*, *lda*, *w*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

jobz

indicates the type of computation to be performed, where:

If *jobz* = 'N', eigenvalues only are computed.

If *jobz* = 'V', eigenvalues and eigenvectors are computed.

Specified as: a single character; *jobz* = 'N' or 'V'.

uplo

indicates whether the upper or lower triangular part of the matrix ***A*** is referenced, where:

If *uplo* = 'U', the upper triangular part is referenced.

If *uplo* = 'L', the lower triangular part is referenced.

Specified as: a single character; *uplo* = 'U' or 'L'.

n

is the order of matrix ***A*** used in the computation.

Specified as: an integer; $n \geq 0$.

ap

is the real symmetric or complex Hermitian matrix ***A*** of order *n*. It is stored in an array, referred to as AP, where:

If *uplo* = 'U', it is stored in upper-packed storage.

If *uplo* = 'L', it is stored in lower-packed storage mode.

Specified as: one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 207 on page 893.

a

is the real symmetric or complex Hermitian matrix ***A*** of order *n*.

If *uplo* = 'U', it is stored in upper storage mode.

If *uplo* = 'L', it is stored in lower storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 207 on page 893.

lda

is the leading dimension of the array specified for ***A***.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

w

See On Return.

z

See On Return.

ldz

is the leading dimension of the array specified for ***Z***.

Specified as: an integer; $ldz > 0$ and, if *jobz* = 'V', $ldz \geq n$.

work

is a work area used by these subroutines.

if *lwork* = 0 and *liwork* ≠ -1 and *lrwork* ≠ -1, *work* is ignored.

If *lwork* ≠ -1, and *liwork* ≠ -1 and *lrwork* ≠ -1, *work* is (at least) of length *lwork*.

If *lwork* = -1, or *liwork* = -1, or *lrwork* = -1, *work* is (at least) of length 1.

Specified as: an area of storage containing numbers of the data type indicated in Table 207 on page 893.

lwork

is used to determine the size of the `WORK` array.

Specified as: an integer, where:

If *lwork* = 0 and *liwork* ≠ -1 and *lrwork* ≠ -1, the subroutine dynamically allocates the workspace needed for use during this computation. The dynamically allocated workspace will be freed prior to returning control to the calling program.

If *lwork* = -1 or *liwork* = -1 or *lrwork* = -1, these subroutines perform a work area query for all work areas and return the optimal size of *work* in $work_1$ and *iwork* in $iwork_1$ and *rwork* in $rwork_1$.

Otherwise:

- If $n \leq 1$, *lwork* must be (at least) 1.
- If *jobz* = 'N' and $n > 1$, *lwork* is as follows:

  For SSPEVD, DSPEVD
  lwork must be (at least) $2n$

  For SSYEVD, and DSYEVD
  lwork must be (at least) $2n + 1$

For CHPEVD and ZHPEVD
lwork must be (at least) n

For CHEEVD and ZHEEVD
lwork must be (at least) $n + 1$

- If *jobz* = 'V' and $n > 1$, *lwork* is as follows:

For SSPEVD and DSPEVD
lwork must be (at least) $1 + 6n + n^2$

For SSYEVD and DSYEVD
lwork must be (at least) $1 + 6n + 2n^2$

For CHPEVD and ZHPEVD
lwork must be (at least) $2n$

For CHEEVD and ZHEEVD
lwork must be (at least) $2n + n^2$

Note: These formulas represent the minimum workspace required. For best performance, specify either *lwork* = -1 (to obtain the optimal size to use) or *lwork* = 0 (to direct the subroutine to dynamically allocate the workspace).

rwork
has the following meaning:

if *lrwork* = 0 and *liwork* ≠ -1 and *lwork* ≠ -1, *rwork* is ignored.

If *lrwork* ≠ -1, and *liwork* ≠ -1 and *lwork* ≠ -1, *rwork* is (at least) of length *lrwork*.

If *lrwork* = -1, or *liwork* = -1, or *lwork* = -1, *rwork* is (at least) of length 1.

Specified as: an area of storage containing real numbers of the data type indicated in Table 207 on page 893.

lrwork
is the number of elements in array `rwork`.

Specified as: a fullword integer; where:

If *lrwork* = 0 and *liwork* ≠ -1 and *lwork* ≠ -1 , the subroutine dynamically allocates the workspace needed for use during this computation. The dynamically allocated workspace will be freed prior to returning control to the calling program.

If *lrwork* = -1 or *liwork* = -1 or *lwork* = -1, these subroutines perform a work area query for all work areas and return the optimal size of *work* in $work_1$ and *iwork* in $iwork_1$ and *rwork* in $rwork_1$.

Otherwise:

- If $n \le 1$, *lrwork* must be (at least) 1
- If *jobz* = 'N' and $n > 1$, *lrwork* must be (at least) n
- If *jobz* = 'V' and $n > 1$, *lrwork* must be (at least) $1 + 5n + 2n^2$

iwork
has the following meaning:

if *liwork* = 0 and *lrwork* ≠ -1 and *lwork* ≠ -1, *iwork* is ignored.

If *liwork* ≠ -1, and *lrwork* ≠ -1 and *lwork* ≠ -1, *iwork* is (at least) of length *liwork*.

If *liwork* = -1, or *lrwork* = -1, or *lwork* = -1, *iwork* is (at least) of length 1.

Specified as: an area of storage containing fullword integers.

liwork
is the number of elements in array `IWORK`.

Specified as: a fullword integer; where:

If *liwork* = 0 and *lrwork* ≠ -1 and *lwork* ≠ -1 , the subroutine dynamically allocates the workspace needed for use during this computation. The dynamically allocated workspace will be freed prior to returning control to the calling program.

If *liwork* = -1 or *lrwork* = -1 or *lwork* = -1, these subroutines perform a work area query for all work areas and return the optimal size of *work* in $work_1$ and *iwork* in $iwork_1$ and *rwork* in $rwork_1$.

Otherwise:

- If $n \le 1$, *liwork* must be (at least) 1
- If *jobz* = 'N' and $n > 1$, *liwork* must be (at least) 1
- If *jobz* = 'V' and $n > 1$, *lwork* must be (at least) 3+ 5*n*

On Return

ap

the matrix ***A*** is overwritten by values generated during the reduction to tridiagonal form.

If *uplo* = 'U', the diagonal and first superdiagonal of the tridiagonal matrix ***T*** overwrite the corresponding elements of ***A***.

If *uplo* = 'L', the diagonal and first subdiagonal of ***T*** overwrite the corresponding elements of ***A***.

Returned as: a one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 207 on page 893.

a

If *jobz* = 'V' and *info* = 0, *a* contains the orthonormal eigenvectors corresponding to the computed eigenvalues, with the *i*-th column of ***A*** holding the eigenvector associated with w_i

If *jobz* = 'V' and *info* ≠ 0, no eigenvectors are valid.

If *jobz* = 'N':

- If *uplo* = 'U', the upper triangle of matrix ***A*** is overwritten.
- If *uplo* = 'L', the lower triangle of matrix ***A*** is overwritten.

Returned as: an array of dimension *lda* by (at least) *n*, containing numbers of the data type indicated in Table 207 on page 893.

w

If *info* = 0, *w* is the vector ***w***, containing the computed eigenvalues in ascending order.

If *info* ≠ 0, no eigenvalues are valid.

Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 207 on page 893.

z

has the following meaning, where:

If *jobz* = 'N', then *z* is ignored.

If *jobz* = 'V' and *info* = 0, *z* contains the orthonormal eigenvectors corresponding to the computed eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with w_i.

If *jobz* = 'V' and *info* ≠ 0, no eigenvectors are valid.

Returned as: an *ldz* by *n* array, containing numbers of the data type indicated in Table 207 on page 893.

work

is a work area used by these subroutines.

Returned as: an area of storage where:

If *lwork* ≥ 1 or *lwork* = -1 or *liwork* = -1 or *lrwork* = -1, then $work_1$ is set to the optimal *lwork* value and contains numbers of the data type indicated in Table 207 on page 893.

Except for $work_1$, the contents of *work* are overwritten on return.

rwork
is a work area used by these subroutines.

Returned as: an area of storage where:

If *lrwork* ≥ 1 or *lrwork* = -1 or *liwork* = -1 or *lwork* = -1, then $rwork_1$ is set to the optimal *lrwork* value and contains numbers of the data type indicated in Table 207 on page 893.

Except for $rwork_1$, the contents of *rwork* are overwritten on return.

iwork
is a work area used by these subroutines.

Returned as: an area of storage where:

If *liwork* ≥ 1 or *liwork* = -1 or *lrwork* = -1 or *lwork* = -1, then $iwork_1$ is set to the optimal *liwork* value and contains numbers of the data type indicated in Table 207 on page 893.

Except for $iwork_1$, the contents of *iwork* are overwritten on return.

info
has the following meaning:

If *info* = 0, then all eigenvalues converged. This indicates a normal exit.

If *info* = *i* and *jobz* = 'N', 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* and *jobz* = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *info* / (*n* + 1) through mod(*info*, *n* + 1). No eigenvalues are valid.

Returned as:

- For SSPEVD, DSPEVD, CHPEVD, ZHPEVD, SSYEVD, DSYEVD, CHEEVD, and ZHEEVD returned as: an integer; info ≥ 0.
- For LAPACKE_sspevd, LAPACKE_dspevd, LAPACKE_chpevd, LAPACKE_zhpevd, LAPACKE_ssyevd, LAPACKE_dsyevd, LAPACKE_cheevd, and LAPACKE_zheevd returned as an integer function value; info ≥ 0.

Notes

1. This subroutine accepts lowercase letters for the *jobz* and *uplo* arguments.
2. In your C program, the argument *info* must be passed by reference.
3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.
4. ***A***, ***Z***, *w*, *work*, *rwork*, *iwork* must have no common elements; otherwise, results are unpredictable.
5. For a description of how real symmetric matrices are stored in lower- or upper-packed storage mode, see "Lower-Packed Storage Mode" on page 80 or "Upper-Packed Storage Mode" on page 82, respectively.

   For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see "Complex Hermitian Matrix" on page 84.
6. For a description of how real symmetric matrices are stored in lower or upper storage mode, see "Lower Storage Mode" on page 83 or "Upper Storage Mode" on page 83, respectively.

   For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see "Complex Hermitian Matrix" on page 84.
7. For best performance specify *lwork* = 0, *liwork* = 0, and *lrwork* = 0.

Function

These subroutines compute eigenvalues and, optionally, the eigenvectors of a real symmetric matrix or a complex Hermitian matrix.

If eigenvalues only are computed, these subroutines compute the eigenvalues using the Pal-Walker-Kahan variant of the QL or QR algorithm.

If eigenvectors are computed, the subroutine uses a divide-and-conquer method to compute them:

- SSPEVD and DSPEVD compute eigenvalues and, optionally, the eigenvectors of real symmetric matrix ***A***, stored in lower- or upper-packed storage mode.
- CHPEVD and ZHPEVD compute eigenvalues and, optionally, the eigenvectors of complex Hermitian matrix ***A***, stored in lower- or upper-packed storage mode.
- SSYEVD and DSYEVD compute eigenvalues and, optionally, the eigenvectors of real symmetric matrix ***A***, stored in lower or upper storage mode.
- CHEEVD and ZHEEVD compute eigenvalues and, optionally, the eigenvectors of complex Hermitian matrix ***A***, stored in lower or upper storage mode.

Eigenvalues are returned in vector w and eigenvectors are returned in matrix ***Z*** (for subroutines SSPEVD, DSPEVD, CHPEVD, ZHPEVD) or in matrix ***A*** (for subroutines SSYEVD, DSYEVD, CHEEVD, ZHEEVD):

$\boldsymbol{A}\boldsymbol{z} = w\boldsymbol{z}$

where $\boldsymbol{A} = \boldsymbol{A}^T$ or $\boldsymbol{A} = \boldsymbol{A}^H$.

The computation involves the following steps:

1. If necessary, scale the matrix ***A***.
2. Reduce matrix ***A*** to tridiagonal form.
3. Compute the eigenvalues and, optionally, the eigenvectors of the symmetric tridiagonal matrix. The algorithm used depends on the value specified for *jobz*:
   - If *jobz* = 'N', compute all the eigenvalues using the Pal-Walker-Kahan variant of the QL or QR algorithms.
   - Otherwise, compute both the eigenvalues and eigenvectors using a divide-and-conquer algorithm, then apply Householder transformations to the eigenvector matrix.
4. Rescale eigenvalues appropriately if the matrix was scaled.

If *n* = 0, the subroutine returns after completing parameter checking.

For more information on these methods, see references [“9” on page 1253], [“76” on page 1257], and [“35” on page 1255].

Error conditions

Resource Errors

1. *lwork* = 0, and unable to allocate work area.
2. *lrwork* = 0, and unable to allocate work area.
3. *liwork* = 0, and unable to allocate work area.

Computational Errors

1. If *info* = *i* and *jobz* = 'N', then the algorithm failed to converge. *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.
2. If *info* = *i* and *jobz* = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *info* / (*n* + 1) through mod(*info*, *n* + 1). No eigenvalues are valid.

Input-Argument Errors

1. *jobz* ≠ 'N' or 'V'
2. *uplo* ≠ 'U' or 'L'
3. *n* < 0
4. *lda* ≤ 0
5. *n* > *lda*
6. *ldz* ≤ 0
7. *n* > *ldz* and *jobz* = 'V'
8. *lwork* ≠ 0 and *lwork* ≠ -1 and *liwork* ≠ -1 and *lrwork* ≠ -1, and *lwork* < the minimum required value
9. *liwork* ≠ 0 and *liwork* ≠ -1 and *lwork* ≠ -1 and *lrwork* ≠ -1, and *liwork* < the minimum required value
10. *lrwork* ≠ 0 and *lrwork* ≠ -1 and *liwork* ≠ -1 and *lwork* ≠ -1 and *lrwork* < the minimum required value
11. The size of a work array is greater than 2147483647 when 32-bit integers are used.

Examples

Example 1

This example shows how to find the eigenvalues only of a real symmetric matrix of order 4, stored in lower-packed storage mode.

Notes:

1. Because *lwork* = 0, the subroutine dynamically allocates `WORK`.
2. Because *liwork* = 0, the subroutine dynamically allocates `IWORK`.
3. ***Z*** is not used when *jobz* = 'N'.
4. On output, array AP is overwritten.
5. This matrix is Example 4.1 in referenced text [“76” on page 1257].

Matrix ***A*** is:

```
                 |  5.0 4.0 1.0 1.0 |
                 |  4.0 5.0 1.0 1.0 |
                 |  1.0 1.0 4.0 2.0 |
                 |  1.0 1.0 2.0 4.0 |
```

Call Statement and Input:

```
          JOBZ  UPLO N  AP  W  Z  LDZ  WORK  LWORK  IWORK  LIWORK  INFO
            |    |   |   |  |  |   |     |     |      |       |      |
CALL DSPEVD ('N', 'L', 4, AP, W, Z,  4,  WORK,   0,   IWORK,    0,   INFO)

AP       =  (5.0, 4.0, 1.0, 1.0, 5.0, 1.0, 1.0, 4.0, 2.0, 4.0)
```

Output:

```
        |  1.000000 |
W    =  |  2.000000 |
        |  5.000000 |
        | 10.000000 |
```

```
INFO = 0
```

Example 2

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix of order 4, stored in upper-packed storage mode.

Notes:

1. Because *lwork* = 0, the subroutine dynamically allocates `WORK`.
2. Because *liwork* = 0, the subroutine dynamically allocates `IWORK`.
3. On output, array AP is overwritten.
4. This matrix is Example 4.1 in referenced text [“76” on page 1257].

Matrix ***A*** is the same as in Example 1.

Call Statement and Input:

```
            JOBZ  UPLO N  AP  W  Z  LDZ  WORK  LWORK  IWORK  LIWORK  INFO
              |    |   |   |  |  |   |     |     |      |      |      |
CALL DSPEVD ('V', 'U', 4, AP, W, Z,  4,  WORK,   0,   IWORK,    0,   INFO)

AP       =  (5.0,4.0,5.0,1.0,1.0,4.0,1.0,1.0,2.0,4.0)
```

Output:

```
         |  1.000000 |
W    =   |  2.000000 |
         |  5.000000 |
         | 10.000000 |
```

```
         |-0.707107  0.000000  0.316228 -0.632456|
Z    =   | 0.707107  0.000000  0.316228 -0.632456|
         | 0.000000 -0.707107 -0.632456 -0.316228|
         | 0.000000  0.707107 -0.632456 -0.316228|
```

```
INFO = 0
```

Example 3

This example shows how to find the eigenvalues only of a complex Hermitian matrix of order 3, stored in lower-packed storage mode.

Notes:

1. Because *lwork* = 0, the subroutine dynamically allocates `WORK`.
2. Because *lrwork* = 0, the subroutine dynamically allocates `RWORK`.
3. Because *liwork* = 0, the subroutine dynamically allocates `IWORK`.
4. ***Z*** is not used when *jobz* = 'N'.
5. On output, array AP is overwritten.
6. This matrix is Example 4.1 in referenced text [“76” on page 1257].

Matrix ***A*** is:

```
                |( 2.0, 0.0) ( 0.0, 1.0) ( 0.0, 0.0)|
                |( 0.0,-1.0) ( 2.0, 0.0) ( 0.0, 0.0)|
                |( 0.0, 0.0) ( 0.0, 0.0) ( 3.0, 0.0)|
```

Call Statement and Input:

```
            JOBZ UPLO  N  AP  W  Z  LDZ  WORK  LWORK
              |    |   |   |  |  |   |     |     |
CALL ZHPEVD ('N', 'L', 3, AP, W, Z,  3,  WORK,   0,

            RWORK  LRWORK  IWORK  LIWORK  INFO
              |      |       |       |      |
            RWORK,   0,    IWORK,    0,   INFO)
```

```
AP       =  ((2.0, . ),(0.0,-1.0),(0.0,0.0),
             (2.0, . ),(0.0,0.0),(3.0, . ))
```

Output:

```
        |  1.000000 |
W    =  |  3.000000 |
        |  3.000000 |
```

```
INFO  = 0
```

Example 4

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix of order 3, stored in upper-packed storage mode.

Notes:

1. Because *lwork* = 0, the subroutine dynamically allocates WORK.
2. Because *lrwork* = 0, the subroutine dynamically allocates RWORK.
3. Because *liwork* = 0, the subroutine dynamically allocates IWORK.
4. On output, array AP is overwritten.
5. This matrix is Example 4.1 in referenced text [“76” on page 1257].

Matrix ***A*** is the same as in Example 3.

Call Statement and Input:

```
            JOBZ UPLO  N  AP  W  Z  LDZ  WORK  LWORK  RWORK  LRWORK  IWORK  LIWORK  INFO
             |    |    |   |  |  |   |     |      |     |       |      |       |      |
CALL ZHPEVD ('V', 'U', 3, AP, W, Z,  3,  WORK,    0,  RWORK,    0,   IWORK,    0,   INFO)

AP      =  ((2.0, . ),(0.0,-1.0),(0.0,0.0),(2.0, . ),(0.0,0.0),(3.0, . ))
```

Output:

```
        |  1.000000 |
W    =  |  3.000000 |
        |  3.000000 |
```

```
        |( -0.7071, 0.0) ( -0.7071, 0.0) ( 0.0, 0.0)|
Z    =  |(  0.0,-0.7071) (  0.0, 0.7071) ( 0.0, 0.0)|
        |(  0.0, 0.0   ) (  0.0, 0.0   ) ( 1.0, 0.0)|
```

```
INFO  = 0
```

Example 5

This example shows how to find the eigenvalues only of a real symmetric matrix of order 4, stored in lower storage mode.

Notes:

1. Because *lwork* = 0, the subroutine dynamically allocates WORK.
2. Because *liwork* = 0, the subroutine dynamically allocates IWORK.
3. On output, array ***A*** is overwritten.
4. This matrix is Example 4.1 in referenced text [“76” on page 1257].

Matrix ***A*** is:

```
                 |  5.0 4.0 1.0 1.0 |
                 |  4.0 5.0 1.0 1.0 |
                 |  1.0 1.0 4.0 2.0 |
                 |  1.0 1.0 2.0 4.0 |
```

Call Statement and Input:

```
            JOBZ UPLO  N  A  LDA  W  WORK  LWORK  IWORK  LIWORK  INFO
              |    |   |  |   |   |    |     |      |      |      |
CALL DSYEVD ('N', 'L', 4, A,  4,  W, WORK,   0,   IWORK,    0,   INFO)


A       =  | 5.0   .    .    .  |
           | 4.0  5.0   .    .  |
           | 1.0  1.0  4.0   .  |
           | 1.0  1.0  2.0  4.0 |
```

Output:

```
        |  1.000000 |
W    =  |  2.000000 |
        |  5.000000 |
        | 10.000000 |
```

```
INFO = 0
```

Example 6

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix of order 4, stored in upper storage mode.

Notes:

1. Because *lwork* = 0, the subroutine dynamically allocates `WORK`.
2. Because *liwork* = 0, the subroutine dynamically allocates `IWORK`.
3. This matrix is Example 4.1 in referenced text [“76” on page 1257].

Matrix ***A*** is the same as in Example 5.

Call Statement and Input:

```
            JOBZ UPLO  N  A  LDA  W  WORK  LWORK  IWORK  LIWORK  INFO
              |    |   |  |   |   |    |     |      |      |      |
CALL DSYEVD ('V', 'U', 4, A,  4,  W, WORK,   0,    IWORK,   0,   INFO)


A       =  | 5.0  4.0  1.0  1.0 |
           |  .   5.0  1.0  1.0 |
           |  .    .   4.0  2.0 |
           |  .    .    .   4.0 |
```

Output:

```
        |-0.707107  0.000000  0.316228 -0.632456|
A    =  | 0.707107  0.000000  0.316228 -0.632456|
        | 0.000000 -0.707107 -0.632456 -0.316228|
        | 0.000000  0.707107 -0.632456 -0.316228|
```

```
        |  1.000000 |
W    =  |  2.000000 |
        |  5.000000 |
        | 10.000000 |
```

```
INFO = 0
```

Example 7

This example shows how to find the eigenvalues only of a complex Hermitian matrix of order 3, stored in lower storage mode.

Notes:

1. Because *lwork* = 0, the subroutine dynamically allocates WORK.
2. Because *lrwork* = 0, the subroutine dynamically allocates RWORK.
3. Because *liwork* = 0, the subroutine dynamically allocates IW0RK.
4. On output, array***A*** is overwritten.
5. This matrix is Example 4.1 in referenced text [“76” on page 1257].

Matrix ***A*** is:

```
                 |( 2.0, 0.0) ( 0.0, 1.0) ( 0.0, 0.0)|
                 |( 0.0,-1.0) ( 2.0, 0.0) ( 0.0, 0.0)|
                 |( 0.0, 0.0) ( 0.0, 0.0) ( 3.0, 0.0)|
```

Call Statement and Input:

```
           JOBZ UPLO  N  A  LDA  W  WORK  LWORK  RWORK  LRWORK  IWORK  LIWORK  INFO
             |    |   |  |   |   |   |      |      |       |      |      |      |
CALL ZHEEVD ('N', 'L', 3, A,  3,  W, WORK,   0,   RWORK,    0,    IWORK,   0,   INFO)


           |( 2.0,  . )       .           .     |
A       =  |( 0.0,-1.0) ( 2.0,  . )       .     |
           |( 0.0, 0.0) ( 0.0, 0.0) ( 3.0,  . )|
```

Output:

```
        |  1.000000 |
W    =  |  3.000000 |
        |  3.000000 |
```

```
INFO = 0
```

Example 8

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix ***A*** of order 3, stored in upper storage mode.

Notes:

1. Because *lwork* = 0, the subroutine dynamically allocates WORK.
2. Because *lrwork* = 0, the subroutine dynamically allocates RWORK.
3. Because *liwork* = 0, the subroutine dynamically allocates IW0RK.
4. This matrix is Example 4.1 in referenced text [“76” on page 1257].

Matrix ***A*** is the same as in Example 7.

Call Statement and Input:

```
            JOBZ UPLO  N  A  LDA  W  WORK  LWORK  RWORK  LRWORK  IWORK  LIWORK  INFO
              |    |   |  |   |   |    |     |      |       |      |      |      |
CALL ZHEEVD ('V', 'U', 3, A,  3,  W, WORK,   0,   RWORK,    0,    IWORK,   0,   INFO)


A       =  |( 2.0,  . ) ( 0.0, 1.0) ( 0.0, 0.0)|
           |     .      ( 2.0,  . ) ( 0.0, 0.0)|
           |     .           .      ( 3.0,  . )|
```

Output:

```
A    =  |( -0.7071, 0.0) ( -0.7071, 0.0) ( 0.0, 0.0)|
        |(  0.0,-0.7071) (  0.0, 0.7071) ( 0.0, 0.0)|
        |(  0.0, 0.0   ) (  0.0, 0.0   ) ( 1.0, 0.0)|
```

```
        |  1.000000 |
W    =  |  3.000000 |
        |  3.000000 |
```

```
INFO = 0
```

SGGEV, DGGEV, CGGEV, ZGGEV, SGGEVX, DGGEVX, CGGEVX, and ZGGEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix Generalized Eigenproblem)

Purpose

- SGGEV, DGGEV, CGGEV, and ZGGEV compute the eigenvalues and, optionally, the left and/or right eigenvectors of a general matrix generalized eigenproblem.
- SGGEVX, DGGEVX, CGGEVX, and ZGGEVX compute the eigenvalues and, optionally, right eigenvectors, left eigenvectors, reciprocal condition numbers for eigenvalues, and reciprocal condition numbers for right eigenvectors of a general matrix generalized eigenproblem.

For the left eigenvectors:

$vl^{\mathbf{H}}\boldsymbol{A} = \lambda vl^{\mathbf{H}}\boldsymbol{B}$

For the right eigenvectors:

$\boldsymbol{A}vr = \lambda\boldsymbol{B}vr$

The eigenvalues are returned in two parts, α and β, where:

For SGGEV, DGGEV, SGGEVX, and DGGEVX, α and β are returned in vectors *alphar*, *alphai*, and *beta*, where *alphar* contains the real part of α and *alphai* contains the imaginary part of α.

- If $alphai_j = 0$, then the *j*-th eigenvalue is real.
- If $alphai_j > 0$, then the *j*-th and (*j*+1)-th eigenvalues are a complex conjugate pair.

For CGGEV, ZGGEV, CGGEVX, and ZGGEVX, α and β are returned in vectors *alpha* and *beta*.

For SGGEV, DGGEV, SGGEVX, and DGGEVX:

- If $alphai_j = 0$, then the *j*-th eigenvalue is real:
  - $\alpha_j = alphar_j$
  - $\beta_j = beta_j$
- If $alphai_j > 0$, then the *j*-th and (*j*+1)-th eigenvalues α_{j+1} are a complex conjugate pair:
  - $\alpha_j = (alphar_j, alphai_j)$

- β_j = *beta*$_j$
- α_{j+1} = (*alphar*$_j$, -*alphai*$_j$)
- β_{j+1} = *beta*$_j$

For CGGEV, ZGGEV, CGGEVX, and ZGGEVX:

- α_j = *alpha*$_j$
- β_j = *beta*$_j$

Left eigenvectors are returned in matrix ***VL*** and right eigenvectors are returned in matrix ***VR***.

Table 209: Data Types.

A*, *B*, *VL*, *VR*, *bwork*, *work* α, *β	***alphar*, *alphai*, *lscale*, *rscale*, *abnrm*, *rconde*, *rcondv*, *rwork***	**Subroutine**
Short-precision real	Short-precision real	SGGEV$^{\Delta}$ SGGEVX$^{\Delta}$
Long-precision real	Long-precision real	DGGEV$^{\Delta}$ DGGEVX$^{\Delta}$
Short-precision complex	Short-precision real	CGGEV$^{\Delta}$ CGGEVX$^{\Delta}$
Long-precision complex	Long-precision real	ZGGEV$^{\Delta}$ ZGGEVX$^{\Delta}$
$^{\Delta}$LAPACK		

Syntax

Language	**Syntax**
Fortran	CALL SGGEV | DGGEV (*jobvl*, *jobvr*, *n*, *a*, *lda*, *b*, *ldb*, *alphar*, *alphai*, *beta*, *vl*, *ldvl*, *vr*, *ldvr*, *work*, *lwork*, *info*) CALL CGGEV | ZGGEV (*jobvl*, *jobvr*, *n*, *a*, *lda*, *b*, *ldb*, *alpha*, *beta*, *vl*, *ldvl*, *vr*, *ldvr*, *work*, *lwork*, *rwork*, *info*) CALL SGGEVX | 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 | 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*)

Language	Syntax

C and C++	sggev | dggev (*jobvl*, *jobvr*, *n*, *a*, *lda*, *b*, *ldb*, *alphar*, *alphai*, *beta*, *vl*, *ldvl*, *vr*, *ldvr*, *work*, *lwork*, *info*);
	
	cggev | zggev (*jobvl*, *jobvr*, *n*, *a*, *lda*, *b*, *ldb*, *alpha*, *beta*, *vl*, *ldvl*, *vr*, *ldvr*, *work*, *lwork*, *rwork*, *info*);
	
	sggevx | 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*);
	
	cggevx | 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*);

LAPACKE	info = LAPACKE_sggev | LAPACKE_dggev (*matrix_layout*, *jobvl*, *jobvr*, *n*, *a*, *lda*, *b*, *ldb*, *alphar*, *alphai*, *beta*, *vl*, *ldvl*, *vr*, *ldvr*);
	
	info = LAPACKE_cggev | LAPACKE_zggev (*matrix_layout*, *jobvl*, *jobvr*, *n*, *a*, *lda*, *b*, *ldb*, *alpha*, *beta*, *vl*, *ldvl*, *vr*, *ldvr*);
	
	info = LAPACKE_sggevx | LAPACKE_dggevx (*matrix_layout*, *balanc*, *jobvl*, *jobvr*, *sense*, *n*, *a*, *lda*, *b*, *ldb*, *alphar*, *alphai*, *beta*, *vl*, *ldvl*, *vr*, *ldvr*, *ilo*, *ihi*, *lscale*, *rscale*, *abnrm*, *bbnrm*, *rconde*, *rcondv*);
	
	info = LAPACKE_cggevx | LAPACKE_zggevx (*matrix_layout*, *balanc*, *jobvl*, *jobvr*, *sense*, *n*, *a*, *lda*, *b*, *ldb*, *alpha*, *beta*, *vl*, *ldvl*, *vr*, *ldvr*, *ilo*, *ihi*, *lscale*, *rscale*, *abnrm*, *bbnrm*, *rconde*, *rcondv*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

balanc

indicates whether or not to scale ***A*** diagonally and whether or not to permute its rows and columns to improve the conditioning of its eigenvalues, where *balanc* can have any of the following values:

N

Neither diagonally scale nor permute ***A***.

P

Permute ***A***, but do not diagonally scale it.

S

Diagonally scale ***A***, but do not permute it.

B

Both diagonally scale and permute ***A***.

When diagonal scaling is specified, the subroutine replaces ***A*** with $\boldsymbol{DAD}^{-1}$ where ***D*** is a diagonal matrix chosen to make the rows and columns of ***A*** more equal in norm and the condition numbers of its eigenvalues and eigenvectors smaller.

When permuting is specified, the subroutine makes ***A*** more nearly upper triangular.

The computed reciprocal condition numbers correspond to the balanced matrix. In exact arithmetic, permuting rows and columns does not change the condition numbers, but diagonal scaling does change the condition numbers.

Specified as: a single character. It must be 'N', 'P', 'S', or 'B'.

jobvl
indicates the type of computation to be performed, where:

If *jobvl* = 'N', the left eigenvectors are not computed.

If *jobvl* = 'V', the left eigenvectors are computed.

Specified as: a single character. It must be 'N' or 'V'.

jobvr
indicates the type of computation to be performed, where:

If *jobvr* = 'N', the right eigenvectors are not computed.

If *jobvr* = 'V', the right eigenvectors are computed.

Specified as: a single character. It must be 'N' or 'V'.

sense
indicates which reciprocal numbers to compute (if any), where *sense* can have any of the following values:

N
Do not compute reciprocal condition numbers.

E
Compute reciprocal condition numbers for eigenvalues only.

V
Compute reciprocal condition numbers for right eigenvectors only.

B
Compute reciprocal condition numbers for eigenvalues and right eigenvectors.

Specified as: a single character. It must be 'N', 'E', 'V', or 'B'.

n
is the order of the general matrices ***A*** and ***B***.

Specified as: an integer; $n \geq 0$.

a
is the general matrix ***A*** of order *n*.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 209 on page 906.

lda
is the leading dimension of the array specified for matrix ***A***.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

b
is the general matrix ***B*** of order *n*.

Specified as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 209 on page 906.

ldb
is the leading dimension of the array specified for matrix ***B***.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

alphar
See On Return.

alphai
See On Return.

alpha
See On Return.

beta
See On Return.

vl
See On Return.

ldvl
is the leading dimension of the array specified for *vl*.

Specified as: an integer; *ldvl* > 0; if *jobvl* = 'V', *ldvl* ≥ *n*.

vr
See On Return.

ldvr
is the leading dimension of the array specified for *vr*.

Specified as: an integer; *ldvr* > 0; if *jobvr* = 'V', *ldvr* ≥ *n*.

ilo
See On Return.

ihi
See On Return.

lscale
See On Return.

rscale
See On Return.

abnrm
See On Return.

bbnrm
See On Return.

rconde
See On Return.

rcondv
See On Return.

work
is the storage work area used by this subroutine. Its size is specified by *lwork*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 209 on page 906.

lwork
is the number of elements in array `WORK`.

Specified as an integer, where:

- If *lwork* = 0, the subroutine dynamically allocates the workspace needed for use during this computation. The dynamically allocated workspace will be freed prior to returning control to the calling program.
- If *lwork* = -1, a workspace query is assumed. The subroutine only calculates the optimal size of the `WORK` array and returns this value as the first entry of the `WORK` array.

Otherwise:

- For SGGEV and DGGEV:
  - *lwork* ≥ max(1, 8*n*)
- For CGGEV and ZGGEV:
  - *lwork* ≥ max(1, 2*n*)

- For SGGEVX and DGGEVX:
  - If *sense* = 'N':
    - If *balanc* = 'N' or 'P' and *jobvl* = 'N' and *jobvr* = 'N', *lwork* ≥ max(1,2*n*) ; Otherwise, *lwork* ≥ max(1,6*n*)
  - If *balanc* = 'S' or 'B' or if *jobvl* = 'V' or *jobvr* = 'V', *lwork* ≥ max(1, 6*n*)
  - If *sense* = 'N', *lwork* ≥ max(1, 2*n*)
  - If *sense* = 'E', *lwork* ≥ max(1, 10*n*)
  - If *sense* = 'V' or 'B', *lwork* ≥ $2n^2$+8*n*+16
- For CGGEVX and ZGGEVX:
  - If *sense* = 'N', *lwork* ≥ max(1, 2*n*)
  - If *sense* = 'E', *lwork* ≥ max(1, 4*n*)
  - If *sense* = 'V' or 'B', *lwork* ≥ max(1,2*n*(*n*+1))

These formulas represent the minimum workspace required. For best performance, specify either *lwork* = -1 (to obtain the optimal size to use) or *lwork* = 0 (to direct the subroutine to dynamically allocate the workspace).

rwork

For CGGEV, and ZGGEV, *rwork* is a storage work area of size 8*n*.

For CGEEVX, and ZGGEVX, if *balanc* = 'S' or 'B', *rwork* must be at least max(1,6*n*). Otherwise, *rwork* is of size max(1,2*n*).

Specified as: an area of storage containing numbers of the data type indicated in Table 209 on page 906.

iwork

For SGGEVX and DGGEVX, *iwork* is a storage work area of *n*+6.

For CGEEVX and ZGGEVX, *iwork* is a storage work area of size 2*n*-2.

If *sense* = 'N' or 'E', *iwork* is not referenced by the subroutine.

Specified as: an area of storage containing logical values of the data type indicated in Table 209 on page 906.

bwork

is a storage work area of size *n*. If *sense* = 'N', *bwork* is not referenced by the subroutine.

Specified as: an area of storage containing numbers of the data type indicated in Table 209 on page 906.

On Return

a

is the updated general matrix ***A*** of order *n*. On output, A is overwritten; that is, the original input is not preserved.

- For SGGEVX, DGGEVX, CGGEVX, and ZGGEVX, if *jobvl* = 'V' or *jobvr* = 'V' and *balanc* ≠ 'N', A contains the Schur form of the balanced matrices.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 209 on page 906.

b

is the updated general matrix ***B*** of order *n*. On output, B is overwritten; that is, the original input is not preserved.

- For SGGEVX, DGGEVX, CGGEVX, and ZGGEVX, if *jobvl* = 'V' or *jobvr* = 'V' and *balanc* ≠ 'N', B contains the Schur form of the balanced matrices.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 209 on page 906.

alphar

is the vector of length *n*, containing the real part of the numerators of the eigenvalues. For details, see "Function" on page 914.

Returned as: an array of (at least) length *n*, containing numbers of the data type indicated in Table 209 on page 906.

alphai

is the vector of length *n*, containing the imaginary part of the numerators of the eigenvalues. For details, see "Function" on page 914

Returned as: an array of (at least) length *n*, containing numbers of the data type indicated in Table 209 on page 906.

alpha

is the vector α of length *n*, containing the numerators of the eigenvalues. For details, see "Function" on page 914

Returned as: an array of (at least) length *n*, containing numbers of the data type indicated in Table 209 on page 906.

beta

is the vector β of length *n*, containing the denominators of the eigenvalues. For details, see "Function" on page 914

Returned as: an array of (at least) length *n*, containing numbers of the data type indicated in Table 209 on page 906.

vl

contains the left eigenvectors.

- For SGGEV, DGGEV, SGGEVX, and DGGEVX:
  - If *jobvl* = 'V', the left eigenvectors are stored in the columns of *vl*, in the same order as their eigenvalues.
    - If the *j*-th eigenvalue is real or complex, then the *j*-th column of *vl* contains its eigenvector.
    - If the *j*-th and (*j*+1)-th eigenvalues form a complex conjugate pair, then the *j*-th and (*j*+1)-th columns of *vl* contain the real and imaginary parts of the eigenvector corresponding to the *j*-th eigenvalue. The conjugate of this eigenvector is the eigenvector for the (*j*+1)-th eigenvalue.
  - If *jobvl* = 'N', *vl* is not referenced.
- For CGGEV, ZGGEV, CGGEVX, and ZGGEVX:
  - If *jobvl* = 'V', the left eigenvectors are stored in the columns of *vl*, in the same order as their eigenvalues.
  - If *jobvl* = 'N', *vl* is not referenced.

Returned as: an array of size (*ldvl*, *n*) containing numbers of the data type indicated in Table 209 on page 906.

vr

contains the right eigenvectors.

- For SGGEV, DGGEV, SGGEVX, and DGGEVX:
  - If *jobvr* = 'V', the right eigenvectors are stored in the columns of *vr*, in the same order as their eigenvalues.
    - If the *j*-th eigenvalue is real or complex, then the *j*-th column of *vr* contains its eigenvector.
    - If the *j*-th and (*j*+1)-th eigenvalues form a complex conjugate pair, then the *j*-th and (*j*+1)-th columns of *vr* contain the real and imaginary parts of the eigenvector corresponding to the *j*-th eigenvalue. The conjugate of this eigenvector is the eigenvector for the (*j*+1)-th eigenvalue.

- If *jobvr* = 'N', *vr* is not referenced.

- For CGGEV, ZGGEV, CGGEVX, and ZGGEVX:
  - If *jobvr* = 'V', the right eigenvectors are stored in the columns of *vr*, in the same order as their eigenvalues.
  - If *jobvr* = 'N', *vr* is not referenced.

Returned as: an array of size (*ldvr*, *n*) containing numbers of the data type indicated in Table 209 on page 906.

ilo

has the following meaning:

If *balanc* = 'N', *ilo* = 1.

Otherwise, the value of *ilo* is determined when ***A*** is balanced.

The balanced $\boldsymbol{a}_{ij}$ = 0 if $i > j$ and j = 1, ..., (*ilo*-1) or i = (*ihi*+1), ..., *n*.

Returned as: an integer; 1 ≤ *ilo* ≤ *n*.

ihi

has the following meaning:

If *balanc* = 'N', *ihi* = *n*.

Otherwise, the value of *ihi* is determined when ***A*** is balanced.

The balanced $\boldsymbol{a}_{ij}$ = 0 if $i > j$ and j = 1, ..., (*ilo*-1) or i = (*ihi*+1), ..., *n*.

Returned as: an integer; 1 ≤ *ihi* ≤ *n*.

lscale

contains the details of the permutations and scaling factors applied to the left side when balancing ***A*** and ***B***.

If pl_j is the index of the row and column interchanged with row and column *j*, and dl_j is the scaling factor applied to row and column *j*, then:

- $lscale_j = pl_j$, for *j* = 1, ..., (*ilo*-1)
- $lscale_j = dl_j$, for *j* = *ilo*, ..., *ihi*
- $lscale_j = pl_j$, for *j* = (*ihi*+1), ..., *n*

Returned as: a one-dimensional array of (at least) length *n* containing numbers of the data type indicated in Table 209 on page 906.

rscale

contains the details of the permutations and scaling factors applied to the right side when balancing ***A*** and ***B***.

If pr_j is the index of the row and column interchanged with row and column *j*, and dr_j is the scaling factor applied to row and column *j*, then:

- $rscale_j = pr_j$, for *j* = 1, ..., (*ilo*-1)
- $rscale_j = dr_j$, for *j* = *ilo*, ..., *ihi*
- $rscale_j = pr_j$, for *j* = (*ihi*+1), ..., *n*

Returned as: a one-dimensional array of (at least) length *n* containing numbers of the data type indicated in Table 209 on page 906.

abnrm

is the one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).

Returned as: a number of the data type indicated in Table 209 on page 906; *abnrm* ≥ 0.

bbnrm

is the one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).

Returned as: a number of the data type indicated in Table 209 on page 906; *bbnrm* ≥ 0.

rconde

contains the computed reciprocal condition numbers of the eigenvalues, where $rconde_j$ is the reciprocal condition number of the *j*th eigenvalue.

Returned as: an array of dimension *n* containing numbers of the data type indicated in Table 209 on page 906.

rcondv

contains the computed reciprocal condition numbers of the eigenvectors, where $rcondv_j$ is the reciprocal condition number of the *j*th right eigenvector.

Returned as: an array of dimension *n* containing numbers of the data type indicated in Table 209 on page 906.

work

is the work area used by this subroutine if *lwork* ≠ 0, where:

If *lwork* ≠ 0 and *lwork* ≠ -1, its size is (at least) of length *lwork*.

If *lwork* = -1, its size is (at least) of length 1.

Returned as: an area of storage, where:

If *lwork* ≥ 1 or *lwork* = -1, then $work_1$ is set to the optimal *lwork* value and contains numbers of the data type indicated in Table 209 on page 906.

Except for $work_1$, the contents of *work* are overwritten on return.

info

If *info* = 0, the subroutine completed successfully.

If 1 ≤ *info* ≤ *n*, the QZ algorithm failed to compute all the eigenvalues, and no eigenvectors were computed. However:

- For SGGEV, DGGEV, SGGEVX, DGGEVX, LAPACKE_sggev, LAPACKE_dggev, LAPACKE_sggevx, and LAPACKE_dggevx, $alphar_j$, $alphai_j$, $beta_j$ are valid for *j* = (*info*+1,...,*n*)
- For CGGEV, ZGGEV, CGGEVX, ZGGEVX, LAPACKE_cggev, LAPACKE_zggev, LAPACKE_cggevx, and LAPACKE_zggevx, $alpha_j$ and $beta_j$ are valid for *j* = (*info*+1,...,*n*)

If *info* = *n* + 1, the eigenvalues failed to converge in the computation of shifts.

If *info* = *n* + 2, the eigenvectors failed to converge because the 2-by-2 block did not have a complex eigenvalue.

Returned as:

- For SGGEV, DGGEV, CGGEV, ZGGEV, SGGEVX, DGGEVX, and ZGGEVX, returned as: an integer; info ≥ 0.
- For LAPACKE_sggev, LAPACKE_dggev, LAPACKE_cggev, LAPACKE_zggev, LAPACKE_sggevx, LAPACKE_dggevx, LAPACKE_cggevx, and LAPACKE_zggevx returned as an integer function value; info ≥ 0.

Notes

1. The vectors and matrices used in the computation must have no common elements; otherwise, results are unpredictable.
2. These subroutines accept lowercase letters for the *balanc*, *jobvl*, *jobvr*, and *sense* arguments.
3. When you specify *jobvl* = 'N', you must specify a dummy argument for *vl*.

4. When you specify *jobvr* = 'N', you must specify a dummy argument for *vr*.
5. These subroutines accept lowercase letters for the *jobvl* and *jobvr* arguments.
6. When you specify *sense* = 'N', you must specify a dummy argument for *rconde*.
7. When you specify *sense* = 'N' or 'E', you must specify dummy arguments for *rcondv* and *iwork*.
8. In your C program, the *ilo*, *ihi*, *abnrm*, *info* arguments must be passed by reference.
9. In your C program, the *info* arguments must be passed by reference.
10. For best performance, specify *lwork* = 0.
11. The eigenvalue quotients might easily over- or underflow, and β might be zero. However, α is always less than and usually comparable with NORM(A) in magnitude, and β is always less than and usually comparable with NORM(B) in magnitude.

Function

- SGGEV, DGGEV, CGGEV, and ZGGEV compute the eigenvalues and, optionally, the left and/or right eigenvectors of a general matrix generalized eigenproblem.
- SGGEVX, DGGEVX, CGGEVX, and ZGGEVX compute the eigenvalues and, optionally, right eigenvectors, left eigenvectors, reciprocal condition numbers for eigenvalues, and reciprocal condition numbers for right eigenvectors of a general matrix generalized eigenproblem.

For the left eigenvectors:

$vl^{\mathbf{H}}\boldsymbol{A} = \lambda vl^{\mathbf{H}}\boldsymbol{B}$

For the right eigenvectors:

$\boldsymbol{A}vr = \lambda\boldsymbol{B}vr$

The eigenvalues are returned in two parts, α and β, where:

For SGGEV, DGGEV, SGGEVX, and DGGEVX, α and β are returned in vectors *alphar*, *alphai*, and *beta*, where *alphar* contains the real part of α and *alphai* contains the imaginary part of α.

- If $alphai_j = 0$, then the *j*-th eigenvalue is real.
- If $alphai_j > 0$, then the *j*-th and (*j*+1)-th eigenvalues are a complex conjugate pair.

For CGGEV, ZGGEV, CGGEVX, and ZGGEVX, α and β are returned in vectors *alpha* and *beta*.

For SGGEV, DGGEV, SGGEVX, and DGGEVX:

- If $alphai_j = 0$, then the *j*-th eigenvalue is real:
  - $\alpha_j = alphar_j$
  - $\beta_j = beta_j$
- If $alphai_j > 0$, then the *j*-th and (*j*+1)-th eigenvalues α_{j+1} are a complex conjugate pair:
  - $\alpha_j = (alphar_j, alphai_j)$
  - $\beta_j = beta_j$
  - $\alpha_{j+1} = (alphar_j, -alphai_j)$
  - $\beta_{j+1} = beta_j$

For CGGEV, ZGGEV, CGGEVX, and ZGGEVX:

- $\alpha_j = alpha_j$
- $\beta_j = beta_j$

Left eigenvectors are returned in matrix ***VL*** and right eigenvectors are returned in matrix ***VR***.

For SGGEV, DGGEV, CGGEV, and ZGGEV, the computation involves the following steps:

1. If necessary, scale the matrices ***A*** and ***B***.

2. Balance the matrices ***A*** and ***B***.
3. Reduce the balanced matrix ***A*** to an upper Hessenberg matrix and reduce the balanced matrix ***B*** to an upper triangular form.
4. Compute the eigenvalues of the Hessenberg-triangular pair, using the QZ algorithm.
5. If desired, compute the eigenvectors.
6. Undo balancing.
7. If necessary, undo scaling.

For SGGEVX, DGGEVX, CGGEVX, and ZGGEVX, the computation involves the following steps:

1. If necessary, scale the matrices ***A*** and ***B***.
2. Balance the matrices ***A*** and ***B***.
3. Reduce the balanced matrix ***A*** to triangular form.
4. Apply the unitary transformation to matrix ***A***.
5. Reduce the matrix pair (***A***,***B***) to generalized Hessenberg form.
6. Compute the eigenvalues and eigenvectors using the QZ algorithm and estimate condition numbers, if desired.
7. Undo balancing.
8. If necessary, undo scaling.

If *n* = 0, no computation is performed and the subroutine returns after doing some parameter checking.

For more information, see references [“15” on page 1254], [“16” on page 1254], “49” on page 1255], [“53” on page 1256], [“57” on page 1256], [“67” on page 1257], [“68” on page 1257], [“71” on page 1257], [“75” on page 1257], [“76” on page 1257], [“87” on page 1257], [“88” on page 1257], [“102” on page 1258], [“108” on page 1258], [“118” on page 1259], [“120” on page 1259], and [“121” on page 1259].

Error conditions

Resource Errors

lwork = 0, and unable to allocate work area.

Computational Errors

1. If 1 ≤ *info* ≤ *n*, the QZ algorithm failed to compute all the eigenvalues, and no eigenvectors were computed.
2. If *info* = *n* + 1, the eigenvalues failed to converge in the computation of shifts.
3. If *info* = *n* + 2, the eigenvectors failed to converge because the 2-by-2 block did not have a complex eigenvalue.

Input-Argument Errors

1. *balanc* ≠ 'N', 'S', 'P', or 'B'
2. *jobvl* ≠ 'N', or 'V'
3. *jobvr* ≠ 'N', or 'V'
4. *sense* ≠ 'N', 'E', 'V', or 'B'
5. *n* < 0
6. *lda* ≤ 0
7. *n* > *lda*
8. *ldb* ≤ 0
9. *n* > *ldb*

10. *ldvl* ≤ 0
11. *n* > *ldvl* and *jobvl* = 'V'
12. *ldvr* ≤ 0
13. *n* > *ldvr* and *jobvr* = 'V'
14. *jobvl* ='V' and *ldvl* ≤ 0
15. *jobvr* ='V' and *ldvr* ≤ 0
16. *jobvl* ='V' and *ldvl* < *n*
17. *jobvr* ='V' and *ldvr* < *n*
18. *lwork* ≠ 0 and *lwork* ≠ -1 and *lwork* < the minimum required value.
19. The size of a work array is greater than 2147483647 when 32-bit integers are used.

Examples

Example 1

This example shows how to find the eigenvalues only of a long-precision real generalized eigenproblem (***A***, ***B***).

Note:

1. *ldvl* and *ldvr* are set to 1 to avoid an error condition.
2. On output, matrices ***A*** and ***B*** are overwritten.
3. Because *lwork* = 0, the subroutine dynamically allocates `WORK`.

Call Statement and Input:

```
           JOBVL JOBVR  N  A  LDA  B  LDB ALPHAR  ALPHAI  BETA  VL  LDVL VR LDVR  WORK  LWORK  INFO
             |     |    |  |   |   |   |   |        |      |    |    |   |   |     |     |      |
CALL DGGEV( 'N',  'N',  3, A,  3,  B,  3, ALPHAR, ALPHAI, BETA, VL,   1, VR,  1,  WORK,   0,   INFO)
```

```
A    =  | 10.0    1.0    2.0 |
        |  1.0    2.0   -1.0 |
        |  1.0    1.0    2.0 |
```

```
B    =  |  1.0    2.0    3.0 |
        |  4.0    5.0    6.0 |
        |  7.0    8.0    9.0 |
```

Output:

```
           |  2.092346 |
ALPHAR  =  | -4.789188 |
           |  4.490731 |
```

```
           | 0.000000 |
ALPHAI  =  | 0.000000 |
           | 0.000000 |
```

```
           | 12.711351 |
BETA    =  |  0.998541 |
            |  0.000000 |
```

```
INFO   =   0
```

Example 2

This example shows how to find the eigenvalues and the left and right eigenvectors of a long-precision real generalized eigenproblem (***A***, ***B***).

Note:

1. On output, matrices ***A*** and ***B*** are overwritten.
2. Because *lwork* = 0, the subroutine dynamically allocates WORK.
3. This matrix is used on page 263 in referenced text ["6" on page 1253].

Call Statement and Input:

```
          JOBVL  JOBVR   N   A   LDA   B   LDB  ALPHAR   ALPHAI   BETA   VL   LDVL  VR  LDVR   WORK   LWORK  INFO
            |      |     |   |    |    |    |      |        |       |     |     |    |    |      |       |      |
CALL DGGEV( 'V',   'V',  5,  A,   5,   B,   5,  ALPHAR,  ALPHAI,  BETA,  VL,    1,  VR,   1,   WORK,    0,   INFO)
```

```
A    =  |  2.0    3.0    4.0    5.0     6.0|
        |  4.0    4.0    5.0    6.0     7.0|
        |  0.0    3.0    6.0    7.0     8.0|
        |  0.0    0.0    2.0    8.0     9.0|
        |  0.0    0.0    0.0    1.0    10.0|
```

```
B    =  |  1.0   -1.0   -1.0   -1.0    -1.0|
        |  0.0    1.0   -1.0   -1.0    -1.0|
        |  0.0    0.0    1.0   -1.0    -1.0|
        |  0.0    0.0    0.0    1.0    -1.0|
        |  0.0    0.0    0.0    0.0     1.0|
```

Output:

```
            |  7.950050 |
            | -0.277338 |
ALPHAR   =  |  2.149669 |
            |  6.720718 |
            | 10.987556 |
```

```
            | 0.000000 |
            | 0.000000 |
ALPHAI   =  | 0.000000 |
            | 0.000000 |
            | 0.000000 |
```

```
           |  0.374183 |
           |  1.480299 |
BETA    =  |  1.636872 |
           |  1.213574 |
           |  0.908837 |
```

```
          | -0.003801   1.000000  -0.778812  -0.032145  -0.005114 |
          | -0.018291  -0.546838   0.133707  -0.028432  -0.012900 |
VL    =   | -0.074427  -0.174281   1.000000   0.076908  -0.009062 |
          | -0.279354  -0.051285  -0.696402   0.285338   0.123783 |
          | -1.000000  -0.026864   0.282389  -1.000000   1.000000 |
```

```
          | -1.000000   0.483408  -0.540696  -1.000000  -1.000000 |
          | -0.565497  -1.000000  -0.684441  -0.722065  -0.610415 |
VR    =   | -0.180429   0.661372   1.000000   0.089003  -0.116987 |
          | -0.034182  -0.180646  -0.363671   0.223599   0.038979 |
          | -0.003039   0.017732   0.041865  -0.050111   0.018653 |
```

```
INFO   =   0
```

Example 3

This example shows how to find the eigenvalues only of a long-precision complex generalized eigenproblem (***A***, ***B***).

Note:

1. *ldvl* and *ldvr* have been set to 1 to avoid an error condition.
2. On output, matrices ***A*** and ***B*** are overwritten.
3. Because *lwork* = 0, the subroutine dynamically allocates WORK.

Call Statement and Input:

```
            JOBVL JOBVR  N  A  LDA  B  LDB ALPHAR  ALPHAI  BETA  VL  LDVL VR LDVR  WORK  LWORK RWORK
INFO
              |     |    |  |   |   |   |     |       |      |    |    |   |   |     |     |     |      |
 CALL ZGGEV( 'N',  'N',  4, A,  4,  B,  4, ALPHAR, ALPHAI, BETA, VL,   1, VR,  1,  WORK,   0,  RWORK,
INFO)
```

```
          | ( 2, 4) ( 1, 6) ( 2, 8) ( 4, 4) |
 A    =   | ( 3, 3) ( 6, 1) ( 5, 3) ( 0, 0) |
          | ( 5, 1) ( 8, 5) ( 3, 2) ( 8, 5) |
          | ( 7, 6) ( 3, 7) ( 2, 1) ( 5, 4) |
```

```
          | ( 0, 0) ( 1, 0) ( 0, 0) ( 0, 0) |
B     =   | ( 0, 0) ( 0, 0) ( 1, 0) ( 0, 0) |
          | ( 0, 0) ( 0, 0) ( 0, 0) ( 1, 0) |
          | ( 1, 0) ( 0, 0) ( 0, 0) ( 0, 0) |
```

Output:

```
             | (15.8864, 15.0474) |
ALPHA    =   | ( 7.0401,  2.0585) |
             | ( 1.7083,  4.1133) |
             | (-3.6348, -1.2193) |
```

```
             | (1.0, 0.0) |
BETA     =   | (1.0, 0.0) |
             | (1.0, 0.0) |
             | (1.0, 0.0) |
```

```
INFO   =   0
```

Example 4

This example shows how to find the eigenvalues and the left and right eigenvectors of a long-precision complex eigenproblem (***A***, ***B***).

Note:

1. On output, matrices ***A*** and ***B*** are overwritten.
2. Because *lwork* = 0, the subroutine dynamically allocates `WORK`.
3. This matrix is used on page 263 in referenced text [“6” on page 1253].

Call Statement and Input:

```
            JOBVL JOBVR  N  A  LDA  B  LDB ALPHAR  ALPHAI  BETA  VL  LDVL VR LDVR  WORK  LWORK RWORK
INFO
              |     |    |  |   |   |   |     |       |      |    |    |   |   |     |     |     |      |
 CALL ZGGEV( 'V',  'V',  3, A,  3,  B,  3, ALPHAR, ALPHAI, BETA, VL,   3, VR,  3,  WORK,   0,  RWORK,
INFO)
```

```
A     =  |  ( 1.0, 2.0)   ( 3.0, 4.0)   (21.0,22.0)  |
         |  (43.0,44.0)   (13.0,14.0)   (15.0,16.0)  |
         |  ( 5.0, 6.0)   ( 7.0, 8.0)   (25.0,26.0)  |
```

```
B     =  |  ( 2.0, 0.0)   ( 0.0,-1.0)   ( 0.0, 0.0)  |
         |  ( 0.0, 1.0)   ( 2.0, 0.0)   ( 0.0, 0.0)  |
         |  ( 0.0, 0.0)   ( 0.0, 0.0)   ( 3.0, 0.0)  |
```

Output:

```
            | ( 15.863783, 41.115283) |
ALPHA    =  | (-12.917205, 19.973815) |
            | (  3.215518, -4.912439) |
```

```
            | (  1.668461,  0.0) |
BETA     =  | (  2.024212,  0.0) |
            | (  2.664836,  0.0) |
```

```
           | ( 0.0634, -0.8686) ( 0.8988, -0.1012) (-0.6456, -0.3544) |
VL      =  | (-0.3652, -0.3826) ( 0.0108, -0.3479) ( 0.0001, -0.0852) |
           | ( 0.3605, -0.6395) (-0.2029,  0.6348) ( 0.4537,  0.4128) |
```

```
           | ( 0.3799, -0.1986) ( 0.2712, -0.0943) (-0.1470,  0.1422) |
VR      =  | ( 0.0132, -0.9868) (-0.5085, -0.4915) ( 0.3239, -0.6761) |
           | (-0.0976, -0.2383) (-0.0633,  0.1388) (-0.0395,  0.1663) |
```

```
INFO   =   0
```

Example 5

This example shows how to find the eigenvalues, the left and right eigenvectors, and the reciprocal condition numbers of the eigenvalues and right eigenvectors of a long-precision real generalized eigenproblem (***A***,***B***).

Note:

1. On output, matrices ***A*** and ***B*** are overwritten.
2. Because *lwork* = 0, the subroutine dynamically allocates `WORK`.

Call Statement and Input:

```
             BALANC JOBVL JOBVR SENSE N  A  LDA B LDB ALPHAR  ALPHAI  BETA  VL LDVL VR LDVR  ILO   IHI
```

```
LSCALE  RSCALE
              |     |     |     |   |  |   |  |  |     |       |      |    |   |   |   |    |
|       |       |
 CALL DGGEVX( 'P',  'V',  'V',  'B', 6, A,  6, B, 6, ALPHAR, ALPHAI, BETA, VL,  6, VR,  6,  ILO,  IHI,
LSCALE, RSCALE,

             ABNRM  BBNRM  RCONDE  RCONDV  WORK  LWORK IWORK  BWORK  INFO
               |      |      |       |      |     |     |      |      |
             ABNRM, BBNRM, RCONDE, RCONDV, WORK,   0,  IWORK  BWORK, INFO)
```

```
          |   9.0  98.0   1.0  96.0  30.0  94.0  |
          |   1.0  92.0  91.0  90.0   8.0  88.0  |
A    =    |  66.0   1.0  68.0   0.0  40.0   1.0  |
          |   0.0   1.0  64.0  50.0  66.0   1.0  |
          |  10.0   0.0  10.0  86.0   8.0  10.0  |
          |   8.0   1.0  20.0  12.0  20.0  15.0  |
```

```
          |  21.0  24.0   9.0  22.0  25.0  18.0  |
          |  19.0   9.0  20.0  25.0  15.0   6.0  |
B    =    |   5.0   4.0   9.0  14.0  2 .0  21.0  |
          |  18.0   7.0   8.0  24.0   9.0   1.0  |
          |  15.0  20.0   9.0   1.0  14.0  24.0  |
          |  21.0  23.0   9.0  14.0  88.0  88.0  |
```

Output:

```
             |  64.6956  |
             |  55.0897  |
ALPHAR   =   | -28.2595  |
             |  -0.0472  |
             |  -0.0437  |
             |  13.8550  |
```

```
             |  70.4260  |
             | -59.9693  |
ALPHAI   =   |   0.0000  |
             |  28.9059  |
             | -26.7715  |
             |   0.0000  |
```

```
             |  15.2997  |
             |  13.0280  |
BETA     =   |  15.2267  |
             |  26.2607  |
             |  24.3216  |
             |  47.4385  |
```

```
           | -0.8391    0.3207   -0.5181    0.5129     0.1237    0.1230 |
           | -0.1545    0.6066    0.0489   -0.4870     0.0269   -0.0401 |
VL     =   | -0.5621    0.6337   -1.0000   -0.3687     0.5350    0.1137 |
           | -0.2946    0.2303   -0.5078    0.4050    -0.2545    0.1577 |
           | -0.4076    0.5085   -0.4921   -0.1974    -0.0241    0.0910 |
           | -0.7701    0.8407   -1.0000   -0.4524     0.0892    0.9064 |
```

```
           |   0.8937   -0.6723   -0.1299   -0.2564   -0.3186    0.3349 |
           |   0.1062    0.0314    0.5842    0.0377   -0.0345   -0.0360 |
VR     =   |   1.0000    0.4950    0.1818   -0.5869    0.0451   -0.3523 |
           |  -0.9722    0.0642   -0.0423    0.1502    0.1791   -0.0818 |
           |  -0.0277   -0.1835   -0.0507   -0.1421   -0.1836    0.2469 |
            |   0.0654   -0.8883    0.0497   -0.1390    0.1957    1.0000 |
```

```
             |  27.2117  |
             |  27.2117  |
RCONDE   =   |  11.8821  |
             |   6.6118  |
             |   6.6118  |
             |  26.3358  |
```

```
             |   0.1657  |
             |   0.1657  |
RCONDV   =   |   5.2466  |
             |   1.8139  |
             |   1.8139  |
             |   6.5921  |
```

```
INFO   =   0
```

Example 6

This example shows how to find the eigenvalues, the left and right eigenvectors, and the reciprocal condition numbers of the eigenvalues and right eigenvectors of a long-precision complex generalized eigenproblem (***A***,***B***).

Note:

1. On output, matrices ***A*** and ***B*** are overwritten.
2. Because *lwork* = 0, the subroutine dynamically allocates WORK.

Call Statement and Input:

```
            BALANC JOBVL JOBVR SENSE N  A  LDA B LDB ALPHA  BETA  VL LDVL VR LDVR  ILO   IHI   LSCALE
RSCALE
               |     |     |     |   |  |   |  |  |    |      |    |   |   |   |    |     |
|       |
 CALL ZGGEVX( 'S',  'V',  'V',  'B', 3, A,  3, B, 3, ALPHA, BETA, VL,  3, VR,  3,  ILO,  IHI,  LSCALE,
RSCALE,

             ABNRM  BBNRM  RCONDE  RCONDV  WORK  LWORK RWORK  IWORK  BWORK  INFO
               |       |      |       |      |     |     |      |      |
             ABNRM, BBNRM, RCONDE, RCONDV, WORK,   0,  RWORK, IWORK  BWORK, INFO)
```

```
          |   (10.0, 1.0)  ( 1.0, 0.0)  ( 2.0,-1.0) |
A    =    |   ( 1.0, 2.5)  ( 2.0,-1.0)  (-1.0, 0.0) |
          |   ( 1.0, 0.0)  ( 1.0, 0.0)  ( 2.0, 1.5) |
```

```
          |   ( 1.0, 0.0)  ( 2.0, 0.0)  ( 3.0, 0.0) |
B    =    |   ( 4.0, 1.0)  ( 5.0,-5.0)  ( 6.0,-2.0) |
          |   ( 7.0, 5.0)  ( 8.0,-8.0)  ( 9.0, 0.5) |
```

Output:

```
             |  (-6.6742,  -0.5632) |
ALPHA    =   |  ( 2.1073,   1.6591) |
             |  ( 1.2922,  -3.5963) |
```

```
             |  (  0.4561, 0.0000) |
BETA     =  |  ( 17.7898, 0.0000) |
            |  (  1.9190, 0.0000) |
```

```
          |  (0.7337,  0.0000) (0.1686,  0.0647) (0.9185,  0.0814) |
VL      = |  ( 0.0000, 0.0000) ( 0.0000, 0.0000) ( 1.0000, 0.0000) |
          |  ( 0.0000, 0.0000) ( 0.0000, 0.0000) ( 0.0000, 0.0000) |
```

```
          |  ( 1.0000, 0.0000) ( 0.0000, 0.0000) ( 0.0000, 0.0000) |
VR      = |  (-0.1260, -0.8739) (0.1299, -0.2557) (0.6423, 0.0000) |
          |  ( 0.0000, 0.0000) ( 0.0000, 0.0000) ( 0.0000, 0.0000) |
```

```
            |   4.1319  |
RCONDE   =  |  15.2697  |
            |   2.4329  |
```

```
            |   3.0489  |
RCONDV   =  |   4.1998  |
            |   2.3870  |
```

```
INFO   =   0
```

>|SSYGVD, DSYGVD, CHEGVD, ZHEGVD (Eigenvalues, and Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem Using a Divide-and-Conquer Algorithm)

Purpose

These subroutines compute eigenvalues, and optionally, the eigenvectors of a positive definite real symmetric or complex Hermitian generalized eigenproblem by using a divide-and-conquer algorithm that solves one of the following problems:

- If *itype* = 1, the problem is $\boldsymbol{Ax} = \lambda \boldsymbol{Bx}$
- If *itype* = 2, the problem is $\boldsymbol{ABx} = \lambda \boldsymbol{x}$
- If *itype* = 3, the problem is $\boldsymbol{BAx} = \lambda \boldsymbol{x}$

In the formulas:

- ***A*** represents the real symmetric or complex Hermitian matrix ***A***.
- ***B*** represents the positive definite real symmetric or complex Hermitian matrix ***B***.

The data type for each array and matrix, and the corresponding subroutines are listed in the following table:

Table 210: Data types.

A*, *B*, *work	***w*, *rwork***	**Subroutine**
Short-precision real	Short-precision real	SSYGVD$^{\Delta}$
Long-precision real	Long-precision real	DSYGVD$^{\Delta}$

Table 210: Data types. (continued)

A*, *B*, *work	***w*, *rwork***	**Subroutine**
Short-precision complex	Short-precision real	CHEGVD$^{\Delta}$
Long-precision complex	Long-precision real	ZHEGVD$^{\Delta}$
$^{\Delta}$LAPACK		

Syntax

Table 211: Syntax for supported computing languages.

Language	**Syntax**
Fortran	CALL SSYGVD | DSYGVD (*itype*, *jobz*, *uplo*, *n*, *a*, *lda*, *b*, *ldb*, *w*, *work*, *lwork*, *iwork*, *liwork*, *info*) CALL CHEGVD | ZHEGVD (*itype*, *jobz*, *uplo*, *n*, *a*, *lda*, *b*, *ldb*, *w*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*)

C and C++	ssygvd | dsygvd (*itype*, *jobz*, *uplo*, *n*, *a*, *lda*, *b*, *ldb*, *w*, *work*, *lwork*, *iwork*, *liwork*, *info*);
	chegvd | zhegvd (*itype*, *jobz*, *uplo*, *n*, *a*, *lda*, *b*, *ldb*, *w*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*);

LAPACKE	info = LAPACKE_ssygvd | LAPACKE_dsygvd (*matrix_layout*, *itype*, *jobz*, *uplo*, *n*, *a*, *lda*, *b*, *ldb*, *w*);
	info = LAPACKE_chegvd | LAPACKE_zhegvd (*matrix_layout*, *itype*, *jobz*, *uplo*, *n*, *a*, *lda*, *b*, *ldb*, *w*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

itype

specifies the problem type, where:

If *itype* = 1, the problem is $\boldsymbol{Ax} = \lambda \boldsymbol{Bx}$.

If *itype* = 2, the problem is $\boldsymbol{ABx} = \lambda \boldsymbol{x}$.

If *itype* = 3, the problem is $\boldsymbol{BAx} = \lambda \boldsymbol{x}$.

Specified as: an integer; *itype* = 1, 2, or 3.

jobz

indicates the type of computation to be performed, where:

If *jobz* = 'N', only eigenvalues are computed.

If *jobz* = 'V', both eigenvalues and eigenvectors are computed.

Specified as: a single character; *jobz* = 'N' or 'V'.

uplo
indicates whether the upper or lower triangular parts of the matrices ***A*** and ***B*** are referenced, where:

If *uplo* = 'U', the upper triangular part is referenced.

If *uplo* = 'L', the lower triangular part is referenced.

Specified as: a single character; *uplo* = 'U' or 'L'.

n
is the order of matrices ***A*** and ***B*** that is used in the computation.

Specified as: an integer; $n \geq 0$.

a
is the real symmetric or complex Hermitian matrix ***A*** of order *n*.

If *uplo* = 'U', the matrix is stored in upper storage mode.

If *uplo* = 'L', the matrix is stored in lower storage mode.

Specified as: an *lda* by (at least) *n* array, which contains numbers of the data type indicated in Table 210 on page 922.

lda
is the leading dimension of the array specified for the matrix ***A***.

Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

b
is the positive definite real symmetric or complex Hermitian matrix ***B*** of order *n*.

If *uplo* = 'U', the matrix is stored in upper storage mode.

If *uplo* = 'L', the matrix is stored in lower storage mode.

Specified as: an *ldb* by (at least) *n* array, which contains numbers of the data type indicated in Table 210 on page 922.

ldb
is the leading dimension of the array specified for the matrix ***B***.

Specified as: an integer; *ldb* > 0 and *ldb* ≥ *n*.

work
is a work area used by these subroutines, where:

If *lwork* = 0, *work* is ignored.

If *lwork* ≠ 0, the size of *work* is determined as follows:

- If *lwork* ≠ -1, *work* is (at least) of length *lwork*.
- If *lwork* = -1, *work* is (at least) of length 1.

Specified as: an area of storage that contains numbers of the data type indicated in Table 210 on page 922.

lwork
is the number of elements in array `WORK`.

Specified as: an integer; where:

- If *lwork* = -1, the subroutine performs a workspace query and returns the optimal required size of the `WORK` array in the $work_1$ entry. No computation is performed and the subroutine returns after checking for errors.
- If *lwork* = 0, the subroutine dynamically allocates the workspace that is required during this computation. The work area is deallocated before returning the control to the calling program.
- If *lwork* > 0:

For SSYGVD and DSYGVD:

If $n \leq 1$, *lwork* ≥ 1.

If *jobz* = 'N' and $n > 1$, *lwork* ≥ $2n + 1$.

If *jobz* = 'V' and $n > 1$, *lwork* ≥ $1 + 6n + 2n^2$.

For CHEGVD and ZHEGVD:

If $n \leq 1$, *lwork* ≥ 1.

If *jobz* = 'N' and $n > 1$, *lwork* ≥ $n + 1$.

If *jobz* = 'V' and $n > 1$, *lwork* ≥ $2n + n^2$.

These formulas represent the minimum workspace that is required. For best performance, specify either *lwork* = -1 (to obtain the optimal required size) or *lwork* = 0 (to direct the subroutine to dynamically allocate the workspace).

rwork
See "On Return" on page 925.

lrwork
is the number of elements in array RWORK.

Specified as: an integer; where:

- If *lrwork* = -1, the subroutine performs a workspace query and returns the optimal required size of the RWORK array in the $work_1$ entry.
- If *lrwork* = 0, the subroutine dynamically allocates the workspace that is required during this computation. The work area is deallocated before returning the control to the calling program.
- If *lrwork* > 0:
  - If $n \leq 1$, *lrwork* ≥ 1.
  - If *jobz* = 'N' and $n > 1$, *lrwork* ≥ n.
  - If *jobz* = 'V' and $n > 1$, *lrwork* ≥ $1 + 5n + 2n^2$.

iwork
See "On Return" on page 925.

liwork
is the number of elements in array IWORK.

Specified as: an integer; where:

- If *liwork* = -1, the subroutine performs a workspace query and returns the optimal required size of the IWORK array in the $work_1$ entry.
- If *liwork* = 0, the subroutine dynamically allocates the workspace that is required during this computation. The work area is deallocated before returning the control to the calling program.
- If *liwork* > 0:
  - If $n \leq 1$, *liwork* ≥ 1.
  - If *jobz* = 'N' and $n > 1$, *liwork* ≥ 1.
  - If *jobz* = 'V' and $n > 1$, *liwork* ≥ $3 + 5n$.

info
See "On Return" on page 925.

On Return

a
is overwritten as follows:

- If *jobz* = 'V' and if *info* = 0, the matrix $\boldsymbol{A}$ contains the matrix $\boldsymbol{Z}$ of eigenvectors. The eigenvectors are normalized as follows:
  - if *itype* = 1 or 2, $\boldsymbol{Z}^{\boldsymbol{H}} \boldsymbol{B}\boldsymbol{Z} = \boldsymbol{I}$;
  - if *itype* = 3, $\boldsymbol{Z}^{\boldsymbol{H}}\boldsymbol{B}^{-1}\boldsymbol{Z} = \boldsymbol{I}$.

Returned as: an array of dimension *lda* by (at least) *n*, which contains numbers of the data type indicated in Table 210 on page 922.

b

contains the results of the Cholesky factorization.

If *uplo* = 'U', and if *info* ≤ *n*, the leading *n* by *n* upper triangular part of matrix $\boldsymbol{B}$ contains the triangular factor $\boldsymbol{U}$ from the Cholesky factorization $\boldsymbol{B} = \boldsymbol{U}^{\boldsymbol{H}}\boldsymbol{U}$.

If *uplo* = 'L', and if *info* ≤ *n*, the leading *n* by *n* lower triangular part of matrix $\boldsymbol{B}$ contains the triangular factor $\boldsymbol{L}$ from the Cholesky factorization $\boldsymbol{B} = \boldsymbol{L}\boldsymbol{L}^{\boldsymbol{H}}$.

Returned as: an array of dimension *ldb* by (at least) *n*, which contains numbers of the data type indicated in Table 210 on page 922.

w

is the vector $\boldsymbol{w}$, which contains the computed eigenvalues.

If *info* = 0, the eigenvalues are in ascending order.

Returned as: a one-dimensional array of (at least) length *n*, which contains numbers of the data type indicated in Table 210 on page 922.

work

is a work area used by this subroutine for computation.

If *lwork* ≠ 0 and *info* = 0, the $work_1$ entry in the `WORK` array returns the optimal *lwork* value.

Returned as: an area of storage, where:

If *lwork* ≥ 1 or *lwork* = -1, the $work_1$ entry is set to the optimal *lwork* value and all other elements of the *work* argument are overwritten.

rwork

is a work area used by this subroutine for computation.

If *lrwork* ≠ 0 and *info* = 0, the subroutine returns the optimal *lrwork* value.

Returned as: an area of storage of dimension max (1, *lrwork*).

iwork

is a work array.

If *liwork* ≠ 0 and *info* = 0, the $iwork_1$ entry returns the optimal *liwork* value.

Returned as: an integer array of dimension max (1, *liwork*).

info

has the following meaning:

If *info* = 0, the subroutine completed successfully.

If *info* ≤ *n*, and

- If *info* = *i* and *jobz* = 'N', the algorithm failed to converge. The *i* off-diagonal elements of an intermediate tridiagonal form did not converge to zero.
- If *info* = *i* and *jobz* = 'V', the algorithm failed to compute an eigenvalue while working on the submatrix located in rows and columns *info*/(*n* + 1) through mod(*info*, *n* + 1).

If *info* > *n* and if *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, the leading minor of order i of matrix $\boldsymbol{B}$ is not positive definite. The factorization of matrix $\boldsymbol{B}$ cannot be completed and eigenvalues or eigenvectors are not computed.

Returned as: an integer; *info* ≥ 0.

Notes:

1. These subroutines accept lowercase letters for the *jobz* and *uplo* arguments.
2. In your C program, the *info* argument must be passed by reference.
3. *a*, *b*, *w*, *work*, *rwork*, and *iwork* must have no common elements; otherwise, results are unpredictable.
4. For a description of how real symmetric matrices are stored in lower or upper storage mode, see "Lower Storage Mode" on page 83 or "Upper Storage Mode" on page 83.

   For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see "Complex Hermitian Matrix" on page 84.
5. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrices ***A*** and ***B*** are assumed to be zero. Therefore, do not set these values. On output, for matrix ***B***, they are set to zero.
6. For best performance, specify either *lwork* = -1 (to obtain the optimal required size) or *lwork* = 0 (to direct the subroutine to dynamically allocate the workspace).

Function

These subroutines compute all eigenvalues, and optionally, the eigenvectors of a positive definite real symmetric or complex Hermitian generalized eigenproblem by using the divide-and-conquer algorithm that solves one of the following problems:

- If *itype* = 1, the problem is $\boldsymbol{Ax} = \lambda \boldsymbol{Bx}$
- If *itype* = 2, the problem is $\boldsymbol{ABx} = \lambda \boldsymbol{x}$
- If *itype* = 3, the problem is $\boldsymbol{BAx} = \lambda \boldsymbol{x}$

In the formulas:

- ***A*** represents the real symmetric or complex Hermitian matrix ***A***.
- ***B*** represents the positive definite real symmetric or complex Hermitian matrix ***B***.

If eigenvectors are required, the subroutine uses a divide-and-conquer algorithm. The computation involves the following steps:

1. Compute the Cholesky factorization of matrix ***B***.
2. Reduce the generalized eigenproblem to the following standard symmetric eigenvalue problem, which is solved for the eigenvalues, and optionally, the eigenvectors.
3. Backtransform the eigenvectors to obtain the eigenvectors of the original problem.

Error conditions

Resource Errors

1. *lwork* = 0 and unable to allocate work area.

Computational Errors

1. The matrix ***B*** is not positive definite. See output argument *info* for more details.
2. The algorithm failed to converge.
3. The algorithm failed to compute an eigenvalue while working on the submatrix.

Input-Argument Errors

1. *itype* < 1 or *itype* > 3
2. *jobz* ≠ 'N' or 'V'
3. *uplo* ≠ 'U' or 'L'
4. *n* < 0
5. *lda* ≤ 1

6. *lda* < *n*
7. *ldb* ≤ 1
8. *ldb* < *n*
9. *lwork* ≠ 0 and *lwork* ≠ -1 and *lwork* < the minimum required value
10. *lrwork* ≠ 0 and *lrwork* ≠ -1 and *lrwork* < the minimum required value
11. *liwork* ≠ 0 and *liwork* ≠ -1 and *liwork* < the minimum required value
12. The size of a work array is greater than 2147483647 when 32-bit integers are used

Examples

Example 1

This example shows how to find the eigenvalues of a real symmetric positive generalized eigenproblem of the form: ***Ax*** = λ***Bx***. Matrices ***A*** and ***B*** are stored in upper storage mode.

Matrix ***A*** is:

```
                 ┌                 ┐
                 |  10    1    2   |
                 |   .    2   -1   |
                 |   .    .    2   |
                 └                 ┘
```

Matrix ***B*** is:

```
                 ┌                 ┐
                 |   1    0    0   |
                 |   .    1    0   |
                 |   .    .    1   |
                 └                 ┘
```

Call Statement and Input:

```
               ITYPE JOBZ UPLO   N  A   LDA B LDB   W  WORK   LWORK IWORK LIWORK  INFO
                 |     |    |    |  |   |   |  |    |   |      |      |     |      |
CALL DSYGVD ( 1,   'N', 'U', 3, A, 3,  B, 3,  W, WORK,  0,   IWORK,  0,  INFO)
```

Output:

Matrix ***A*** is overwritten.

```
          ┌         ┐
W =       |   0.5153  |
          |   2.9466  |
          |  10.5381  |
          └         ┘
```

```
          ┌                 ┐
          |   1    0    0   |
B =       |   .    1    0   |
          |   .    .    1   |
          └                 ┘
```

```
INFO = 0
```

Example 2

This example shows how to find the eigenvalues and eigenvectors of a real symmetric positive generalized eigenproblem of the form: ***Ax*** = λ***Bx***.

Matrix ***A*** is:

```
                 ┌                 ┐
                 |   6    4    4   |
                 |   .    6    1   |
                 |   .    .    6   |
                 └                 ┘
```

Matrix ***B*** is:

```
                 ┌                 ┐
                 |   1    0    0   |
                 |   .    1    0   |
                 |   .    .    1   |
                 └                 ┘
```

Call Statement and Input:

```
            ITYPE JOBZ UPLO N   A LDA   B LDB W   WORK LWORK IWORK LIWORK INFO
              |     |    |   |   |  |    |  |   |    |     |     |      |     |
CALL DSYGVD ( 2,  'V', 'U', 3, A, 3,  B, 3, W, WORK, 0,   IWORK, 0,   INFO)
```

Output:

```
                 ┌              ┐
                 |   0.8210916  |
          W =    |   5.000000   |
                 |  12.178908   |
                 └              ┘
```

```
                 ┌                                             ┐
                 |   -0.737578    0.4774816    0.4774816   |
          A =    |    0.0000000  -0.707106     0.7071067   |
                 |   -0.675261   -0.521546    -0.521546    |
                 └                                             ┘
```

```
                 ┌                 ┐
                 |   1    0    0   |
          B =    |   .    1    0   |
                 |   .    .    1   |
                 └                 ┘
```

```
INFO = 0
```

Example 3

This example shows how to find the eigenvalues of a real symmetric positive generalized eigenproblem of the form: ***BAx*** = λ***x***. Matrices ***A*** and ***B*** are stored in upper storage mode.

Matrix ***A*** is:

```
                 ┌                                         ┐
                 |  (10.0, 1.0)  (1.0, 0 )    (2.0, -1)   |
                 |  ( .       )  (2.0, -1)    (-1.0, 0)   |
                 |  ( .       )  ( .     )    (2.0, 1.5)  |
                 └                                         ┘
```

Matrix ***B*** is:

```
                 ┌                                           ┐
                 |  (1.0, 0.0)   (0.0, 0.0)   (0.0, 0.0)   |
                 |     ( . )     (1.0, 0.0)   (0.0, 0.0)   |
                 |     ( . )       ( . )      (1.0, 0.0)   |
                 └                                           ┘
```

Call Statement and Input:

```
           ITYPE JOBZ UPLO  N  A LDA  B LDB  W  WORK LWORK RWORK LRWORK IWORK LIWORK INFO
             |     |    |   |  |  |   |  |   |   |    |     |     |      |     |      |
CALL ZHEGVD ( 3,   'N', 'U', 3, A, 3,  B, 3,  W, WORK, 0,   RWORK, 0,   IWORK,   0,   INFO)
```

Output:

Matrix ***A*** is overwritten.

```
                ┌              ┐
                |   0.4537610  |
        W =     |   2.8973095  |
                |  10.648929   |
                └              ┘
```

```
                ┌                                            ┐
                |  (1.0, 0.0)   (0.0, 0.0)   (0.0, 0.0)   |
        B =     |     ( . )     (1.0, 0.0)   (0.0, 0.0)   |
                |     ( . )       ( . )      (1.0, 0.0)   |
                └                                            ┘
```

```
INFO = 0
```

Example 4

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian generalized eigenproblem of the form: ***ABx*** = λ***x***.

Matrix ***A*** is:

```
                ┌                                            ┐
                |  (6.0, 0.0)   (   .     )   (   .     )   |
                |  (4.0, 0.0)   (6.0, 0.0)   (   .     )   |
                |  (4.0, 0.0)   (1.0, 0.0)   (6.0, 0.0)   |
                └                                            ┘
```

Matrix ***B*** is:

```
                ┌                                          ┐
                |  (1.0, 0.0)  (   .     )  (   .     )   |
                |  (0.0, 0.0)  (1.0, 0.0)  (   .     )   |
                |  (0.0, 0.0)  (0.0, 0.0)  (1.0, 0.0)   |
                └                                          ┘
```

Call Statement and Input:

```
          ITYPE JOBZ UPLO N  A LDA B LDB W  WORK LWORK RWORK LRWORK IWORK LIWORK INFO
            |    |    |   |  |  |  |  |  |   |     |     |     |      |     |     |
CALL ZHEGVD ( 2,   'V', 'L', 3, A, 3, B, 3, W, WORK,  0,  RWORK,  0,   IWORK,  0,   INFO)
```

Output:

```
                ┌              ┐
                |   0.8210916  |
        W =     |   5.0000000  |
                |  12.178908   |
                └              ┘
```

```
                ┌                                                        ┐
                |  (-0.737578, 0.0) (0.47748165,0.0) (0.4774816,0.0)   |
        A =     |        (0.0, 0.0) (-0.7071067,0.0) (0.7071067,0.0)   |
                |  (-0.675261, 0.0) (-0.5215469,0.0) (-0.521546,0.0)   |
                └                                                        ┘
```

```
                ┌                                              ┐
                |  (1.0, 0.0)       ( . )        ( . )        |
        B =     |  (0.0, 0.0)     (1.0, 0.0)   (0.0, 0.0)    |
                |  (0.0, 0.0)     (0.0, 0.0)   (1.0, 0.0)   |
                └                                           ┘
```

```
INFO = 0
```

|<

SSYGV, DSYGV, CHEGV, ZHEGV, SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX (Eigenvalues, and Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem)

Purpose

These subroutines compute eigenvalues and, optionally, the eigenvectors of a positive definite real symmetric or complex Hermitian generalized eigenproblem:

- If *itype* = 1, the problem is $\boldsymbol{Ax} = \lambda \boldsymbol{Bx}$
- If *itype* = 2, the problem is $\boldsymbol{ABx} = \lambda \boldsymbol{x}$
- If *itype* = 3, the problem is $\boldsymbol{BAx} = \lambda \boldsymbol{x}$

In the formulas:

- ***A*** represents the real symmetric or complex Hermitian matrix ***A***.
- ***B*** represents the positive definite real symmetric or complex Hermitian matrix ***B***.

Eigenvalues and eigenvectors can be selected by specifying a range of values or a range of indices for the required eigenvalues.

Table 212: Data types.

A*, *B*, *Z*, *work	***vl*, *vu*, *abstol*, *w*, *rwork***	**Subroutine**
Short-precision real	Short-precision real	>|SSYGV$^{\Delta}$|< SSPGVX$^{\Delta}$ SSYGVX$^{\Delta}$
Long-precision real	Long-precision real	>|DSYGV$^{\Delta}$|< DSPGVX$^{\Delta}$ DSYGVX$^{\Delta}$
Short-precision complex	Short-precision real	>|CHEGV$^{\Delta}$|< CHPGVX$^{\Delta}$ CHEGVX$^{\Delta}$
Long-precision complex	Long-precision real	>|ZHEGV$^{\Delta}$|< ZHPGVX$^{\Delta}$ ZHEGVX$^{\Delta}$

$^{\Delta}$LAPACK

Syntax

Table 213: Syntax for supported computing languages.

Language	Syntax
Fortran	>|CALL SSYGV | DSYGV (*itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, info*)|< >|CALL CHEGV | ZHEGV (*itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, info*)|< CALL SSPGVX | DSPGVX (*itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info*) CALL CHPGVX | ZHPGVX (*itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info*) CALL SSYGVX | DSYGVX (*itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu, abstol, m, w, z, ldz, work, lwork, iwork, ifail, info*) CALL CHEGVX | 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*)
C and C++	>|ssygv | dsygv (*itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, info*);|< >|chegv | zhegv (*itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, info*);|< sspgvx | dspgvx (*itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info*); chpgvx | zhpgvx (*itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info*); ssygvx | dsygvx (*itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu, abstol, m, w, z, ldz, work, lwork, iwork, ifail, info*); chegvx | 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*);

LAPACKE	>|info = LAPACKE_ssygv | LAPACKE_dsygv (*matrix_layout, itype, jobz, uplo, n, a, lda, b, ldb, w*);|<
	 >|info = LAPACKE_chegv | LAPACKE_zhegv (*matrix_layout, itype, jobz, uplo, n, a, lda, b, ldb, w*);|<
	 info = LAPACKE_sspgvx | LAPACKE_dspgvx (*matrix_layout, itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z, ldz, ifail*);
	 info = LAPACKE_chpgvx | LAPACKE_zhpgvx (*matrix_layout, itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z, ldz, ifail*);
	 info = LAPACKE_ssygvx | LAPACKE_dsygvx (*matrix_layout, itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu, abstol, m, w, z, ldz, ifail*);
	 info = LAPACKE_chegvx | LAPACKE_zhegvx (*matrix_layout, itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu, abstol, m, w, z, ldz, ifail*);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. The value of the *matrix_layout* argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

itype

specifies the problem type, where:

If *itype* = 1, the problem is ***Ax*** = λ***Bx***.

If *itype* = 2, the problem is ***ABx*** = λ***x***.

If *itype* = 3, the problem is ***BAx*** = λ***x***.

Specified as: an integer; *itype* = 1, 2, or 3.

jobz

indicates the type of computation to be performed, where:

If *jobz* = 'N', eigenvalues only are computed.

If *jobz* = 'V', eigenvalues and eigenvectors are computed.

Specified as: a single character; *jobz* = 'N' or 'V'.

range

indicates which eigenvalues to compute, where:

If *range* = 'A', all eigenvalues are to be found.

If *range* = 'V', all eigenvalues in the interval [*vl*, *vu*] are to be found.

If *range* = 'I', the *il*-th through *iu*-th eigenvalues are to be found.

Specified as: a single character; *range* = 'A', 'V', or 'I'.

uplo

indicates whether the upper or lower triangular parts of the matrices ***A*** and ***B*** are referenced, where:

If *uplo* = 'U', the upper triangular part is referenced.

If *uplo* = 'L', the lower triangular part is referenced.

Specified as: a single character; *uplo* = 'U' or 'L'.

n

is the order of matrices ***A*** and ***B*** that is used in the computation.

Specified as: an integer; $n \geq 0$.

ap

is the real symmetric or complex Hermitian matrix ***A*** of order *n*. It is stored in an array, referred to as AP, where:

If *uplo* = 'U', it is stored in upper-packed storage mode.

If *uplo* = 'L', it is stored in lower-packed storage mode.

Specified as: one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 212 on page 931.

bp

is the positive definite real symmetric or complex Hermitian matrix ***B*** of order *n*. It is stored in an array, referred to as BP, where:

If *uplo* = 'U', it is stored in upper-packed storage mode.

If *uplo* = 'L', it is stored in lower-packed storage mode.

Specified as: one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 212 on page 931.

a

is the real symmetric or complex Hermitian matrix ***A*** of order *n*.

If *uplo* = 'U', it is stored in upper storage mode.

If *uplo* = 'L', it is stored in lower storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 212 on page 931.

lda

is the leading dimension of the array specified for ***A***.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

b

is the positive definite real symmetric or complex Hermitian matrix ***B*** of order *n*.

If *uplo* = 'U', it is stored in upper storage mode.

If *uplo* = 'L', it is stored in lower storage mode.

Specified as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 212 on page 931.

ldb

is the leading dimension of the array specified for ***B***.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

vl

has the following meaning:

If *range* = 'V', it is the lower bound of the interval to be searched for eigenvalues.

If *range* ≠ 'V', this argument is ignored.

Specified as: a number of the data type indicated in Table 212 on page 931. If *range* = 'V', $vl < vu$.

vu

has the following meaning:

If *range* = 'V', it is the upper bound of the interval to be searched for eigenvalues.

If *range* ≠ 'V', this argument is ignored.

Specified as: a number of the data type indicated in Table 212 on page 931. If *range* = 'V', $vl < vu$.

il

has the following meaning:

If *range* = 'I', it is the index (from smallest to largest) of the smallest eigenvalue to be returned.

If *range* ≠ 'I', this argument is ignored.

Specified as: an integer; $il \geq 1$.

iu

has the following meaning:

If *range* = 'I', it is the index (from smallest to largest) of the largest eigenvalue to be returned.

If *range* ≠ 'I', this argument is ignored.

Specified as: an integer; $\min(il, n) \leq iu \leq n$.

abstol

is the absolute tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*, *b*] of width less than or equal to:

- $abstol+\varepsilon(\max(|a|, |b|))$

where ε is the machine precision. If *abstol* ≤ zero, then ε(norm(***T***)) is used in its place, where norm(***T***) is the 1-norm of the tridiagonal matrix obtained by reducing the standard form of the generalized problem to tridiagonal form. For most problems, this is the appropriate level of accuracy to request.

For certain strongly graded matrices, greater accuracy can be obtained in very small eigenvalues by setting *abstol* to a very small positive number. However, if *abstol* is less than:

$$\sqrt{unfl}$$

where *unfl* is the underflow threshold, then:

$$\sqrt{unfl}$$

is used in its place.

Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold—that is, (2)(*unfl*).

If *jobz* = 'V', then setting *abstol* to *unfl*, the underflow threshold, yields the most orthogonal eigenvectors.

Note:

1. The approximate values of the constants used for *abstol* are listed as follows:

   For SSPGVX, CHPGVX, SSYGVX, and CHEGVX:

   ε
   : 0.119209289550781250E-06

   unfl
   : 0.1175494351E-37

   $\sqrt{unfl}$
   : 0.1084202172E-18

   For DSPGVX, ZHPGVX, DSYGVX, and ZHEGVX:

   ε
   : 0.222044604925031308E-15

   unfl
   : 0.222507385850720138E-307

   $\sqrt{unfl}$
   : 0.149166814624004135E-153

2. The value of *abstol* can affect which algorithm is used to compute the eigenvalues and eigenvectors. See Function.

Specified as: a number of the data type indicated in Table 212 on page 931.

m
: See "On Return" on page 937.

w
: See "On Return" on page 937.

z
: See "On Return" on page 937.

ldz
: is the leading dimension of the array specified for ***Z***.

Specified as: an integer; $ldz > 0$ and $ldz \geq n$.

work
is a work area used by these subroutines, where:

For SSPGVX and DSPGVX:
Its size is $8n$.

For CHPGVX and ZHPGVX:
Its size is $2n$.

>|**For SSYGV, DSYGV, CHEGV, ZHEGV, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX:**|<

If *lwork* = 0, *work* is ignored.

If *lwork* ≠ 0, the size of *work* is determined as follows:

- If *lwork* ≠ -1, *work* is (at least) of length *lwork*.
- If *lwork* = -1, *work* is (at least) of length 1.

Specified as: an area of storage containing numbers of the data type indicated in Table 212 on page 931.

lwork
is the number of elements in array `WORK`.

Specified as: an integer; where:

- If *lwork* = 0, the subroutine dynamically allocates the workspace that is needed for use during this computation. The work area is deallocated before control is returned to the calling program.
- If *lwork* = -1, subroutine performs a workspace query and returns the optimal required size of *work* in $work_1$. No computation is performed and the subroutine returns after error checking is complete.
- Otherwise:

  For SSYGV and DSYGV:

  $lwork \geq \max(1, 3n - 1)$.

  For CHEGV and ZHEGV:

  $lwork \geq \max(1, 2n - 1)$.

  For SSYGVX and DSYGVX:
  $lwork \geq \max(1, 8n)$.

  For CHEGVX and ZHEGVX:
  $lwork \geq \max(1, 2n)$.

  Note: >|These formulas represent the minimum required workspace. For best performance, specify either *lwork* = -1 (to obtain the optimal size of workspace) or *lwork* = 0 (to direct the subroutine to dynamically allocate the workspace).|<

rwork
is a work area of size $7n$.

Specified as: an area of storage containing numbers of the data type indicated in Table 212 on page 931.

iwork
is a work area of size $5n$.

Specified as: an area of storage containing integers.

ifail
See "On Return" on page 937.

info
See "On Return" on page 937.

On Return

ap

is overwritten.

Returned as: a one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 212 on page 931.

bp

contains the results of the Cholesky factorization.

For SSPGVX and DSPGVX:

If *uplo* = 'U', if $info \le n$, the triangular factor $\boldsymbol{U}$ from the Cholesky factorization $\boldsymbol{B} = \boldsymbol{U}^T\boldsymbol{U}$ stored in upper-packed storage format.

If *uplo* = 'L', if $info \le n$, the triangular factor $\boldsymbol{L}$ from the Cholesky factorization $\boldsymbol{B} = \boldsymbol{L}\boldsymbol{L}^T$ stored in lower-packed storage mode.

For CHPGVX and ZHPGVX:

If *uplo* = 'U', if $info \le n$, the triangular factor $\boldsymbol{U}$ from the Cholesky factorization $\boldsymbol{B} = \boldsymbol{U}^H\boldsymbol{U}$ stored in upper-packed storage format.

If *uplo* = 'L', if $info \le n$, the triangular factor $\boldsymbol{L}$ from the Cholesky factorization $\boldsymbol{B} = \boldsymbol{L}\boldsymbol{L}^H$ stored in lower-packed storage mode.

Returned as: one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 212 on page 931.

a

is overwritten as follows:

For SSYGV, DSYGV, CHEGV, and ZHEGV:

- If *jobz* = 'V' and if *info* = 0, the matrix $\boldsymbol{A}$ contains the matrix $\boldsymbol{Z}$ of eigenvectors. The eigenvectors are normalized as follows:
  - If *itype* = 1 or 2, $\boldsymbol{Z}^{\boldsymbol{H}}\boldsymbol{BZ} = \boldsymbol{I}$.
  - If *itype* = 3, $\boldsymbol{Z}^{\boldsymbol{H}}\boldsymbol{B}^{\boldsymbol{-1}}\boldsymbol{Z} = \boldsymbol{I}$.
- If *jobz* = 'V' and if $info \ne 0$, the eigenvectors are not valid.
- If *jobz* = 'N' and,
  - If *uplo* = 'U', the leading *n* by *n* upper triangular part of $\boldsymbol{A}$ is overwritten.
  - If *uplo* = 'L', the leading *n* by *n* lower triangular part of $\boldsymbol{A}$ is overwritten.

For SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX:

- If *uplo* = 'U', the leading *n* by *n* upper triangular part of $\boldsymbol{A}$ is overwritten.
- If *uplo* = 'L', the leading *n* by *n* lower triangular part of $\boldsymbol{A}$ is overwritten.

Returned as: an array of dimension *lda* by (at least) *n*, containing numbers of the data type indicated in Table 212 on page 931.

b

contains the results of the Cholesky factorization.

>|**For SSYGV, DSYGV, SSYGVX, and DSYGVX:**|<

If *uplo* = 'U', if $info \le n$, the leading *n* by *n* upper triangular part of $\boldsymbol{B}$ contains the triangular factor $\boldsymbol{U}$ from the Cholesky factorization $\boldsymbol{B} = \boldsymbol{U}^T\boldsymbol{U}$.

If *uplo* = 'L', if $info \le n$, the leading *n* by *n* lower triangular part of $\boldsymbol{B}$ contains the triangular factor $\boldsymbol{L}$ from the Cholesky factorization $\boldsymbol{B} = \boldsymbol{L}\boldsymbol{L}^T$.

>|**For CHEGV, ZHEGV, CHEGVX, and ZHEGVX:**|<

If *uplo* = 'U', if *info* ≤ *n*, the leading *n* by *n* upper triangular part of ***B*** contains the triangular factor ***U*** from the Cholesky factorization $\boldsymbol{B} = \boldsymbol{U}^H\boldsymbol{U}$.

If *uplo* = 'L', if *info* ≤ *n*, the leading *n* by *n* lower triangular part of ***B*** contains the triangular factor ***L*** from the Cholesky factorization $\boldsymbol{B} = \boldsymbol{L}\boldsymbol{L}^H$.

Returned as: an array of dimension *ldb* by (at least) *n*, containing numbers of the data type indicated in Table 212 on page 931.

m

is the number of eigenvalues found.

Returned as: an integer; $0 \leq m \leq n$.

w

is the vector ***w***, containing the computed eigenvalues in ascending order in the first *m* elements of *w*.

Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 212 on page 931.

z

has the following meaning, where:

If *jobz* = 'N', then *z* is ignored.

If *jobz* = 'V' and *info* = 0, the first *m* columns of ***Z*** contain the eigenvectors 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:

For SSYGVX, DSYGVX, SSPGVX, and DSPGVX:

If *itype* = 1 or 2, $\boldsymbol{Z}^T\boldsymbol{B}\boldsymbol{Z} = \boldsymbol{I}$.

If *itype* = 3, $\boldsymbol{Z}^T\boldsymbol{B}^{-1}\boldsymbol{Z} = \boldsymbol{I}$.

For CHPGVX, ZHPGVX, CHEGVX, and ZHEGVX:

If *itype* = 1 or 2, $\boldsymbol{Z}^H\boldsymbol{B}\boldsymbol{Z} = \boldsymbol{I}$.

If *itype* = 3, $\boldsymbol{Z}^H\boldsymbol{B}^{-1}\boldsymbol{Z} = \boldsymbol{I}$.

where ***I*** is the identity matrix.

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.

Returned as: an *ldz* by (at least) max(1, *m*) array, containing numbers of the data type indicated in Table 212 on page 931.

work

is a work area used by this subroutine if *lwork* ≠ 0, where:

If *lwork* ≠ 0 and *lwork* ≠ -1, its size is (at least) of length *lwork*.

If *lwork* = -1, its size is (at least) of length 1.

Returned as: an area of storage, where:

If *lwork* ≥ 1 or *lwork* = -1, then $work_1$ is set to the optimal *lwork* value and all other elements of *work* are overwritten.

rwork

is a work area used by these subroutines for computation.

Returned as: an area of storage of dimension, max (1, 3*n*-2).

ifail

has the following meaning:

If *jobz* = 'N', *ifail* is ignored.

If *jobz* = 'V':

- If *info* = 0, the first *m* elements of *ifail* are zero.
- If *info* > 0, *ifail* contains the indices of the eigenvectors that failed to converge.

Returned as: an array of length *n*, containing integers.

info

has the following meaning:

If *info* = 0, the subroutine completed successfully.

>|If *info* > 0,

- if *info* = *n* + i, for 1 ≤ i ≤ *n*, the leading minor of order i of matrix ***B*** is not positive definite. The factorization of matrix ***B*** cannot be completed and eigenvalues or eigenvectors are not computed.
- For SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, ZHEGVX, LAPACKE_sspgvx, LAPACKE_dspgvx, LAPACKE_chpgvx, LAPACKE_zhpgvx, LAPACKE_ssygvx, LAPACKE_dsygvx, LAPACKE_chegvx, and LAPACKE_zhegvx:

  If *info* = *i*, *i* eigenvectors failed to converge. The indices of the eigenvectors are saved in array *ifail*.
- For SSYGV, DSYGV, CHEGV, ZHEGV, LAPACKE_ssygv, LAPACKE_dsygv, LAPACKE_chegv, LAPACKE_zhegv:

  If *info* = *i*, the algorithm failed to converge. The i off-diagonal elements of an intermediate tridiagonal form did not converge to zero.

|<

>|Returned as:

- For SSYGV, DSYGV, CHEGV, ZHEGV, SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX: an integer; *info* ≥ 0.
- For LAPACKE_ssygv, LAPACKE_dsygv, LAPACKE_chegv, LAPACKE_zhegv, LAPACKE_sspgvx, LAPACKE_dspgvx, LAPACKE_chpgvx, LAPACKE_zhpgvx, LAPACKE_ssygvx, LAPACKE_dsygvx, LAPACKE_chegvx, and LAPACKE_zhegvx: an integer function value; *info* ≥ 0.

|<

Notes:

1. These subroutines accept lowercase letters for the *jobz*, *range*, and *uplo* arguments.
2. In your C program, arguments *m* and *info* must be passed by reference.
3. *ap*, *bp*, *a*, *b*, *w*, *z*, *work*, *rwork*, *iwork* and *ifail* must have no common elements; otherwise, results are unpredictable.
4. For a description of how real symmetric matrices are stored in lower- or upper-packed storage mode, see “Lower-Packed Storage Mode” on page 80 or “Upper-Packed Storage Mode” on page 82, respectively.

   For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see “Complex Hermitian Matrix” on page 84.
5. For a description of how real symmetric matrices are stored in lower or upper storage mode, see “Lower Storage Mode” on page 83 or “Upper Storage Mode” on page 83, respectively.

   For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see “Complex Hermitian Matrix” on page 84.

6. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrices ***A*** and ***B*** are assumed to be zero, so you do not have to set these values. On output, for matrix ***B*** they are set to zero.
7. >|For best performance, specify either *lwork* = -1 (to obtain the optimal required size of workspace) or *lwork* = 0 (to direct the subroutine to dynamically allocate the workspace).|<

Function

These subroutines compute eigenvalues and, optionally, the eigenvectors of a positive definite real symmetric or complex Hermitian generalized eigenproblem:

- If *itype* = 1, the problem is ***Ax*** = λ***Bx***
- If *itype* = 2, the problem is ***ABx*** = λ***x***
- If *itype* = 3, the problem is ***BAx*** = λ***x***

In the formulas:

- ***A*** represents the real symmetric or complex Hermitian matrix ***A***
- ***B*** represents the positive definite real symmetric or complex Hermitian matrix ***B***

Eigenvalues and eigenvectors can be selected by specifying a range of values or a range of indices for the required eigenvalues.

The computation involves the following steps:

1. Compute the Cholesky factorization of ***B***.
2. Reduce the positive definite real symmetric or complex Hermitian generalized eigenproblem to standard form.
3. Compute the requested eigenvalues and, optionally, the eigenvectors of the standard form.
4. Backtransform the eigenvectors to obtain the eigenvectors of the original problem.

If *n* = 0, no computation is performed and the subroutine returns after doing some parameter checking.

See reference [“9” on page 1253].

Error conditions

Resource Errors

1. *lwork* = 0 and unable to allocate work area.

Computational Errors

1. The matrix ***B*** is not positive definite. See output argument *info* for more details.
2. Bisection failed to converge for some eigenvalues. The eigenvalues may not be as accurate as the absolute and relative tolerances.
3. The number of eigenvalues that are computed does not match the number of eigenvalues requested.
4. No eigenvalues were computed because the Gershgorin interval initially used was incorrect.
5. Some eigenvectors failed to converge. The indices are stored in ***ifail***.
6. >|If *info* = i, the algorithm failed to converge. The value of i indicates the number of elements of an intermediate tridiagonal form that did not converge to zero.|<

Informational Errors

1. ESSL computed the eigenvalues using multiple algorithms. Performance may be degraded.

Input-Argument Errors

1. *itype* < 1 or *itype* > 3
2. *jobz* ≠ 'N' or 'V'
3. *range* ≠ 'A', 'V', or 'I'
4. *uplo* ≠ 'U' or 'L'
5. *n* < 0
6. *lda* ≤ 0
7. *lda* < *n*
8. *ldb* ≤ 0
9. *ldb* < *n*
10. *range* = 'V', *n* > 0, and *vu* ≤ *vl*
11. *range* = 'I' and (*il* < 1 or *il* > max(1, *n*))
12. *range* = 'I' and (*iu* < min(*n*, *il*) or *iu* > *n*)
13. *ldz* ≤ 0
14. *jobz* = 'V' and *ldz* < *n*
15. *lwork* ≠ 0 and *lwork* ≠ -1 and *lwork* < the minimum required value

Examples

Example 1

This example shows how to find the eigenvalues of a real symmetric positive generalized eigenproblem of the form: ***Ax*** = λ***Bx***. Matrices ***A*** and ***B*** are stored in upper storage mode.

Matrix ***A*** is:

```
                 |  10    1    2   |
                 |   .    2   -1   |
                 |   .    .    2   |
                 └
```

Matrix ***B*** is:

```
                 |   1    0    0   |
                 |   .    1    0   |
                 |   .    .    1   |
                 └
```

Call Statement and Input:

```
            ITYPE JOBZ UPLO   N  A   LDA B LDB  W  WORK  LWORK INFO
              |     |    |    |  |    |  |  |   |   |      |     |
CALL DSYGV  ( 1,   'N', 'U', 3, A,  3, B, 3,  W, WORK,  0,   INFO)
```

Output:

Matrix ***A*** is overwritten.

```
          W =    |    0.5153   |
                 |    2.9466   |
                 |   10.5381   |
                 └
```

```
                 |   1    0    0   |
          B =    |   .    1    0   |
                |   .    .    1   |
                └
```

```
INFO = 0
```

Example 2

This example shows how to find the eigenvalues and eigenvectors of a real symmetric positive generalized eigenproblem of the form: ***ABx*** = λ***x***. Matrices ***A*** and ***B*** are stored in upper storage mode.

Matrix ***A*** is:

```
                |   6    4    4   |
                |   .    6    1   |
                |   .    .    6   |
                └
```

Matrix ***B*** is:

```
                |   1    0    0   |
                |   .    1    0   |
                |   .    .    1   |
                └
```

Call Statement and Input:

```
            ITYPE JOBZ UPLO N  A LDA  B LDB W  WORK LWORK INFO
              |     |    |   |  |  |   |  |  |   |    |     |
CALL DSYGV  ( 2,  'V', 'U', 3, A, 3,  B, 3, W, WORK, 0,   INFO)
```

Output:

```
         W =    |   0.8210916  |
                |   5.000000   |
                |  12.178908   |
                └
```

```
         A =    |   -0.737578    0.4774816    0.4774816   |
                |    0.0000000  -0.707106     0.7071067   |
                |   -0.675261   -0.521546    -0.521546    |
                └
```

```
         B =    |   1    0    0   |
                |   .    1    0   |
                |   .    .    1   |
                └
```

```
INFO = 0
```

Example 3

This example shows how to find the eigenvalues of a real symmetric positive generalized eigenproblem of the form: ***BAx*** = λ***x***. Matrices ***A*** and ***B*** are stored in upper storage mode.

Matrix ***A*** is:

```
                |  (10.0, 1.0)  (1.0, 0 )    (2.0, -1)   |
                |  ( .       )  (2.0, -1)    (-1.0, 0)   |
                 |  ( .        )  ( .      )    (2.0, 1.5)  |
                 └
```

Matrix ***B*** is:

```
                 ┌                                            ┐
                 |  (1.0, 0.0)   (0.0, 0.0)   (0.0, 0.0)   |
                 |     ( . )     (1.0, 0.0)   (0.0, 0.0)   |
                 |     ( . )       ( . )      (1.0, 0.0)   |
                 └                                            ┘
```

Call Statement and Input:

```
            ITYPE JOBZ UPLO  N  A LDA  B LDB  W  WORK LWORK RWORK INFO
              |    |    |   |  |  |   |  |   |   |    |     |     |
CALL ZHEGV  ( 3,   'N', 'U', 3, A, 3,  B, 3,  W, WORK, 0,  RWORK, INFO)
```

Output:

Matrix ***A*** is overwritten.

```
                 ┌              ┐
          W =    |   0.4537610   |
                 |   2.8973095   |
                 |  10.648929    |
                 └              ┘
```

```
                 ┌                                            ┐
                 |  (1.0, 0.0)   (0.0, 0.0)   (0.0, 0.0)   |
          B =    |     ( . )     (1.0, 0.0)   (0.0, 0.0)   |
                 |     ( . )       ( . )      (1.0, 0.0)   |
                 └                                            ┘
```

```
INFO = 0
```

Example 4

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian generalized eigenproblem of the form: ***ABx*** = λ***x***. Matrices ***A*** and ***B*** are stored in lower storage mode.

Matrix ***A*** is:

```
                 ┌                                            ┐
                 |  (6.0, 0.0)   (  .      )   (  .      )   |
                 |  (4.0, 0.0)   (6.0, 0.0)   (  .      )   |
                 |  (4.0, 0.0)   (1.0, 0.0)   (6.0, 0.0)   |
                 └                                            ┘
```

Matrix ***B*** is:

```
                 ┌                                          ┐
                 |  (1.0, 0.0)  (  .      )  (  .      )   |
                 |  (0.0, 0.0)  (1.0, 0.0)  (  .      )   |
                 |  (0.0, 0.0)  (0.0, 0.0)  (1.0, 0.0)   |
                 └                                          ┘
```

Call Statement and Input:

```
             ITYPE JOBZ UPLO N  A LDA B LDB W  WORK LWORK RWORK INFO
              |    |    |   |  |  |  |  |  |   |     |     |     |
CALL ZHEGV  ( 2,   'V', 'L', 3, A, 3, B, 3, W, WORK,  0,  RWORK, INFO)
```

Output:

```
        W =    |   0.8210916   |
               |   5.0000000   |
               |  12.178908    |
               └
```

```
        A =    |  (-0.737578, 0.0) (0.47748165,0.0) (0.4774816,0.0)   |
               |        (0.0, 0.0) (-0.7071067,0.0) (0.7071067,0.0)   |
               |  (-0.675261, 0.0) (-0.5215469,0.0) (-0.521546,0.0)   |
               └
```

```
        B =    |  (1.0, 0.0)       ( . )        ( . )       |
               |  (0.0, 0.0)     (1.0, 0.0)   (0.0, 0.0)   |
               |  (0.0, 0.0)     (0.0, 0.0)   (1.0, 0.0)   |
               └
```

```
INFO = 0
```

Example 5

This example shows how to find the eigenvalues of a real symmetric positive generalized eigenproblem of the form: ***Ax*** = λ***Bx***. Matrices ***A*** and ***B*** are stored in lower-packed storage mode.

Matrix ***A*** is:

```
               |  6.0   4.0   4.0   1.0 |
               |  4.0   6.0   1.0   4.0 |
               |  4.0   1.0   6.0   4.0 |
               |  1.0   4.0   4.0   6.0 |
```

Matrix ***B*** is:

```
               |  1.0   0.0   0.0   0.0 |
               |  0.0   1.0   0.0   0.0 |
               |  0.0   0.0   1.0   0.0 |
               |  0.0   0.0   0.0   1.0 |
```

Notes:

1. Because *jobz* = 'N', ***Z*** and *ifail* are not referenced.
2. Because *range* = 'A', arguments *vl*, *vu*, *il*, and *iu* are not referenced.

Call Statement and Input:

```
            ITYPE JOBZ RANGE UPLO   N   AP   BP VL   VU   IL   IU ABSTOL M   W   Z   LDZ WORK   IWORK   IFAIL
INFO
              |     |    |     |    |  |    |  |    |    |    |      |   |  |  |  |    |       |       |
|
CALL DSPGVX ( 1,   'N', 'A',  'L', 4, AP, BP VL, VU, IL, IU, -1.0, M, W, Z, 4,  WORK, IWORK, IFAIL,
INFO)

     AP       =  (6.0, 4.0, 4.0, 1.0, 6.0, 1.0, 4.0, 6.0, 4.0, 6.0)

     BP       =  (1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0)
```

Output:

Matrix AP is overwritten.

```
BP   =  (1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0)
```

```
M    =    4
```

```
W    =  (-1.0, 5.0, 5.0, 15.0)
```

```
INFO = 0
```

Example 6

This example shows how to find the eigenvalues and eigenvectors of a real symmetric positive generalized eigenproblem of the form: ***ABx*** = λ***x***. Matrices ***A*** and ***B*** are stored in upper-packed storage mode.

This example illustrates the use of the *il* and *iu* parameters when range = 'I'.

Matrices ***A*** and ***B*** are the same as in Example 5.

Note: Because *range* = 'I', arguments *vl* and *vu* are not referenced.

Call Statement and Input:

```
            ITYPE JOBZ RANGE UPLO   N   AP   BP VL   VU   IL   IU ABSTOL M   W   Z   LDZ WORK   IWORK   IFAIL
INFO
              |     |    |     |    |  |    |  |    |    |    |      |   |  |  |   |    |      |       |
|
CALL DSPGVX ( 2,   'V', 'I',  'U', 4, AP, BP VL, VU, 1,  2,  -1.0, M, W, Z, 4,  WORK, IWORK, IFAIL,
INFO)

     AP       =  (6.0, 4.0, 6.0, 4.0, 1.0, 6.0, 1.0, 4.0, 4.0, 6.0)

     BP       =  (1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0)
```

Output:

Matrix AP is overwritten.

```
BP    =  (1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0)
```

```
M    =    2
```

```
W    =  (-1.0, 5.0, . , . )
```

```
        | 0.500000  0.544042      .         .   |
Z    =  |-0.500000  0.451683      .         .   |
        |-0.500000 -0.451683      .         .   |
        | 0.500000 -0.544042      .         .   |
```

```
IFAIL = (0, 0, . , . )
```

```
INFO = 0
```

Example 7

This example shows how to find all eigenvalues only of a positive definite complex Hermitian generalized eigenproblem of the form: ***Ax*** = λ***Bx***. Matrices ***A*** and ***B*** are stored in lower-packed storage mode.

Matrix ***A*** is:

```
                 |  (6.0, 0.0)   (4.0, 0.0)   (4.0, 0.0)   (1.0, 0.0) |
                 |  (4.0, 0.0)   (6.0, 0.0)   (1.0, 0.0)   (4.0, 0.0) |
                  |  (4.0, 0.0)   (1.0, 0.0)   (6.0, 0.0)   (4.0, 0.0) |
                  |  (1.0, 0.0)   (4.0, 0.0)   (4.0, 0.0)   (6.0, 0.0) |
```

Matrix ***B*** is:

```
                  |  (1.0, 0.0)   (0.0, 0.0)   (0.0, 0.0)   (0.0, 0.0) |
                  |  (0.0, 0.0)   (1.0, 0.0)   (0.0, 0.0)   (0.0, 0.0) |
                  |  (0.0, 0.0)   (0.0, 0.0)   (1.0, 0.0)   (0.0, 0.0) |
                  |  (0.0, 0.0)   (0.0, 0.0)   (0.0, 0.0)   (1.0, 0.0) |
```

Notes:

1. Because *jobz*= 'N', ***Z*** and *ifail* are not referenced.
2. Because *range* = 'A', arguments *vl*, *vu*, *il*, and *iu* are not referenced.
3. On input, the imaginary parts of the Hermitian matrix ***A*** are assumed to be zero, values. On output, they are set to zero.

Call Statement and Input:

```
            ITYPE JOBZ RANGE UPLO  N  AP  BP VL  VU  IL  IU ABSTOL M  W  Z  LDZ WORK RWORK  IWORK  IFAIL  INFO
              |     |    |     |   |  |   |  |   |   |   |     |   |  |  |  |    |     |      |      |     |
CALL ZHPGVX ( 1,   'N', 'A',  'L', 4, AP, BP VL, VU, IL, IU, -1.0, M, W, Z, 4, WORK, RWORK, IWORK, IFAIL, INFO)

AP      =  ((6.0, . ), (4.0, 0.0), (4.0, 0.0), (1.0, 0.0),(6.0, . ), (1.0, 0.0), (4.0, 0.0), (6.0, . ),
            (4.0, 0.0), (6.0, . ))

BP      =  ((1.0, . ), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0),(1.0, . ), (0.0, 0.0), (0.0, 0.0), (1.0, . ),
            (0.0, 0.0), (1.0, . ))
```

Output:

Matrix AP is overwritten.

```
M    =    4
```

```
BP    =  ((1.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0),(1.0, 0.0), (0.0, 0.0), (0.0, 
0.0),
          (1.0, 0.0), (0.0, 0.0), (1.0, 0.0))
```

```
W    =  (-1.0, 5.0, 5.0, 15.0)
```

```
INFO = 0
```

Example 8

This example shows how to find all eigenvalues and eigenvectors of a positive definite complex Hermitian generalized eigenproblem of the form: ***ABx*** = λ***x***. Matrices ***A*** and ***B*** are stored in upper-packed storage mode.

This example illustrates the use of the *il* and *iu* parameters when range = 'I'.

Matrices ***A*** and ***B*** are the same as in Example 7.

Notes:

1. Because *range* = 'I', arguments *vl* and *vu* are not referenced.
2. On input, the imaginary parts of the Hermitian matrix ***A*** are assumed to be zero, values. On output, they are set to zero.

Call Statement and Input:

```
           ITYPE JOBZ RANGE UPLO  N  AP  BP VL  VU  IL  IU ABSTOL M  W  Z  LDZ WORK RWORK  IWORK  IFAIL  INFO
             |     |    |     |   |  |   |  |   |   |   |     |   |  |  |  |    |     |      |      |     |
CALL ZHPGVX ( 2,   'V', 'I',  'L', 4, AP, BP VL, VU, 1,  2,  -1.0, M, W, Z, 4, WORK, RWORK, IWORK, IFAIL, INFO)

AP       =  ((6.0, . ), (4.0, 0.0), (4.0, 0.0), (1.0, 0.0),(6.0, . ), (1.0, 0.0), (4.0, 0.0), (6.0, . ),
             (4.0, 0.0), (6.0, . ))

BP       =  ((1.0, . ), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0),(1.0, . ), (0.0, 0.0), (0.0, 0.0), (1.0, . ),
             (0.0, 0.0), (1.0, . ))
```

Output:

Matrix AP is overwritten.

```
BP    =  ((1.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0),(1.0, 0.0), (0.0, 0.0), (0.0, 
0.0),
          (1.0, 0.0), (0.0, 0.0), (1.0, 0.0))
```

```
M    =    2
```

```
W    =  (-1.0, 5.0, . , . )
```

```
        |( 0.500000, 0.0) ( 0.544042, 0.0)      .         .   |
Z    =  |(-0.500000, 0.0) ( 0.451683, 0.0)      .         .   |
        |(-0.500000, 0.0) (-0.451683, 0.0)      .         .   |
        |( 0.500000, 0.0) (-0.544042, 0.0)      .         .   |
```

```
IFAIL = (0, 0, . , . )

INFO = 0
```

Example 9

This example shows how to find the eigenvalues only of a positive definite real symmetric generalized eigenproblem of the form: ***Ax*** = λ***Bx***. Matrices ***A*** and ***B*** are stored in lower storage mode.

Matrices ***A*** and ***B*** are the same as in Example 5

Notes:

1. Because *jobz* = 'N', ***Z*** and *ifail* are not referenced.
2. Because *range* = 'A', arguments *vl*, *vu*, *il*, and *iu* are not referenced.
3. Because *lwork* = 0, the subroutine dynamically allocates WORK.

Call Statement and Input:

```
           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 ( 1,   'N', 'A',  'L', 4, A, 4, B, 4, VL, VU, IL, IU, -1.0, M, W, Z,  4, WORK,  0,  IWORK, IFAIL, INFO)
```

```
        |   6.0   .    .    .   |
A    =  |   4.0  6.0   .    .   |
        |   4.0  1.0  6.0   .   |
        |   1.0  4.0  4.0  6.0  |
```

```
        |   1.0   .    .    .   |
B    =  |   0.0  1.0   .    .   |
        |   0.0  0.0  1.0   .   |
        |   0.0  0.0  0.0  1.0  |
```

Output:

Matrix ***A*** is overwritten.

```
M    =    4
```

```
        |   1.000000   .         .         .        |
B    =  |   0.000000  1.000000   .         .        |
        |   0.000000  0.000000  1.000000   .        |
        |   0.000000  0.000000  0.000000  1.000000  |
```

```
        | - 1.000000  |
W    =  |   5.000000  |
        |   5.000000  |
        |  15.000000  |
```

```
INFO = 0
```

Example 10

This example shows how to find the eigenvalues and eigenvectors of a positive definite real symmetric generalized eigenproblem of the form: ***ABx*** = λ***x***. Matrices ***A*** and ***B*** are stored in upper storage mode.

This example illustrates the use of the *il* and *iu* parameters when *range* = 'I'.

Matrices ***A*** and ***B*** are the same as in Example 5.

Notes:

1. Because *range* = 'I', arguments *vl* and *vu* are not referenced.
2. Because *lwork* = 0, the subroutine dynamically allocates WORK.

Call Statement and Input:

```
            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 ( 2,   'V',  'I',  'U', 4, A, 4, B, 4, VL, VU, 1, 2, -1.0, M, W, Z,  4, WORK,  0,  IWORK, IFAIL, INFO)
```

Output:

Matrix ***A*** is overwritten.

```
M    =    2
```

```
        |   1.000000  0.000000  0.000000  0.000000  |
B    =  |    .        1.000000  0.000000  0.000000  |
        |    .         .        1.000000  0.000000  |
        |    .         .         .        1.000000  |
```

```
W    =  |  -1.000000  |
        |   5.000000  |
        |    .        |
        |    .        |
```

```
        |   0.500000  0.543058   .         .        |
Z    =  |  -0.500000  0.452866   .         .        |
        |  -0.500000 -0.452866   .         .        |
        |   0.500000 -0.543058   .         .        |
```

```
IFAIL = (0, 0, ., .)
```

```
INFO = 0
```

Example 11

This example shows how to find the eigenvalues and eigenvectors of a positive definite real symmetric generalized eigenproblem of the form: ***BAx*** = λ***x***. Matrices ***A*** and ***B*** are stored in upper storage mode.

This example illustrates the use of the *vl* and *vu* parameters when *range* = 'V'.

Matrices ***A*** and ***B*** are the same as in Example 5.

Notes:

1. Because *range* = 'V', arguments *il* and *iu* are not referenced.
2. On output, array A is overwritten.
3. Because *lwork* = 0, the subroutine dynamically allocates `WORK`.

Call Statement and Input:

```
            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 ( 3,   'V',  'V',  'U', 4, A, 4, B, 4, 2.0, 6.0, IL, IU, -1.0, M, W, Z,  4, WORK,  0,  IWORK, IFAIL, INFO)
```

```
        |   6.0  4.0  4.0  1.0  |
A    =  |    .   6.0  1.0  4.0  |
        |    .    .   6.0  4.0  |
        |    .    .    .   6.0  |
        └
```

```
        |   1.0  0.0  0.0  0.0  |
B    =  |    .   1.0  0.0  0.0  |
        |    .    .   1.0  0.0  |
        |    .    .    .   1.0  |
        └
```

Output:

Matrix ***A*** is overwritten.

```
M    =    2
```

```
        |   1.000000  0.000000  0.000000  0.000000  |
B    =  |    .        1.000000  0.000000  0.000000  |
        |    .         .        1.000000  0.000000  |
        |    .         .         .        1.000000  |
```

```
W    =  |   5.000000  |
        |   5.000000  |
        |    .        |
        |    .        |
```

```
        |  -0.123202 -0.696291   .         .        |
Z    =  |   0.696291 -0.123202   .         .        |
        |  -0.696291  0.123202   .         .        |
        |   0.123202  0.696291   .         .        |
```

```
IFAIL = (0, 0, ., .)

INFO = 0
```

Example 12

This example shows how to find the eigenvalues only of a positive definite complex Hermitian generalized eigenproblem of the form: ***Ax*** = λ***Bx***. Matrices ***A*** and ***B*** are stored in lower-packed storage mode.

Matrices ***A*** and ***B*** are the same as in Example 3.

Notes:

1. Because *jobz* = 'N', ***Z*** and *ifail* are not referenced.
2. Because *range* = 'A', arguments *vl*, *vu*, *il*, and *iu* are not referenced.
3. Because *lwork* = 0, the subroutine dynamically allocates WORK.
4. On input, the imaginary parts of the Hermitian matrix ***A*** are assumed to be zero, values. On output, they are set to zero.

Call Statement and Input:

```
            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 ( 1,   'N', 'A',  'L', 4, A, 4, B, 4, VL, VU, IL, IU, -1.0, M, W, Z,  4, WORK,  0,  RWORK, IWORK, IFAIL, INFO)
```

```
        |  (6.0,  . )      .          .          .      |
A    =  |  (4.0, 0.0)   (6.0,  . )     .          .      |
        |  (4.0, 0.0)   (1.0, 0.0)   (6.0,  . )     .      |
        |  (1.0, 0.0)   (4.0, 0.0)   (4.0, 0.0)   (6.0, . )  |
        └
```

```
        |  (1.0,  . )      .          .          .      |
B    =  |  (0.0, 0.0)   (1.0,  . )     .          .      |
        |  (0.0, 0.0)   (0.0, 0.0)   (1.0,  . )     .      |
        |  (0.0, 0.0)   (0.0, 0.0)   (0.0, 0.0)   (1.0,  . ) |
        └
```

Output:

Matrix ***A*** is overwritten.

```
M    =    4
```

```
         |  (1.0, 0.0)       .            .            .       |
B    =   |  (0.0, 0.0)   (1.0, 0.0)       .            .       |
         |  (0.0, 0.0)   (0.0, 0.0)   (1.0, 0.0)       .       |
         |  (0.0, 0.0)   (0.0, 0.0)   (0.0, 0.0)   (1.0, 0.0) |
```

```
         |  -1.000000  |
W    =   |   5.000000  |
         |   5.000000  |
         |  15.000000  |
```

```
INFO = 0
```

Example 13

This example shows how to find the eigenvalues and eigenvectors of a positive definite complex Hermitian generalized eigenproblem of the form: ***ABx*** = λ***x***. Matrices ***A*** and ***B*** are stored in upper-packed storage mode.

This example illustrates the use of the *il* and *iu* parameters when range = 'I'.

Matrices ***A*** and ***B*** are the same as in Example 3.

Notes:

1. Because *range* = 'I', arguments *vl* and *vu* are not referenced.
2. Because *lwork* = 0, the subroutine dynamically allocates WORK.
3. On input, the imaginary parts of the Hermitian matrix ***A*** are assumed to be zero, values. On output, they are set to zero.

Call Statement and Input:

```
             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 ( 2,   'V', 'I',  'L', 4, A, 4, B, 4, VL, VU,  1,  2, -1.0, M, W, Z,  4, WORK,  0,  RWORK, IWORK, IFAIL, INFO)
```

```
A    =  |  (6.0,  . )       .           .           .      |
        |  (4.0, 0.0)   (6.0,  . )      .           .      |
        |  (4.0, 0.0)   (1.0, 0.0)   (6.0,  . )     .      |
        |  (1.0, 0.0)   (4.0, 0.0)   (4.0, 0.0)   (6.0, . )  |
        └
```

```
B    =  |  (1.0,  . )       .           .           .      |
        |  (0.0, 0.0)   (1.0,  . )      .           .      |
        |  (0.0, 0.0)   (0.0, 0.0)   (1.0,  . )     .      |
        |  (0.0, 0.0)   (0.0, 0.0)   (0.0, 0.0)   (1.0,  . ) |
        └
```

Output:

Matrix ***A*** is overwritten.

```
M    =    2
```

```
B    =  |  (1.0, 0.0)       .            .            .       |
        |  (0.0, 0.0)   (1.0, 0.0)       .            .       |
        |  (0.0, 0.0)   (0.0, 0.0)   (1.0, 0.0)       .       |
        |  (0.0, 0.0)   (0.0, 0.0)   (0.0, 0.0)   (1.0, 0.0) |
```

```
W    =  |  -1.000000  |
        |   5.000000  |
```

```
Z    =  |( 0.500000, 0.0) (-0.154815, 0.0)      .         .   |
        |(-0.500000, 0.0) (-0.689950, 0.0)      .         .   |
        |(-0.500000, 0.0) ( 0.689950, 0.0)      .         .   |
        |( 0.500000, 0.0) ( 0.154815, 0.0)      .         .   |
```

```
IFAIL = (0, 0, ., .)

INFO = 0
```

Chapter 12. Fourier Transforms, Convolutions and Correlations, and Related Computations

The signal processing subroutines, provided in three areas, are described here.

Overview of the Signal Processing Subroutines

This describes the subroutines in each of the three signal processing areas:

- Fourier transform subroutines
- Convolution and correlation subroutines
- Related-computation subroutines

Fourier Transforms Subroutines

The Fourier transform subroutines perform mixed-radix transforms in one, two, and three dimensions.

Table 214: List of Fourier Transform Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SCFTD^{GPU}	DCFTD^{GPU}	"SCFTD and DCFTD (Multidimensional Complex Fourier Transform)" on page 962
SRCFTD^{GPU}	DRCFTD^{GPU}	"SRCFTD and DRCFTD (Multidimensional Real-to-Complex Fourier Transform)" on page 970
SCRFTD^{GPU}	DCRFTD^{GPU}	"SCRFTD and DCRFTD (Multidimensional Complex-to-Real Fourier Transform)" on page 978
SCFT § SCFTP$^{§, ND}$	DCFT§	"SCFT and DCFT (Complex Fourier Transform)" on page 986
SRCFT§	DRCFT§	"SRCFT and DRCFT (Real-to-Complex Fourier Transform)" on page 993
SCRFT§	DCRFT§	"SCRFT and DCRFT (Complex-to-Real Fourier Transform)" on page 1000
SCOSF SCOSFT$^{§, ND}$	DCOSF	"SCOSF and DCOSF (Cosine Transform)" on page 1008
SSINF	DSINF	"SSINF and DSINF (Sine Transform)" on page 1015
SCFT2§ SCFT2P$^{§, ND}$	DCFT2§	"SCFT2 and DCFT2 (Complex Fourier Transform in Two Dimensions)" on page 1021
SRCFT2§	DRCFT2§	"SRCFT2 and DRCFT2 (Real-to-Complex Fourier Transform in Two Dimensions)" on page 1028
SCRFT2§	DCRFT2§	"SCRFT2 and DCRFT2 (Complex-to-Real Fourier Transform in Two Dimensions)" on page 1034

Table 214: List of Fourier Transform Subroutines. (continued)

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SCFT3§ SCFT3P§, ND	DCFT3§	"SCFT3 and DCFT3 (Complex Fourier Transform in Three Dimensions)" on page 1041
SRCFT3§	DRCFT3§	"SRCFT3 and DRCFT3 (Real-to-Complex Fourier Transform in Three Dimensions)" on page 1047
SCRFT3§	DCRFT3§	"SCRFT3 and DCRFT3 (Complex-to-Real Fourier Transform in Three Dimensions)" on page 1053

§ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs.

ND Documentation for this subroutine is no longer provided.

GPU This subroutine can offload computations to the GPUs when the ESSL SMP CUDA library is used.

Convolution and Correlation Subroutines

The convolution and correlation subroutines provide the choice of using Fourier methods or direct methods. The Fourier-method subroutines contain a high-performance mixed-radix capability. There are also several direct-method subroutines that provide decimated output.

Table 215: List of Convolution and Correlation Subroutines.

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SCON§ SCOR§		"SCON and SCOR (Convolution or Correlation of One Sequence with One or More Sequences)" on page 1060
SCOND SCORD		"SCOND and SCORD (Convolution or Correlation of One Sequence with Another Sequence Using a Direct Method)" on page 1065
SCONF SCORF		"SCONF and SCORF (Convolution or Correlation of One Sequence with One or More Sequences Using the Mixed-Radix Fourier Method)" on page 1070
SDCON SDCOR	DDCON DDCOR	"SDCON, DDCON, SDCOR, and DDCOR (Convolution or Correlation with Decimated Output Using a Direct Method)" on page 1079
SACOR§		"SACOR (Autocorrelation of One or More Sequences)" on page 1083
SACORF		"SACORF (Autocorrelation of One or More Sequences Using the Mixed-Radix Fourier Method)" on page 1086

§ These subroutines are provided only for migration from earlier releases of ESSL and are not intended for use in new programs.

Related-Computation Subroutines

The related-computation subroutines consist of a group of computations that can be used in general signal processing applications. They are similar to those provided on the IBM 3838 Array Processor; however, the ESSL subroutines generally solve a wider range of problems.

Table 216: List of Related-Computation Subroutines.

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SPOLY	DPOLY	"SPOLY and DPOLY (Polynomial Evaluation)" on page 1091
SIZC	DIZC	"SIZC and DIZC (I-th Zero Crossing)" on page 1094
STREC	DTREC	"STREC and DTREC (Time-Varying Recursive Filter)" on page 1096
SQINT	DQINT	"SQINT and DQINT (Quadratic Interpolation)" on page 1099
SWLEV CWLEV	DWLEV ZWLEV	"SWLEV, DWLEV, CWLEV, and ZWLEV (Wiener-Levinson Filter Coefficients)" on page 1102

Fourier Transforms, Convolutions, and Correlations Considerations

This describes some global information applying to the Fourier transform, convolution, and correlation subroutines.

Use Considerations

This provides some key points about using the Fourier transform, convolution, and correlation subroutines.

Understanding the Terminology and Conventions Used for Your Array Data

These subroutines use the term "sequences," rather than vectors and matrices, to describe the data that is stored in the arrays.

Some of the sequences used in these computations use a zero origin rather than a one-origin. For example, x_j can be expressed with $j = 0, 1, ..., n-1$ rather than $j = 1, 2, ..., n$. When using the formulas provided to calculate array sizes or offsets into arrays, you need to be careful that you substitute the correct values. For example, the number of x_j elements in the sequence is n, not n-1.

Concerns about Lengths of Transforms

The length of the transform you can use in your program depends on the limits of the addressability of your processor.

Determining an Acceptable Length of a Transform

To determine acceptable lengths of the transforms in the Fourier transform subroutines, you have different choices depending on which subroutine you are using:

- For subroutines in Table 217 on page 956, all transform lengths between 0 and 1073479680 are acceptable.
- For subroutines in Table 218 on page 956, you have two choices:
  - You can use the formula or table of values in "Acceptable Lengths for the Transforms" on page 956 to choose a value.
  - Alternatively, ESSL's input-argument error recovery provides a means of determining an acceptable length of the transform. It uses the optionally-recoverable error 2030. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

Table 217: *Fourier Transform subroutines allowing all lengths between 0 and 1073479680.*

Subroutine Name
SCFTD, DCFTD SRCFTD, DRCFTD SCRFTD, DCRFTD

Table 218: *Fourier Transform subroutines whose lengths are limited to those in* Figure 15 on page 957.

Subroutine Name
SCFT, DCFT SCFTP SRCFT, DRCFT SCRFT, DCRFT SCOSF, DCOSF SCOSFT SSINF, DSINF SCFT2, DCFT2 SCFT2P SRCFT2, DRCFT2 SCRFT2, DCRFT2 SCFT3, DCFT3 SCFT3P SRCFT3, DRCFT3 SCRFT3, DCRFT3

Acceptable Lengths for the Transforms

Use the following formula to determine acceptable transform lengths:

$n = (2^h)(3^i)(5^j)(7^k)(11^m) \quad \text{for } n \leq 37748736$

where:

$h = 1, 2, \ldots, 25$
$i = 0, 1, 2$
$j, k, m = 0, 1$

Figure 15 on page 957 lists all the acceptable values for transform lengths in the Fourier transform subroutines.

```
         2         4         6         8        10        12        14        16        18
        20        22        24        28        30        32        36        40        42
        44        48        56        60        64        66        70        72        80
        84        88        90        96       110       112       120       126       128
       132       140       144       154       160       168       176       180       192
       198       210       220       224       240       252       256       264       280
       288       308       320       330       336       352       360       384       396
       420       440       448       462       480       504       512       528       560
       576       616       630       640       660       672       704       720       768
       770       792       840       880       896       924       960       990      1008
      1024      1056      1120      1152      1232      1260      1280      1320      1344
      1386      1408      1440      1536      1540      1584      1680      1760      1792
      1848      1920      1980      2016      2048      2112      2240      2304      2310
      2464      2520      2560      2640      2688      2772      2816      2880      3072
      3080      3168      3360      3520      3584      3696      3840      3960      4032
      4096      4224      4480      4608      4620      4928      5040      5120      5280
      5376      5544      5632      5760      6144      6160      6336      6720      6930
      7040      7168      7392      7680      7920      8064      8192      8448      8960
      9216      9240      9856     10080     10240     10560     10752     11088     11264
     11520     12288     12320     12672     13440     13860     14080     14336     14784
     15360     15840     16128     16384     16896     17920     18432     18480     19712
     20160     20480     21120     21504     22176     22528     23040     24576     24640
     25344     26880     27720     28160     28672     29568     30720     31680     32256
     32768     33792     35840     36864     36960     39424     40320     40960     42240
     43008     44352     45056     46080     49152     49280     50688     53760     55440
     56320     57344     59136     61440     63360     64512     65536     67584     71680
     73728     73920     78848     80640     81920     84480     86016     88704     90112
     92160     98304     98560    101376    107520    110880    112640    114688    118272
    122880    126720    129024    131072    135168    143360    147456    147840    157696
    161280    163840    168960    172032    177408    180224    184320    196608    197120
    202752    215040    221760    225280    229376    236544    245760    253440    258048
    262144    270336    286720    294912    295680    315392    322560    327680    337920
    344064    354816    360448    368640    393216    394240    405504    430080    443520
    450560    458752    473088    491520    506880    516096    524288    540672    573440
    589824    591360    630784    645120    655360    675840    688128    709632    720896
    737280    786432    788480    811008    860160    887040    901120    917504    946176
    983040   1013760   1032192   1048576   1081344   1146880   1179648   1182720   1261568
   1290240   1310720   1351680   1376256   1419264   1441792   1474560   1572864   1576960
   1622016   1720320   1774080   1802240   1835008   1892352   1966080   2027520   2064384
   2097152   2162688   2293760   2359296   2365440   2523136   2580480   2621440   2703360
   2752512   2838528   2883584   2949120   3145728   3153920   3244032   3440640   3548160
   3604480   3670016   3784704   3932160   4055040   4128768   4194304   4325376   4587520
   4718592   4730880   5046272   5160960   5242880   5406720   5505024   5677056   5767168
   5898240   6291456   6307840   6488064   6881280   7096320   7208960   7340032   7569408
   7864320   8110080   8257536   8388608   8650752   9175040   9437184   9461760  10092544
  10321920  10485760  10813440  11010048  11354112  11534336  11796480  12582912  12615680
  12976128  13762560  14192640  14417920  14680064  15138816  15728640  16220160  16515072
  16777216  17301504  18350080  18874368  18923520  20185088  20643840  20971520  21626880
  22020096  22708224  23068672  23592960  25165824  25231360  25952256  27525120  28385280
  28835840  29360128  30277632  31457280  32440320  33030144  33554432  34603008  36700160
  37748736
```

Figure 15: Table of Acceptable Lengths for the Transforms

Understanding Auxiliary Working Storage Requirements

Auxiliary working storage is required by the Fourier transform subroutines and by the SCONF, SCORF, and SACORF subroutines. This storage is provided through the calling sequence arguments *aux*, *aux1*, and *aux2*. The sizes of these storage areas are specified by the calling sequence arguments *naux*, *naux1*, and *naux2*, respectively.

AUX1

The *aux1* array is used for storing tables and other parameters when you call a Fourier transform, convolution, or correlation subroutine for initialization with *init* = 1. The initialized *aux1* array is then used on succeeding calls with *init* = 0, when the computation is actually done. You should not use this array between the initialization and the computation.

AUX and AUX2

The *aux* and *aux2* arrays are used for temporary storage during the running of the subroutine and are available for use by your program between calls to the subroutine.

AUX3

The *aux3* argument is provided for migration purposes only and is ignored.

Initializing Auxiliary Working Storage

In many of those subroutines requiring *aux1* auxiliary working storage, two invocations of the subroutines are necessary. The first invocation initializes the working storage in *aux1* for the subroutine, and the second performs the computations. (For an explanation of auxiliary working storage, see "Understanding Auxiliary Working Storage Requirements" on page 957.) As a result, the working storage in *aux1* should not be used by the calling program between the two calls to the subroutine. However, it can be reused after intervening calls to the subroutine with different arguments.

If you plan to repeat a computation many times using the same set of arguments, you only need to do one initialization of the *aux1* array; that is, the initialized *aux1* array can be saved and reused as many times as needed for the computation.

If you plan to perform different computations, with different sets of arguments (except for input argument *x*), you need to do an initialization for each different computation; that is, you initialize the various *aux1* arrays for use with the different computations, saving and reusing them until they are not needed any more.

Determining the Amount of Auxiliary Working Storage That You Need

To determine the size of auxiliary storage, you have several choices. First, you can use the formulas provided in each subroutine description. Second, ESSL's input-argument error recovery provides a means of determining the minimum size you need for auxiliary storage. It uses the optionally-recoverable error 2015. For details, see "Using Auxiliary Storage in ESSL" on page 49. Third, you can have ESSL dynamically allocate *aux* and *aux2*. For details, see "Dynamic Allocation of Auxiliary Storage" on page 51.

>|When you use the ESSL SMP CUDA library subroutines, SCFTD, DCFTD, SRCFTD, DRCFTD, SCRFTD, and DCRFTD, use a small auxiliary working storage in host memory if the computation is offloaded to the GPUs. You can save significant memory by using the error handling facility to determine the size of auxiliary storage when you use the ESSL SMP CUDA library. |<

Performance and Accuracy Considerations

The following explain the performance and accuracy considerations for the Fourier transforms, convolution, and correlation subroutines. For further details about performance and accuracy, see Chapter 2, "Planning for ESSL," on page 33.

When Running on the Workstation Processors

There are ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 61.

Defining Arrays

The stride arguments, *inc1h, inc1x, inc1y, inc2x, inc2y, incx, incy, incmx, incmy*, *inc3x*, and *inc3y*, provide great flexibility in defining the input and output data arrays. The arrangement of data in storage, however, can have an effect upon cache performance. >|By using strides, you can have data scattered in storage, which might prevent the ESSL SMP CUDA library subroutines SCFTD, DCFTD, SRCFTD, DRCFTD, SCRFTD, and DCRFTD from running on the GPUs.|< Best performance is obtained with data closely spaced in storage and with elements of the sequence in contiguous locations. The optimum values for *inc1h, inc1x*, and *inc1y* are 1.

In writing the calling program, you may find it convenient to declare X or Y as a two-dimensional array. For example, you can declare X in a DIMENSION statement as `X(INC2X,M)`.

Fourier Transform Considerations

This describes some ways to optimize performance in the Fourier transform subroutines.

Setting Up Your Data

Many of the Fourier transform, convolution, and correlation subroutines provide the facility for processing many sequences in one call. For short sequences, for example 1024 elements or less, this facility should be used as much as possible. This provides improved performance compared to processing only one sequence at a time.

If possible, you should use the same array for input and output.

For improved performance, small values of *inc1x* and *inc1y* should be used, where applicable, preferably *inc1x* = 1 and *inc1y* = 1. A stride of 1 means the sequences elements are stored contiguously. Also, if possible, the sequences should be stored close to each other. For all the Fourier transform subroutines except _RCFT and _CRFT, >|or except when your transforms are offloaded to the GPUs when you use the SCFTD, DCFTD, SRCFTD, DRCFTD, SCRFTD, and DCRFTD subroutines from the ESSL SMP CUDA library,|< for best performance, you must use the STRIDE subroutine to determine the optimal stride(s) for your input or output data. Complete instructions on how to use STRIDE for each of these subroutines is included in "STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)" on page 1195.

To obtain the best performance in the three-dimensional Fourier transform subroutines, you should use strides, *inc2* and *inc3*, provided by the STRIDE subroutine and declare your three-dimensional data structure as a one-dimensional array. The three-dimensional Fourier transform subroutines assume that *inc1* for the array is 1. Therefore, each element x_{ijk} for $i = 0, 1, ..., n1-1$, $j = 0, 1, ..., n2-1$, and $k = 0, 1, ..., n3-1$ of the three-dimensional data structure of dimensions *n1* by *n2* by *n3* is stored in a one-dimensional array `X(0:L)` at location X(*l*), where *l* = *i*+*inc2*(*j*)+*inc3*(*k*). The minimum required value of L is calculated by inserting the maximum values for *i*, *j*, and *k* in the above equation, giving L = (*n1*-1)+*inc2*(*n2*-1)+*inc3*(*n3*-1). The minimum total size of array X is L+1. To ensure that this mapping is unique so no two elements x_{ijk} occupy the same array element, X(*l*), the subroutines have the following restriction: *inc2* ≥ *n1* and *inc3* ≥ (*inc2*)(*n2*). This arrangement of array data in storage leaves some blank space between successive planes of the array X. By determining the best size for this space, specifying an optimum *inc3* stride, the third dimension of the array does not create conflicts in the 3090 storage hierarchy.

If the *inc3* stride value returned by the STRIDE subroutine turns out to be a multiple of *inc2*, the array X can be declared as a three-dimensional array as X(*inc2*,*inc3*/*inc2*,*n3*); otherwise, it can be declared as either a one-dimensional array, `X(0:L)`, as described above, or a two-dimensional array `X(0:`*inc3*`-1,0:`*n3*`-1)`, where x_{ijk} is stored in X(*l*,*k*) where *l* = *i*+(*inc2*)(*j*). In such case, the transform cannot be offloaded to the GPUs when you use the ESSL SMP CUDA library.

Using the Scale Argument

If you must multiply either the input or the output sequences by a common factor, you can avoid the multiplication by letting the *scale* argument contain the factor. The subroutines multiply the sine and cosine values by the scale factor during the initialization. Thus, scaling takes no time after the initialization of the Fourier transform calculations.

>|Using GPUs

Performance may improve by offloading computations to the GPUs for some large transform sizes. To achieve optimal performance, experimentation is necessary by comparing performance results from the ESSL SMP library that uses data with strides provided by the STRIDE subroutine, and the ESSL SMP CUDA library that uses contiguous data.

|<

How the Fourier Transform Subroutines Achieve High Performance

There are two levels of optimization for the fast Fourier transforms (FFTs) in the ESSL library:

- For sequences with a large power of 2 length, we provide efficient implementations by factoring the transform length as follows:

  $N=N_1N_2N_3...N_p$

  where each N_i is a power of 2; the power of 2 used depends on the machine model.

  The cache optimization includes ordering of operations to maximize stride-1 data access and prefetching cache lines.

  Similar optimization techniques are used for sequence lengths which are not a power of 2 and mixed-radix FFT's are performed. Many short sequence FFT's have sequence size specific optimizations. Some of these optimizations were originally developed for a vector machine and have been adapted for cache based RISC machines (see references [“1” on page 1253], [“6” on page 1253], and [“8” on page 1253])
- The other optimization in the FFT routine is to treat multiple sequences as efficiently as possible. Techniques here include blocking sequences to fit into available CPU cache and transposing sequences to ensure stride-1 access. Whenever possible, the highest performance can be obtained when multiple sequences are transformed in a single call.
- >|For systems with NVIDIA GPUs, if the ESSL SMP CUDA library is used, the SCFTD, DCFTD, SRCFTD, DRCFTD, SCRFTD, and DCRFTD subroutines can offload large transforms to GPUs if the sequences length and data layout requirements are satisfied. For details, see the subroutine descriptions.|<

Convolution and Correlation Considerations

This describes some ways to optimize performance in the convolution and correlation subroutines.

Performance Tradeoffs between Subroutines

The subroutines SCON, SCOR, SACOR, SCOND, SCORD, SDCON, SDCOR, DDCON, and DDCOR compute convolutions, correlations, and autocorrelations using essentially the same methods. They make a decision, based on estimated timings, to use one of two methods:

- A direct method that is most efficient when one or both of the input sequences are short
- A direct method that is most efficient when the output sequence is short

Using this approach has the following advantages:

- In most cases, improved performance can be achieved for direct methods because:
  - No initialization is required.
  - No working storage or padding of sequences is necessary.
- In some cases, greater accuracy may be available.
- Negative strides can be used.

In general, using SCONF, SCORF, and SACORF provides the best performance, because the mixed-radix Fourier transform subroutines are used. However, if you can determine from your arguments that a direct method is preferred, you should use SCOND and SCORD instead. These give you better performance for the direct methods, and also give you additional capabilities.

In cases where there is doubt as to the best choice of a subroutine, perform timing experiments.

Special Uses of SCORD

The subroutine SCORD can perform the functions of SCON and SACOR; that is, it can compute convolutions and autocorrelations. To compute a convolution, you must specify a negative stride for ***h*** (see Example 4 in SCORD). To compute the autocorrelation, you must specify the two input sequences to be the same (see Example 5 in SCORD).

Special Uses of _DCON and _DCOR

The _DCON and _DCOR subroutines compute convolutions and correlations, respectively, by the direct method with decimated output. Setting the decimation interval *id* = 1 in SDCON and SDCOR provides the same function as SCOND and SCORD, respectively. Doing the same in DDCON and DDCOR provides long-precision versions of SCOND and SCORD, respectively, which are not otherwise available.

Accuracy When Direct Methods Are Used

The direct methods used by the convolution and correlation subroutines use vector operations to accumulate sums of products. The products are computed and accumulated in long precision. As a result, higher accuracy can be obtained in the final results for some types of data. For example, if input data consists only of integers, and if no intermediate and final numbers become too large (larger than 2^{24}-1 for short-precision computations and larger than 2^{56}-1 for long-precision computations), the results are exact.

However, when short-precision subroutines use the VSX unit to improve performance, they do not accumulate intermediate results in long precision.

Accuracy When Fourier Methods Are Used

The Fourier methods used by the convolution and correlation subroutines compute Fourier transforms of input data that is multiplied element-by-element in short-precision arithmetic. The inverse Fourier transform is then computed. There are internally generated rounding errors in the Fourier transforms. It has been shown in references ["117" on page 1259] and ["105" on page 1258] that, in the case of white noise data, the relative root mean square (RMS) error of the Fourier transform is proportional to $log_2 n$ with a very small proportionality factor. In general, with random, evenly distributed data, this is better than the RMS error of the direct method. However, one must keep in mind the fact that, while the Fourier method may yield a smaller root mean square error, there can be points with large relative errors. Thus, it can happen that some points, usually at the ends of the output sequence, can be obtained with greater relative accuracy with direct methods.

Convolutions and Correlations by Fourier Methods

The convolution and correlation subroutines that use the Fourier methods determine a sequence length *n*, whose Fourier transform is computed using ESSL subroutines. In the simple case where *iy0* = 0 for convolution or *iy0* = -*nh*+1 for correlation, *n* is chosen as a value greater than or equal to the following, which is also acceptable to the Fourier tranform subroutines:

nt = min(*nh*+*nx*-1, *ny*) for convolution and correlation
nt = min(*nx*+*nx*-1, *ny*) for autocorrelation

which is also acceptable to the Fourier subroutines.

Related Computation Considerations

This describes some key points about using the related-computation subroutines.

Accuracy Considerations

- Many of the subroutines performing short-precision computations provide increased accuracy by accumulating results in long precision. This is noted in the functional description for each subroutine.
- There are ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 61.

Fourier Transform Subroutines

This contains the Fourier transform subroutine descriptions.

SCFTD and DCFTD (Multidimensional Complex Fourier Transform)

Purpose

These subroutines compute a set of *m* *d*-dimensional discrete Fourier transforms of complex data.

Table 219: Data Types.

Data Types		
X, Y	*scale*	**Subroutine**
Short-precision complex	Short-precision real	SCFTD
Long-precision complex	Long-precision real	DCFTD

Notes:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.
3. >|On certain platforms that have NVIDIA GPUs, transform might be offloaded to GPUs if problem size and data layout requirements are met when ESSL SMP CUDA library is used.|<

Syntax

Language	Syntax
Fortran	CALL SCFTD | DCFTD (*init*, *d*, *x*, *incx*, *incmx*, *y*, *incy*, *incmy*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	scftd | dcftd (*init*, *d*, *x*, *incx*, *incmx*, *y*, *incy*, *incmy*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init

is a flag, where:

If *init* = 1, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 2, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*, and no SIMD algorithms are used (see "What ESSL Library Do You Want to Use?" on page 33). The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transforms of the given array is computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. The arguments *d*, *incx*, *incmx*, *incy*, *incmy*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, and *naux2* must be the same as when the subroutine was called for initialization with *init* = 1 or *init* = 2.

Specified as: an integer; 0 ≤ *init* ≤ 2.

d
is the dimension of the transform.

Specified as: an integer; $1 \le d \le 3$.

x
is the array X, consisting of *m* sequences of *d*-dimensional complex arrays to be transformed. Using zero-based indexing, $x_{j1,j2,...,jd,mm}$ is stored in location $j1(incx_1) + j2(incx_2) + ... + jd(incx_d) + mm(incmx)$ of the array X.

Specified as: an array of (at least) length $1 + incx_1(n_1\text{-}1) + ... + incx_d(n_d\text{-}1) + incmx(m\text{-}1)$, containing numbers of the data type indicated in Table 219 on page 962.

incx
is an array containing the strides between the elements in array X for each of the *d* dimensions.

Specified as: an array of length *d* containing integers; $incx_{1:d} > 0$.

incmx
is the stride between the first elements of the *d*-dimensional sequences in array X. (If $m = 1$, this argument is ignored.)

Specified as: an integer; $incmx > 0$.

y
See On Return.

incy
is an array containing the strides between the elements in array Y for each of the *d* dimensions.

Specified as: an array of length *d* containing integers; $incy_{1:d} > 0$.

incmy
is the stride between the first elements of the *d*-dimensional sequences in array Y. (If $m = 1$, this argument is ignored.)

Specified as: an integer; $incmy > 0$.

n
is an array containing the lengths of the dimensions of the array to be transformed.

Specified as: an array of length *d* containing integers; $0 \le n_{1:d} \le 1073479680$.

m
is the number of sequences to be transformed.

Specified as: an integer; $m > 0$.

isign
is an array that controls the direction of the transform (from time to frequency or from frequency to time). The sign of $Isign_i$ determines the signs in the exponents of W_{n1}, W_{n2}, ..., W_{nd}, where:

If $isign_i > 0$, $Isign_i = +$ (transforming time to frequency).

If $isign_i < 0$, $Isign_i = -$ (transforming frequency to time).

Specified as: an array of length *d* containing integers; $isign_{1:d} \neq 0$.

scale
is the scaling constant by which the transforms are multiplied. See "Function" on page 966 for its usage.

Specified as: a number of the data type indicated in Table 219 on page 962, where $scale \neq 0.0$.

aux1
is the working storage for this subroutine, where:

If $init > 0$, the working storage is computed.

If $init = 0$, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*.

Specified as: an integer; *naux1* > 7(d+1)+1 and *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas (see Processor-Independent Formulas for SCFTD for NAUX1 and NAUX2 and Processor-Independent Formulas for DCFTD for NAUX1 and NAUX2. For values between 7(d+1)+1 and the minimum value, you have the option of having the minimum value returned in this argument; for details, see On Return and “Using Auxiliary Storage in ESSL” on page 49.

aux2
has the following meaning:

If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*.

Specified as: an integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, the subroutine dynamically allocates the work area. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument; for details, see On Return and “Using Auxiliary Storage in ESSL” on page 49.

On Return

y
has the following meaning, where:

If *init* > 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of the results of the *m* *d*-dimensional discrete Fourier transforms. Using zero-based indexing, $y_{k1,k2,...,kd,mm}$ is stored in location $k1(incy_1) + k2(incy_2) + ... + kd(incy_d) + mm(incmy)$ of the array Y.

Returned as: an array of (at least) length $1 + incy_1(n_1\text{-}1) + ... + incy_d(n_d\text{-}1) + incmy(m\text{-}1)$, containing numbers of the data type indicated in Table 219 on page 962.

aux1
is the working storage for this subroutine, where:

If *init* > 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

naux1
contains the minimum value required for successful processing (as returned by the subroutine), provided that the following are true:

- You specified that error 2015 is recoverable.
- You specified an input value for *naux1* that is at least 7(d+1)+1 (but insufficient for the problem).
- There were no other errors.

Otherwise, it remains unchanged.

Returned as: an integer.

naux2

contains the minimum value required for successful processing (as returned by the subroutine), provided that the following are true:

- You specified that error 2015 is recoverable.
- You specified an input value for *naux2* that is greater than or equal to zero (but insufficient for the problem).
- There were no other errors.

Otherwise, it remains unchanged.

Returned as: an integer.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* > 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* > 0 and *init* = 0.
3. For optimal performance, the preferred value for $incx_1$ and $incy_1$ is 1.

   If you specify the same array for X and Y, then $incx_i$ and $incy_i$ for $i = 1,\ldots,d$ must be equal, and *incmx* and *incmy* must be equal. In this case, output overwrites input. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
4. You have the option of having the minimum required value for *naux1* and *naux2* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

>|The following notes apply to the ESSL SMP CUDA library:

1. Both initialization and computation steps are performed in the same way for the ESSL SMP CUDA library subroutines. The subroutines check whether the transforms can be done on GPUs. For GPU-supported transforms, if `ESSL_CUDA_HYBRID=no`, subroutines offload computations to GPUs. If `ESSL_CUDA_HYBRID=yes`, subroutines run computations on either CPUs or GPUs based on predicted performance. Performance might be affected if transforms are forced to run on GPUs.
2. Subroutines use the cuFFT library to offload computations to GPUs. For more information about cuFFT library, see the cuFFT library user guide.
3. Additional data layout requirements must be met to offload transform to GPUs. Sequence must be stored contiguously, so either the first or the last dimension strides must be 1 for both input and output arrays. Sequences cannot overlap in memory (for example, no row transforms). For three-dimensional transforms, data must be stored as a three-dimensional array.
4. Subroutines determine whether to use GPUs by predicting the relative performances of CPUs and GPUs based on the size of transforms, dimensions of transforms, number of transforms, number of CPU cores and their versions, GPU versions, and the pinning status of host memory. Only large transforms are offloaded to GPUs, if `ESSL_CUDA_HYBRID=yes`.
5. Available GPU memory can limit the size of transform that can be offloaded to GPUs.
6. For systems with more than 1 GPU, subroutines determine the number of GPUs that can be used to achieve better performance. Subroutines can use the cuFFT XT APIs to perform a single transform on multiple GPUs. The cuFFT XT APIs can be performed only on two or four GPUs for one-dimensional transforms.
7. If the transform is offloaded to GPUs, the only requirement for auxiliary storage is the minimum naux1 = 7*(d+1)+1. If the auxiliary storage is allocated by using the Processor-Independent Formulas for naux1 and naux2, subroutines can run successfully regardless of whether GPUs are used. To save memory, if GPUs are used, you can set naux1= 7*(d+1)+1 and naux2=0, and let error handling return

the auxiliary storage to your program. Unchanged value of the naux1 parameter means that computations will be offloaded to GPUs and additional auxiliary storage is not required any more.

8. When you use the ESSL SMP CUDA library, the number of threads that is specified must not be changed between calls to this subroutine with *init* > 0 and *init* = 0.

|<

Formulas

Processor-Independent Formulas for SCFTD for NAUX1 and NAUX2:

NAUX1 Formulas

If $\max(n_1,n_2,\ldots,n_d) \le 2048$, $naux1 = 30000d$.
If $\max(n_1,n_2,\ldots,n_d) > 2048$, $naux1 = 60000d+14.12(n_1+\ldots+n_d)$.

NAUX2 Formulas

If $\max(n_1,n_2,\ldots,n_d) < 252$, $naux2 = 20000$.
If $\max(n_1,n_2,\ldots,n_d) \ge 252$, $naux2 = 20000+(r+256)(s+8.56)$.

where:

$r = \max(n_1,n_2,\ldots,n_d)$ and
$s = \min(64,r)$

Processor-Independent Formulas for DCFTD for NAUX1 and NAUX2:

NAUX1 Formulas

If $\max(n_1,n_2,\ldots,n_d) \le 1024$, $naux1 = 30000d$.
If $\max(n_1,n_2,\ldots,n_d) > 1024$, $naux1 = 60000d+28.24(n_1+\ldots+n_d)$.

NAUX2 Formulas

If $\max(n_1,n_2,\ldots,n_d) < 252$, $naux2 = 20000$.
If $\max(n_1,n_2,\ldots,n_d) \ge 252$, $naux2 = 20000+(2r+256)(s+17.12)$.

where:

$r = \max(n_1,n_2,\ldots,n_d)$ and
$s = \min(64,r)$

Function

The set of *m* *d*-dimensional discrete Fourier transforms of complex data in array X, with results going into array Y, is expressed as follows:

$$y_{k1,k2,\ldots,kd,i} = scale\sum_{j1=0}^{n_1-1}\sum_{j2=0}^{n_2-1}\cdots\sum_{jd=0}^{n_d-1} x_{j1,j2,\ldots,jd,i}W_{n_1}^{(Isign_1)(j1)(k1)}W_{n_2}^{(Isign_2)(j2)(k2)}\cdots W_{n_d}^{(Isign_d)(jd)(kd)}$$

for:

$k1 = 0,\ldots,n_1-1$
$k2 = 0,\ldots,n_2-1$
.
.
.
$kd = 0,\ldots,n_d-1$
$i = 0,\ldots,m-1$

where:

$$W_{n_l} = e^{-2\pi(\sqrt{-1})/n_l}$$

for:

$l = 1,...,d$

and where:

$x_{j1,j2...jd,mm}$ are elements of the d-dimensional sequences in array X.
$y_{k1,k2...kd,mm}$ are elements of the d-dimensional sequences in array Y.

For *scale* = 1.0 and $isign_1 = isign_2 = ... = isign_d = 1$, you obtain the discrete Fourier transform (DFT), a function of frequency. The inverse Fourier transform is obtained with $scale = 1.0/n_1\, n_2...n_d$ and $isign_1 = isign_2 = ... = isign_d = -1$. See references["6" on page 1253], ["8" on page 1253], ["13" on page 1254], and ["32" on page 1254].

Two invocations of this subroutine are necessary:

1. With *init* > 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transforms.

If $n_i = 0$ for any i from 1 to d or if $m = 0$; no initialization or computation is performed.

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors
None

Input-Argument Errors

1. *init* < 0 or *init* > 2
2. $d < 1$ or $d > 3$
3. $incx_i \leq 0$ ($i = 1,...,d$)
4. $incmx \leq 0$
5. $incy_i \leq 0$ ($i = 1,...,d$)
6. $incmy \leq 0$
7. $n_i < 0$ or $n_i > 1073479680$ ($i = 1,...,d$)
8. $m < 0$
9. $isign_i = 0$ ($i = 1,...,d$)
10. *scale* = 0.0
11. *naux1* < 7(d+1)+1
12. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
13. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
14. The subroutine has not been initialized with the present arguments.

Examples

Example 1

This example shows an input array X with a set of four long-precision complex sequences:

$$e^{2\pi\left(\sqrt{-1}\right)jk/n}$$

for *j* = 0, 1, ..., *n*-1 with *n* = 8, and the single frequencies *k* = 0, 1, 2, and 3.

Note: X is the same input array used in Example 1.

The arrays are declared as follows:

```
     COMPLEX*16  X(0:31),Y(0:31)
     REAL*8      AUX1(10000),AUX2(1)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Note: Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT  D  X  INCX INCMX Y  INCY INCMY N  M  ISIGN SCALE AUX1  NAUX1  AUX2 NAUX2
            |    |  |    |    |   |    |    |   |  |    |     |     |     |      |    |
CALL DCFTD(INIT, 1, X, INCX,  8,  Y, INCY,  8,  N, 4, ISIGN, 1.0, AUX1, 10000, AUX2,  0)
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`INCX` is an array of length *d*.
`INCX(1) = 1`
`INCY` is an array of length *d*.
`INCY(1) = 1`
`N` is an array of length *d*.
`N(1) = 8`
`ISIGN` is an array of length *d*.
`ISIGN(1) = 1`
`SCALE = 1.0`

X contains the following four sequences:

```
(1.0000, 0.0000)  ( 1.0000,  0.0000)  ( 1.0000,  0.0000)  ( 1.0000,  0.0000)
(1.0000, 0.0000)  ( 0.7071,  0.7071)  ( 0.0000,  1.0000)  (-0.7071,  0.7071)
(1.0000, 0.0000)  ( 0.0000,  1.0000)  (-1.0000,  0.0000)  ( 0.0000, -1.0000)
(1.0000, 0.0000)  (-0.7071,  0.7071)  ( 0.0000, -1.0000)  ( 0.7071,  0.7071)
(1.0000, 0.0000)  (-1.0000,  0.0000)  ( 1.0000,  0.0000)  (-1.0000,  0.0000)
(1.0000, 0.0000)  (-0.7071, -0.7071)  ( 0.0000,  1.0000)  ( 0.7071, -0.7071)
(1.0000, 0.0000)  ( 0.0000, -1.0000)  (-1.0000,  0.0000)  ( 0.0000,  1.0000)
(1.0000, 0.0000)  ( 0.7071, -0.7071)  ( 0.0000, -1.0000)  (-0.7071, -0.7071)
```

Output:

Y contains the following four sequences:

```
(8.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (8.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (8.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (8.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
```

Example 2

This example shows how to compute a three-dimensional transform. In this example, INCX ≥ INCY, so the same array can be used for both input and output.

Note: X is the same input array used in Example 1.

The STRIDE subroutine is called to select good values for the INCY strides. (As explained below, STRIDE is not called for INCX.) Using the transform lengths (N(1) = 32, N(2) = 64, and N(3) = 40) along with the output data type (short-precision complex: 'C'), STRIDE is called once for each stride needed. First, it is called for INCY(2):

```
   CALL STRIDE (N(2),N(1),INCY(2),'C',0)
```

The output value returned for INCY(2) is 32. Then STRIDE is called again for INCY(3):

```
   CALL STRIDE (N(3),N(2)*INCY(2),INCY(3),'C',0)
```

The output value returned for INCY(3) is 2056. Because INCY(3) is not a multiple of INCY(2), Y is not declared as a three-dimensional array; it is declared as a two-dimensional array, Y(INCY(3),N(3)).

For equivalence, it is required that INCX(2) ≥ INCY(2) and INCX(3) ≥ INCY(3). Therefore, INCX(2) and INCY(2) are set as follows: INCX(2) = INCY(2) = 32.

To enable the X array to be declared as a three-dimensional array, INCX(3) must be a multiple of INCX(2). Therefore, its value is set as INCX(3) = 65(INCX(2)) = 2080.

The arrays are declared as follows:

```
      COMPLEX*8     X(32,65,40),Y(2056,40)
      REAL*8        AUX1(90000),AUX2(1),SCALE
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```
      EQUIVALENCE (X,Y)
```

Note: Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT  D  X  INCX INCMX Y  INCY INCMY N  M  ISIGN SCALE AUX1  NAUX1  AUX2 NAUX2
             |    |  |   |     |   |   |     |   |  |    |     |     |     |      |    |
CALL SCFTD(INIT, 3, X, INCX,  0,  Y, INCY,  0,  N, 1, ISIGN, 1.0, AUX1, 90000, AUX2,  0)
```

```
INIT     =    1 (for initialization)
INIT     =    0 (for computation)
INCX is an array of length d.
INCX(1)  =    1
INCX(2)  =   32
INCX(3)  = 2080
INCY is an array of length d.
INCY(1)  =    1
INCY(2)  =   32
INCY(3)  = 2056 N is an array of length d.
N(1)     =   32
N(2)     =   64
N(3)     =   40 ISIGN is an array of length d.
ISIGN(1) =    1
ISIGN(2) =    1
ISIGN(3) =    1
SCALE    =  1.0
X has (1.0,2.0) in location X(1,1,1) and (0.0,0.0) in all other locations.
```

Output:

Y has (1.0,2.0) in all locations.

SRCFTD and DRCFTD (Multidimensional Real-to-Complex Fourier Transform)

Purpose

These subroutines compute a set of *m* *d*-dimensional complex discrete Fourier transforms of real data.

Table 220: Data Types.

Data Types		
X, *scale*	Y	**Subroutine**
Short-precision real	Short-precision complex	SRCFTD
Long-precision real	Long-precision complex	DRCFTD

Notes:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.
3. >|On certain platforms with NVIDIA GPUs, transforms can be offloaded to GPUs if problems size and data layout requirements are fulfilled when ESSL SMP CUDA library is used.|<

Syntax

Language	Syntax
Fortran	CALL SRCFTD | DRCFTD (*init*, *d*,*x*, *incx*, *incmx*, *y*, *incy*, *incmy*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	srcftd | drcftd (*init*, *d*, *x*, *incx*, *incmx*, *y*, *incy*, *incmy*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init

is a flag, where:

If *init* = 1, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 2, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*, and no SIMD algorithms are used (see "What ESSL Library Do You Want to Use?" on page 33). The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transforms of the given array are computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. The arguments *d*, *incx*, *incmx*, *incy*, *incmy*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, and *naux2* must be the same as when the subroutine was called for initialization with *init* = 1 or *init* = 2.

Specified as: an integer; $0 \le init \le 2$.

d

is the dimension of the transform.

Specified as: an integer; $1 \le d \le 3$.

x

is the array X, consisting of *m* sequences of *d*-dimensional real arrays to be transformed. Using zero-based indexing, $x_{j1,j2,...,jd,mm}$ is stored in location $j1(incx_1) + j2(incx_2) + ... + jd(incx_d) + mm(incmx)$ of the array X.

Specified as: an array of (at least) length $1 + incx_1(n_1\text{-}1) + ... + incx_d(n_d\text{-}1) + incmx(m\text{-}1)$, containing numbers of the data type indicated in Table 220 on page 970.

incx

is an array containing the strides between the elements in array X for each of the *d* dimensions.

Specified as: an array of length *d* containing integers; $incx_{1:d} > 0$.

incmx

is the stride between the first elements of the *d*-dimensional sequences in array X. (If *m* = 1, this argument is ignored.)

Specified as: an integer; *incmx* > 0.

y

See On Return.

incy

is an array containing the strides between the elements in array Y for each of the *d* dimensions.

Specified as: an array of length *d* containing integers; $incy_{1:d} > 0$.

incmy

is the stride between the first elements of the *d*-dimensional sequences in array Y. (If *m* = 1, this argument is ignored.)

Specified as: an integer; *incmy* > 0.

n

is an array containing the lengths of the dimensions of the array to be transformed.

Specified as: an array of length *d* containing integers; $0 \le n_{1:d} \le 1073479680$.

m

is the number of sequences to be transformed.

Specified as: an integer; *m* > 0.

isign

is an array that controls the direction of the transform (from time to frequency or from frequency to time). The sign of $Isign_i$ determines the signs in the exponents of W_{n1}, W_{n2}, ..., W_{nd}, where:

If $isign_i > 0$, $Isign_i$ = + (transforming time to frequency).

If $isign_i < 0$, $Isign_i$ = - (transforming frequency to time).

Specified as: an array of length *d* containing integers; $isign_{1:d} \neq 0$.

scale

is the scaling constant by which the transforms are multiplied. See "Function" on page 974 for its usage.

Specified as: a number of the data type indicated in Table 220 on page 970, where *scale* ≠ 0.0.

aux1

is the working storage for this subroutine, where:

If *init* > 0, the working storage is computed.

If *init* = 0, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1

is the number of doublewords in the working storage specified in *aux1*.

Specified as: an integer; *naux1* > (4*d*+11) and *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas (see "Formulas" on page 974). For values between (4*d*+ 11) and the minimum value, you have the option of having the minimum value returned in this argument; for details, see On Return and "Using Auxiliary Storage in ESSL" on page 49.

aux2

has the following meaning:

If naux2 = 0 and error 2015 is unrecoverable, aux2 is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2

is the number of doublewords in the working storage specified in *aux2*.

Specified as: an integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, the subroutine dynamically allocates the work area. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see On Return and "Using Auxiliary Storage in ESSL" on page 49.

On Return

y

has the following meaning, where:

If *init* > 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of the results of the *m* *d*-dimensional complex discrete Fourier transforms. Using zero-based indexing, $y_{k1,k2,...,kd,mm}$ is stored in location $k1(incy_1) + k2(incy_2) + ... + kd(incy_d) + mm(incmy)$ of the array Y. Due to complex conjugate symmetry, the output consists of only the first $n_1/2+1$ values along the first dimension of the array, for $k1 = 0, 1, ..., n_1/2$.

Returned as: an array of (at least) length $1 + incy_1(n_1-1) + ... + incy_d(n_d-1) + incmy(m-1)$, containing numbers of the data type indicated in Table 220 on page 970.

aux1

is the working storage for this subroutine, where:

If *init* > 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

naux1

contains the minimum value required for successful processing (as returned by the subroutine), provided that the following are true:

- You specified that error 2015 is recoverable.
- You specified an input value for *naux1* that is at least (4*d*+11) (but insufficient for the problem).
- There were no other errors.

Otherwise, it remains unchanged.

Returned as: an integer.

naux2

contains the minimum value required for successful processing (as returned by the subroutine), provided that the following are true:

- You specified that error 2015 is recoverable.
- You specified an input value for *naux2* that is greater than or equal to zero (but insufficient for the problem).
- There were no other errors.

Otherwise, it remains unchanged.

Returned as: an integer.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* > 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* > 0 and *init* = 0.
3. For optimal performance, the preferred value for $incx_1$ and $incy_1$ is 1.

   If you specify the same array for X and Y, then:

   - $incx_i$ must equal $2(incy_i)$, for $i = 2,...,d$
   - *incmx* must be equal to *2(incmy)* if $m > 1$

   In this case, output overwrites input. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
4. You have the option of having the minimum required value for *naux1* and *naux2* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

The following notes apply to the ESSL SMP CUDA library:

1. Both initialization and computation steps are performed in the same way for the ESSL SMP CUDA library subroutines. The subroutines check whether the transforms can be done on GPUs. For GPU-supported transforms, if `ESSL_CUDA_HYBRID=no`, subroutines offload computations to GPUs. If `ESSL_CUDA_HYBRID=yes`, subroutines run computations on either CPUs or GPUs based on predicted performance. Performance might be affected if transforms are forced to run on GPUs.
2. Subroutines use the cuFFT library to offload computations to GPUs. For more information about cuFFT library, see the cuFFT library user guide.
3. Only forward-transform computations (time to frequency) can be offloaded to GPUs.
4. Additional data layout requirements must be met to offload transform to GPUs. Sequence must be stored contiguously, so either the first or the last dimension strides must be 1 for both input and output arrays. Sequences cannot overlap in memory (for example, no row transforms). For three-dimensional transforms, data must be stored as a three-dimensional array.
5. Subroutines determine whether to use GPUs by predicting the relative performances of CPUs and GPUs based on the size of transforms, dimensions of transforms, number of transforms, number of CPU cores and their versions, GPU versions, and the pinning status of host memory. Only large transforms are offloaded to GPUs, if `ESSL_CUDA_HYBRID=yes`.
6. Available GPU memory can limit the size of transform that can be offloaded to GPUs.
7. For systems with more than 1 GPU, subroutines determine the number of GPUs that can be used to achieve better performance. However, a single transform can be offloaded to only one GPU.
8. If the transform is offloaded to GPUs, the only requirement for auxiliary storage is the minimum `naux1` = 4*d+11. If the auxiliary storage is allocated by using the Processor-Independent Formulas for `naux1` and `naux2`, subroutines can run successfully regardless of whether GPUs are used. To save memory, if GPUs are used, you can set `naux1`= 4*d+11 and `naux2`=0, and let error handling return the auxiliary

storage to your program. Unchanged value of the naux1 parameter means that computations will be offloaded to GPUs and additional auxiliary storage is not required any more.

9. When you use the ESSL SMP CUDA library, the number of threads that is specified must not be changed between calls to this subroutine with *init* > 0 and *init* = 0.

|<

Formulas

Processor-Independent Formulas for SRCFTD for NAUX1 and NAUX2:

NAUX1 Formulas

If $\max(n_1,n_2,...,n_d) \le 2048$, $naux1 = 60000d$.
If $\max(n_1,n_2,...,n_d) > 2048$, $naux1 = 60000d+14.12(n_1+...+n_d)$.

NAUX2 Formulas

If $\max(n_1,n_2,...,n_d) < 252$, $naux2 = 20000$.
If $\max(n_1,n_2,...,n_d) \ge 252$, $naux2 = 20000+(r+256)(s+8.56)$.

where:

$r = \max(n_1,n_2,...,n_d)$ and
$s = \min(64,r)$

Processor-Independent Formulas for DRCFTD for NAUX1 and NAUX2:

NAUX1 Formulas

If $\max(n_1,n_2,...,n_d) \le 1024$, $naux1 = 60000d$.
If $\max(n_1,n_2,...,n_d) > 1024$, $naux1 = 60000d+28.24(n_1+...+n_d)$.

NAUX2 Formulas

If $\max(n_1,n_2,...,n_d) < 252$, $naux2 = 20000$.
If $\max(n_1,n_2,...,n_d) \ge 252$, $naux2 = 20000+(2r+256)(s+17.12)$.

where:

$r = \max(n_1,n_2,...,n_d)$ and
$s = \min(64,r)$

Function

The set of *m* *d*-dimensional complex conjugate even discrete Fourier transforms of real data in array *x* with results going into array *y* is expressed as follows:

$$y_{k1,k2,...,kd,i} = scale \sum_{j1=0}^{n_1-1} \sum_{j2=0}^{n_2-1} \cdots \sum_{jd=0}^{n_d-1} x_{j1,j2,...,jd,i} W_{n_1}^{(Isign_1)(j1)(k1)} W_{n_2}^{(Isign_2)(j2)(k2)} \cdots W_{n_d}^{(Isign_d)(jd)(kd)}$$

for:

$k1 = 0,...,n_1-1$
$k2 = 0,...,n_2-1$
.
.
.

$kd = 0,...,n_d-1$
$i = 0,...,m-1$

where:

$$W_{n_l} = e^{-2\pi(\sqrt{-1})/n_l}$$

for:

$l = 1,...,d$

and where:

$x_{j1,j2...jd,mm}$ are elements of the *d*-dimensional sequences in array X.
$y_{k1,k2...kd,mm}$ are elements of the *d*-dimensional sequences in array Y.

For *scale* = 1.0 and $isign_1 = isign_2 = ... = isign_d = 1$, you obtain the discrete Fourier transform (DFT), a function of frequency. The inverse Fourier transform is obtained with $scale = 1.0/n_1\, n_2...n_d$ and $isign_1 = isign_2 = ... = isign_d = -1$. See references["6" on page 1253], ["8" on page 1253], ["13" on page 1254], and ["32" on page 1254]

Two invocations of this subroutine are necessary:

1. With *init* > 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transforms.

If $n_i = 0$ for any *i* from 1 to *d* or if *m* = 0; no initialization or computation is performed.

Error conditions

Resource Errors
Error 2015 is unrecoverable, unable to allocate work area, and internal deallocation error.

Computational Errors
None

Input-Argument Errors

1. *init* < 0 or *init* > 2
2. *d* < 1 or *d* > 3
3. $incx_i \le 0$ ($i = 1,...,d$)
4. *incmx* ≤ 0
5. $incy_i \le 0$ ($i = 1,...,d$)
6. *incmy* ≤ 0
7. $n_i < 0$ or $n_i > 1073479680$ ($i = 1,...,d$)
8. *m* < 0
9. $isign_i = 0$ ($i = 1,...,d$)
10. *scale* = 0.0
11. *naux1* < 4*d*+11
12. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
13. Error 2015 is recoverable or *naux2* ≠ 0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
14. The subroutine has not been initialized with the present arguments.

Examples

Example 1

This example shows an input array X with a set of *m* cosine sequences cos(2π*jk*/*n*), *j* = 0, 1, ..., 15 with the single frequencies *k* = 0, 1, 2, 3. The Fourier transform of the cosine sequence with frequency *k* = 0 or n/2 has 1.0 in the 0 or n/2 position, respectively, and zeros elsewhere. For all other *k*, the Fourier transform has 0.5 in the *k* position and zeros elsewhere. The arrays are declared as follows:

```
     REAL*4     X(0:100)
     COMPLEX*8  Y(0:50)
     REAL*8     AUX1(1000), AUX2
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT  D  X   INCX INCMX  Y  INCY INCMY  N   M  ISIGN   SCALE  AUX1   NAUX1
AUX2   NAUX2
            |     |  |     |    |    |   |    |     |   |    |      |      |      |
|       |
CALL SRCFTD(INIT, 1  X,  INCX, 16,   Y, INCY, 9,    N,  4, ISIGN,  SCALE, AUX1,  1000,
AUX2,    0  )
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`INCX` is an array of length *d*.
`INCX(1) = 1`
`INCY` is an array of length *d*.
`INCY(1) = 1`
`N` is an array of length *d*.
`N(1) = 16`
`ISIGN` is an array of length *d*.
`ISIGN(1) = 1`
`SCALE = 1.0 / 16`

X contains the following four sequences:

```
1.0000  1.0000  1.0000  1.0000
1.0000  0.9239  0.7071  0.3827
1.0000  0.7071  0.0000 -0.7071
1.0000  0.3827 -0.7071 -0.9239
1.0000  0.0000 -1.0000  0.0000
1.0000 -0.3827 -0.7071  0.9239
1.0000 -0.7071  0.0000  0.7071
1.0000 -0.9239  0.7071 -0.3827
1.0000 -1.0000  1.0000 -1.0000
1.0000 -0.9239  0.7071 -0.3827
1.0000 -0.7071  0.0000  0.7071
1.0000 -0.3827 -0.7071  0.9239
1.0000  0.0000 -1.0000  0.0000
1.0000  0.3827 -0.7071 -0.9239
1.0000  0.7071  0.0000 -0.7071
1.0000  0.9239  0.7071  0.3827
```

Output:

Y contains the following four sequences:

```
(1.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.5000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.5000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.5000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
```

```
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
```

Example 2

This example shows how to compute a three-dimensional transform.

The STRIDE subroutine is called to select good values for the INCY strides (as the following explains, STRIDE is not called for INCX.) Using the transform lengths (`N(1) = 33`, `N(2) = 64`, and `N(3) = 40`) along with the output data type (short-precision complex: 'C'), STRIDE is called once for each stride needed. First, it is called for `INCY(2)`:

```
CALL STRIDE (N(2),N(1)/2+1,INCY(2),'C',0)
```

The output value returned for`INCY(2)` is 18. Then STRIDE is called again for `INCY(3)`:

```
CALL STRIDE (N(3),N(2)*INCY(2),INCY(3),'C',0)
```

The output value returned for `INCY(3)` is 1160. Because `INCY(3)` is not a multiple of `INCY(2)`, Y is not declared as a three-dimensional array; it is declared as a two-dimensional array, `Y(INCY(3),N(3))`.

For equivalence, it is required that `INCX(2) = 2(INCY(2))` and `INCX(3) = 2(INCY(3))`. Therefore, `INCX(2)`, `INCY(2)`, `INCX(3)` and `INCY(3)` are set as follows:

```
    INCY(2) = 18
    INCX(2) = 36
    INCY(3) = 1160
    INCX(3) = 2320
```

The arrays are declared as follows:

```
    REAL*4 X(2320,40), SCALE
    COMPLEX*8 Y(1160,40)
    REAL*8 AUX1(5000),AUX2
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```
EQUIVALENCE (X,Y)
```

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT   D  X  INCX  INCMX Y  INCY  INCMY  N   M  ISIGN  SCALE  AUX1   NAUX1
AUX2   NAUX2
             |     |  |   |      |   |   |      |    |   |    |      |     |       |
|       |
CALL SRCFTD(INIT,  3  X, INCX,   0,  Y, INCY,   0    N,  1, ISIGN,  1.0,  AUX1,  5000,
AUX2,    0)
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
INCX is an array of length *d*.
`INCX(1)` = 1
`INCX(2)` = 36
`INCX(3)` = 2320
INCY is an array of length *d*.
`INCY(1)` = 1
`INCY(2)` = 18
`INCY(3)` = 1160
N is an array of length *d*.
`N(1)` = 33
`N(2)` = 64

`N(3)` = 40
`ISIGN` is an array of length *d*.
`ISIGN(1)` = 1
`ISIGN(2)` = 1
`ISIGN(3)` = 1
`SCALE` = 1.0
X has 1.0 in location `X(1,1)` and 0.0 in all other locations.

Output:

Y has (1.0,0.0) in all locations.

SCRFTD and DCRFTD (Multidimensional Complex-to-Real Fourier Transform)

Purpose

These subroutines compute a set of *m* *d*-dimensional real discrete Fourier transforms of complex conjugate even data.

Table 221: Data Types.

Data Types		
X, *scale*	Y	**Subroutine**
Short-precision real	Short-precision complex	SCRFTD
Long-precision real	Long-precision complex	DCRFTD

Notes:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.
3. >|On certain platforms that have NVIDIA GPUs, transform might be offloaded to GPUs if problem size and data layout requirements are met when ESSL SMP CUDA library is used.|<

Syntax

Language	**Syntax**
Fortran	CALL SCRFTD | DCRFTD (*init*, *d*, *x, incx, incmx, y, incy, incmy, n, m, isign, scale, aux1, naux1, aux2, naux2)*
C and C++	scrftd | dcrftd (*init*, *d*, *x*, *incx*, *incmx*, *y*, *incy*, *incmy*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init

is a flag, where:

If *init* = 1, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 2, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*, and no SIMD algorithms are used (see "What ESSL Library Do You Want to Use?" on page 33). The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transforms of the given array are computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. The arguments *d*, *incx*, *incmx*, *incy*, *incmy*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, and *naux2* must be the same as when the subroutine was called for initialization with *init* = 1 or *init* = 2.

Specified as: an integer; $0 \le init \le 2$.

d

is the dimension of the transform.

Specified as: an integer; $1 \le d \le 3$.

x

is the array X, consisting of *m* sequences of *d*-dimensional complex arrays to be transformed. Using zero-based indexing, $x_{j1,j2,\ldots,jd,mm}$ is stored in location $j1(incx_1) + j2(incx_2) + \ldots + jd(incx_d) + mm(incmx)$ of the array X. Due to complex conjugate symmetry, the input consists of only the first $n1/2+1$ values along the first dimension of the array, for $j1 = 0, 1, \ldots, n_1/2$.

Specified as: an array of (at least) length $1 + incx_1(n_1\text{-}1) + \ldots + incx_d(n_d\text{-}1) + incmx(m\text{-}1)$, containing numbers of the data type indicated in Table 221 on page 978.

incx

is an array containing the strides between the elements in array X for each of the *d* dimensions.

Specified as: an array of length *d* containing integers; $incx_{1:d} > 0$.

incmx

is the stride between the first elements of the *d*-dimensional sequences in array X. (If *m* = 1, this argument is ignored.)

Specified as: an integer; $incmx > 0$.

y

See On Return.

incy

is an array containing the strides between the elements in array Y for each of the *d* dimensions.

Specified as: an array of length *d* containing integers; $incy_{1:d} > 0$.

incmy

is the stride between the first elements of the *d*-dimensional sequences in array Y. (If *m* = 1, this argument is ignored.)

Specified as: an integer; $incmy > 0$.

n

is an array containing the lengths of the dimensions of the array to be transformed.

Specified as: an array of length *d* containing integers; $0 \le n_{1:d} \le 1073479680$.

m

is the number of sequences to be transformed.

Specified as: an integer; $m > 0$.

isign

is an array that controls the direction of the transform (from time to frequency or from frequency to time). The sign of $Isign_i$ determines the signs in the exponents of $W_{n1}, W_{n2}, \ldots, W_{nd}$, where:

If $isign_i > 0$, $Isign_i$ = + (transforming time to frequency).

If $isign_i < 0$, $Isign_i$ = - (transforming frequency to time).

Specified as: an array of length *d* containing integers; $isign_{1:d} \ne 0$.

scale

is the scaling constant by which the transforms are multiplied. See "Function" on page 983 for its usage.

Specified as: a number of the data type indicated in Table 221 on page 978, where *scale* ≠ 0.0.

aux1
is the working storage for this subroutine, where:

If *init* > 0, the working storage is computed.

If *init* = 0, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*.

Specified as: an integer; *naux1* > (4*d*+ 11) and *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas (see "Formulas" on page 982). For values between (4*d* + 11) and the minimum value, you have the option of having the minimum value returned in this argument; for details, see On Return and "Using Auxiliary Storage in ESSL" on page 49.

aux2
has the following meaning:

If naux2 = 0 and error 2015 is unrecoverable, aux2 is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*.

Specified as: an integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, the subroutine dynamically allocates the work area. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see On Return and "Using Auxiliary Storage in ESSL" on page 49.

On Return

x
has the following meaning, where:

If *init* > 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this argument is not used, and its contents remain unchanged if one of the following is true:

- $d = 1$
- $incx_1 = 1$ and $incy_1 = 1$

Otherwise, *x* is overwritten; that is, the original input is not preserved.

y
has the following meaning, where:

If *init* > 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of the results of the *m* *d*-dimensional discrete Fourier transforms of complex conjugate even data. Using zero-based indexing, $y_{k1,k2,...,kd,mm}$ is stored in location $k1(incy_1) + k2(incy_2) + ... + kd(incy_d) + mm(incmy)$ of the array Y.

Returned as: an array of (at least) length $1 + incy_1(n_1\text{-}1) + ... + incy_d(n_d\text{-}1) + incmy(m\text{-}1)$, containing numbers of the data type indicated in Table 221 on page 978.

aux1
is the working storage for this subroutine, where:

If *init* > 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

naux1
contains the minimum value required for successful processing (as returned by the subroutine), provided that the following are true:

- You specified that error 2015 is recoverable.
- You specified an input value for *naux1* that is at least (4*d*+11) (but insufficient for the problem).
- There were no other errors.

Otherwise, it remains unchanged.

Returned as: an integer.

naux2
contains the minimum value required for successful processing (as returned by the subroutine), provided that the following are true:

- You specified that error 2015 is recoverable.
- You specified an input value for *naux2* that is greater than or equal to zero (but insufficient for the problem).
- There were no other errors.

Otherwise, it remains unchanged.

Returned as: an integer.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* > 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* > 0 and *init* = 0.
3. If $incx_1$ = 1 and $incy_1$ = 1, then:
   - $incy_i$ must be even for $i = 2,...,d$
   - min(*incmy*, $incy_2$,...,$incy_d$) ≥ 2(n_1/2+1)
4. For optimal performance, the preferred value for $incx_1$ and $incy_1$ is 1.

   If you specify the same array for X and Y, then:

   - $incy_i$ must equal $2(incx_i)$, for $i = 2,...,d$
   - *incmy* must be equal to *2(incmx)* if *m* > 1

   In this case, output overwrites input. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
5. You have the option of having the minimum required value for *naux1* and *naux2* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

>|The following notes apply to the ESSL SMP CUDA library:

1. Both initialization and computation steps are performed in the same way for the ESSL SMP CUDA library subroutines. The subroutines check whether the transforms can be done on GPUs. For GPU-supported transforms, if `ESSL_CUDA_HYBRID=no`, subroutines offload computations to GPUs. If

ESSL_CUDA_HYBRID=yes, subroutines run computations on either CPUs or GPUs based on predicted performance. Performance might be affected if transforms are forced to run on GPUs.

2. Subroutines use the cuFFT library to offload computations to GPUs. For more information about cuFFT library, see the cuFFT library user guide.
3. Only backward-transform computations (frequency to time) can be offloaded to GPUs.
4. Additional data layout requirements must be met to offload transform to GPUs. Sequence must be stored contiguously, so either the first or the last dimension strides must be 1 for both input and output arrays. Sequences cannot overlap in memory (for example, no row transforms). For three-dimensional transforms, data must be stored as a three-dimensional array.
5. Subroutines determine whether to use GPUs by predicting the relative performances of CPUs and GPUs based on the size of transforms, dimensions of transforms, number of transforms, number of CPU cores and their versions, GPU versions, and the pinning status of host memory. Only large transforms are offloaded to GPUs, if ESSL_CUDA_HYBRID=yes.
6. Available GPU memory can limit the size of transform that can be offloaded to GPUs.
7. For systems with more than 1 GPU, subroutines determine the number of GPUs that can be used to achieve better performance. However, a single transform can be offloaded to only one GPU.
8. If the transform is offloaded to GPUs, the only requirement for auxiliary storage is the minimum naux1 = 4*d+11. If the auxiliary storage is allocated by using the Processor-Independent Formulas for naux1 and naux2, subroutines can run successfully regardless of whether GPUs are used. To save memory, if GPUs are used, you can set naux1= 4*d+11 and NAUX2=0, and let error handling return the auxiliary storage to your program. Unchanged value of the naux1 parameter means that computations will be offloaded to GPUs and additional auxiliary storage is not required any more.
9. When you use the ESSL SMP CUDA library, the number of threads that is specified must not be changed between calls to this subroutine with *init* > 0 and *init* = 0.

|<

Formulas

Processor-Independent Formulas for SCRFTD for NAUX1 and NAUX2:

NAUX1 Formulas

If $\max(n_1,n_2,\ldots,n_d) \le 2048$, $naux1 = 60000d$.
If $\max(n_1,n_2,\ldots,n_d) > 2048$, $naux1 = 60000d+14.12(n_1+\ldots+n_d)$.

NAUX2 Formulas

If $\max(n_1,n_2,\ldots,n_d) < 252$, $naux2 = 20000$.
If $\max(n_1,n_2,\ldots,n_d) \ge 252$, $naux2 = 20000+(r+256)(s+8.56)$.

where:

$r = \max(n_1,n_2,\ldots,n_d)$ and
$s = \min(64,r)$

Processor-Independent Formulas for DCRFTD for NAUX1 and NAUX2:

NAUX1 Formulas

If $\max(n_1,n_2,\ldots,n_d) \le 1024$, $naux1 = 60000d$.
If $\max(n_1,n_2,\ldots,n_d) > 1024$, $naux1 = 60000d+28.24(n_1+\ldots+n_d)$.

NAUX2 Formulas

If $\max(n_1,n_2,\ldots,n_d) < 252$, $naux2 = 20000$.
If $\max(n_1,n_2,\ldots,n_d) \ge 252$, $naux2 = 20000+(2r+256)(s+17.12)$.

where:

$r = \max(n_1, n_2, \ldots, n_d)$ and
$s = \min(64, r)$

Function

The set of *m* *d*-dimensional real discrete Fourier transforms of complex conjugate even data in array *x* with results going into array *y* is expressed as follows:

$$y_{k1,k2,\ldots,kd,i} = scale \sum_{j1=0}^{n_1-1} \sum_{j2=0}^{n_2-1} \cdots \sum_{jd=0}^{n_d-1} x_{j1,j2,\ldots,jd,i} W_{n_1}^{(Isign_1)(j1)(k1)} W_{n_2}^{(Isign_2)(j2)(k2)} \cdots W_{n_d}^{(Isign_d)(jd)(kd)}$$

for:

$k1 = 0,\ldots,n_1-1$
$k2 = 0,\ldots,n_2-1$
.
.
.
$kd = 0,\ldots,n_d-1$
$i = 0,\ldots,m-1$

where:

$$W_{n_l} = e^{-2\pi(\sqrt{-1})/n_l}$$

for:

$l = 1,\ldots,d$

and where:

$x_{j1,j2\ldots jd,mm}$ are elements of the *d*-dimensional sequences in array X.
$y_{k1,k2\ldots kd,mm}$ are elements of the *d*-dimensional sequences in array Y.

For *scale* = 1.0 and $isign_1 = isign_2 = \ldots = isign_d = 1$, you obtain the discrete Fourier transform (DFT), a function of frequency. The inverse Fourier transform is obtained with $scale = 1.0/n_1\, n_2 \ldots n_d$ and $isign_1 = isign_2 = \ldots = isign_d = -1$. See references[“6” on page 1253], [“8” on page 1253], [“13” on page 1254], and [“32” on page 1254].

Two invocations of this subroutine are necessary:

1. With *init* > 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transforms.

If $n_i = 0$ for any *i* from 1 to *d* or if *m* = 0; no initialization or computation is performed.

Error conditions

Resource Errors
Error 2015 is unrecoverable, unable to allocate work area, and internal deallocation error.

Computational Errors
None

Input-Argument Errors

1. *init* < 0 or *init* > 2
2. *d* < 1 or *d* >3
3. $incx_i \le 0$ (*i* = 1,...,*d*)
4. *incmx* ≤ 0
5. $incy_i \le 0$ (*i* = 1,...,*d*)
6. *incmy* ≤ 0
7. $n_i < 0$ or $n_i > 1073479680$ (*i* = 1,...,*d*)
8. *m* ≤ 0
9. $isign_i = 0$ (*i* = 1,...,*d*)
10. *scale* = 0.0
11. *naux1*< 4*d*+11.
12. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
13. Error 2015 is recoverable or *naux2* ≠ 0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
14. The subroutine has not been initialized with the present arguments.

Examples

Example 1

This example shows how to compute a single one-dimensional transform.

The arrays are declared as follows:

```
    COMPLEX*8 X(0:6)
    REAL*8 AUX1(100),AUX2
 REAL*4 Y(0:11)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
             INIT   D  X  INCX  INCMX  Y  INCY  INCMY  N   M  ISIGN  SCALE  AUX1   NAUX1
AUX2   NAUX2
              |     |  |   |      |    |   |      |    |   |    |      |     |       |
|       |
CALL SCRFTD( INIT,  1  X, INCX,   7,   Y, INCY,  12,   N,  1, ISIGN,  1.0,  AUX1,   100,
AUX2,    0)
```

INIT = 1 (for initialization)
INIT = 0 (for computation)
INCX is an array of length *d*.
INCX(1) = 1
INCY is an array of length *d*.
INCY(1) = 1
N is an array of length *d*.
N(1) = 12
ISIGN is an array of length *d*
ISIGN(1) = 1

X contains the following sequence:

```
    (1.0, 0.0)
    (0.0, 0.0)
    (0.0, 0.0)
    (0.0, 0.0)
    (0.0, 0.0)
    (0.0, 0.0)
    (0.0, 0.0)
```

Output:

Y contains the following sequence:

```
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
```

Example 2

This example shows how to compute a 3-dimensional Fourier transform. This example requires additional storage for array Y.

The arrays are declared as follows:

```
        COMPLEX*8     X(4,3,2)
        REAL*4        Y(9,3,2)
        REAL*8        AUX1(5000, AUX2)
```

First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
             INIT  D  X  INCX  INCMX   Y  INCY  INCMY  N   M  ISIGN  SCALE  AUX1   NAUX1
AUX2   NAUX2
               |   |  |   |      |     |   |      |    |   |    |      |     |       |
|       |
CALL SCRFTD( INIT, 3  X,  0,   INCMX,  Y, INCY,   0,   N,  1, ISIGN,  1.0,  AUX1,  5000,
AUX2,    0)
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
INCX is an array of length *d*.
`INCX(1) = 1`
`INCX(2) = 4`
`INCX(3) = 12`
INCY is an array of length *d*
`INCX(1) = 1`
`INCX(2) = 9`
`INCX(3) = 27`
N is an array of length *d*
`N(1) = 7`
`N(2) = 3`
`N(3) = 2`
ISIGN is an array of length *d*
`ISIGN(1) = 1`

```
ISIGN(2) = 1
ISIGN(3) = 1
```

X has `(1.0,0.0)` in location `X(1,1,1)` and `(0.0,0.0)` in all other locations.

Output:

Y(*i,j,k*) = 1.0 for *i* = 1,...,7; *j* = 1,...,3; *k* = 1,2

Y(*i,j,k*) is unchanged for *i* = 8,9; *j* = 1,...,3; *k* = 1,2

SCFT and DCFT (Complex Fourier Transform)

Purpose

These subroutines compute a set of *m* complex discrete *n*-point Fourier transforms of complex data.

Table 222: Data Types.

Data Types		
X, Y	*scale*	**Subroutine**
Short-precision complex	Short-precision real	SCFT
Long-precision complex	Long-precision real	DCFT

Note:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	**Syntax**
Fortran	CALL SCFT | DCFT (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	scft | dcft (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init

is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transforms of the given sequences are computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: an integer. It can have any value.

x

is the array X, consisting of *m* sequences of length *n*.

Specified as: an array of (at least) length 1+(*n*-1)*inc1x*+(*m*-1)*inc2x*, containing numbers of the data type indicated in Table 222 on page 986.

inc1x
is the stride between the elements within each sequence in array X.

Specified as: an integer; *inc1x* > 0.

inc2x
is the stride between the first elements of the sequences in array X. (If *m* = 1, this argument is ignored.) Specified as: an integer; *inc2x* > 0.

y
See On Return.

inc1y
is the stride between the elements within each sequence in array Y.

Specified as: an integer; *inc1y* > 0.

inc2y
is the stride between the first elements of each sequence in array Y. (If *m* = 1, this argument is ignored.) Specified as: an integer; *inc2y* > 0.

n
is the length of each sequence to be transformed.

Specified as: an integer; *n* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument, as well as in the optionally-recoverable error 2030. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

m
is the number of sequences to be transformed.

Specified as: an integer; *m* > 0.

isign
controls the direction of the transform, determining the sign *Isign* of the exponent of W_n, where:

If *isign* = positive value, *Isign* = + (transforming time to frequency).

If *isign* = negative value, *Isign* = - (transforming frequency to time).

Specified as: an integer; *isign* > 0 or *isign* < 0.

scale
is the scaling constant *scale*. See "Function" on page 989 for its usage.

Specified as: a number of the data type indicated in Table 222 on page 986, where *scale* > 0.0 or *scale* < 0.0

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, the working storage is computed.

If *init* = 0, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*.

Specified as: an integer; *naux1* > 7 (32-bit integer arguments) or 13 (64-bit integer arguments) and *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 7 (32-bit integer arguments) or 13 (64-bit integer arguments) and the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

naux2
is the number of doublewords in the working storage specified in *aux2*.

Specified as: an integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, SCFT and DCFT dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

On Return

y

has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of the results of the *m* discrete Fourier transforms, each of length *n*.

Returned as: an array of (at least) length 1+(*n*-1)*inc1y*+(*m*-1)*inc2y*, containing numbers of the data type indicated in Table 222 on page 986.

aux1

is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* ≠ 0 and *init* = 0.
3. For optimal performance, the preferred value for *inc1x* and *inc1y* is 1. This implies that the sequences are stored with stride 1. The preferred value for *inc2x* and *inc2y* is *n*. This implies that sequences are stored one after another without any gap.

   It is possible to specify sequences in the transposed form—that is, as rows of a two-dimensional array. In this case, *inc2x* (or *inc2y*) = 1 and *inc1x* (or *inc1y*) is equal to the leading dimension of the array. One can specify either input, output, or both in the transposed form by specifying appropriate values for the stride parameters. For selecting optimal values of *inc1x* and *inc1y* for _CFT, you should use "STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)" on page 1195. Example 1 in the STRIDE subroutine description explains how it is used for _CFT.

   If you specify the same array for X and Y, then *inc1x* and *inc1y* must be equal, and *inc2x* and *inc2y* must be equal. In this case, output overwrites input. If *m* = 1, the *inc2x* and *inc2y* values are not used by the subroutine. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Formulas

Processor-Independent Formulas for SCFT for NAUX1 and NAUX2:

NAUX1 Formulas

For 32-bit integer arguments:

If *n* ≤ 8192, use *naux1* = 20000.
If *n* > 8192, use *naux1* = 20000+1.14*n*.

For 64-bit integer arguments:

If $n \leq 8192$, use *naux1* = 30000.
If $n > 8192$, use *naux1* = 30000+1.14*n*.

NAUX2 Formulas

If $n \leq 8192$, use *naux2* = 20000.
If $n > 8192$, use *naux2* = 20000+1.14*n*.

For the transposed case, where *inc2x* = 1 or *inc2y* = 1, and where $n \geq 252$, add the following to the above storage requirements:

(*n*+256)(min(64, *m*))

Processor-Independent Formulas for DCFT for NAUX1 and NAUX2:

NAUX1 Formulas

For 32-bit integer arguments:

If $n \leq 2048$, use *naux1* = 20000.
If $n > 2048$, use *naux1* = 20000+2.28*n*.

For 64-bit integer arguments:

If $n \leq 2048$, use *naux1* = 30000.
If $n > 2048$, use *naux1* = 30000+2.28*n*.

NAUX2 Formulas

If $n \leq 2048$, use *naux2* = 20000.
If $n > 2048$, use *naux2* = 20000+2.28*n*.

For the transposed case, where *inc2x* = 1 or *inc2y* = 1, and where $n \geq 252$, add the following to the above storage requirements:

(2*n*+256)(min(64, *m*))

Function

The set of *m* complex discrete n-point Fourier transforms of complex data in array X, with results going into array Y, is expressed as follows:

$$y_{ki} = scale \sum_{j=0}^{n-1} x_{ji} W_n^{(Isign)jk}$$

for:

$k = 0, 1, \ldots, n-1$
$i = 1, 2, \ldots, m$

where:

$$W_n = e^{-2\pi\left(\sqrt{-1}\right)/n}$$

and where:

x_{ji} are elements of the sequences in array X.
y_{ki} are elements of the sequences in array Y.
Isign is + or - (determined by argument *isign*).
scale is a scalar value.

For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/*n* and *isign* being negative. See references [“1” on page 1253], [“4” on page 1253], [“5” on page 1253], [“27” on page 1254], and [“28” on page 1254].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transforms.

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors
None

Input-Argument Errors

1. *n* > 37748736
2. *inc1x*, *inc2x*, *inc1y*, or *inc2y* ≤ 0
3. *m* ≤ 0
4. *isign* = 0
5. *scale* = 0.0
6. The subroutine has not been initialized with the present arguments.
7. The length of the transform in *n* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
8. *naux1* ≤ 7
9. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
10. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows an input array X with a set of four short-precision complex sequences:

$$e^{2\pi\left(\sqrt{-1}\right)jk/n}$$

for *j* = 0, 1, ..., *n*-1 with *n* = 8, and the single frequencies *k* = 0, 1, 2, and 3. The arrays are declared as follows:

```
      COMPLEX*8  X(0:1023),Y(0:1023)
      REAL*8     AUX1(1693),AUX2(1)
```

First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
          INIT  X INC1X INC2X Y INC1Y INC2Y N   M ISIGN SCALE  AUX1   NAUX1  AUX2   NAUX2
           |    |   |     |   |   |     |   |   |   |     |     |       |     |       |
CALL SCFT(INIT, X , 1  ,  8 , Y , 1  ,  8 , 8 , 4 , 1 , SCALE, AUX1 , 1693 , AUX2 ,   0)
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`SCALE = 1.0`

X contains the following four sequences:

```
(1.0000, 0.0000)  ( 1.0000,  0.0000)  ( 1.0000,  0.0000)  ( 1.0000,  0.0000)
(1.0000, 0.0000)  ( 0.7071,  0.7071)  ( 0.0000,  1.0000)  (-0.7071,  0.7071)
(1.0000, 0.0000)  ( 0.0000,  1.0000)  (-1.0000,  0.0000)  ( 0.0000, -1.0000)
(1.0000, 0.0000)  (-0.7071,  0.7071)  ( 0.0000, -1.0000)  ( 0.7071,  0.7071)
(1.0000, 0.0000)  (-1.0000,  0.0000)  ( 1.0000,  0.0000)  (-1.0000,  0.0000)
(1.0000, 0.0000)  (-0.7071, -0.7071)  ( 0.0000,  1.0000)  ( 0.7071, -0.7071)
(1.0000, 0.0000)  ( 0.0000, -1.0000)  (-1.0000,  0.0000)  ( 0.0000,  1.0000)
(1.0000, 0.0000)  ( 0.7071, -0.7071)  ( 0.0000, -1.0000)  (-0.7071, -0.7071)
```

Output:

Y contains the following four sequences:

```
(8.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (8.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (8.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (8.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
```

Example 2

This example shows an input array X with a set of four input spike sequences equal to the output of Example 1. This shows how you can compute the inverse of the transform in Example 1 by using a negative *isign*, giving as output the four sequences listed in the input for Example 1. First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
          INIT  X INC1X INC2X Y INC1Y INC2Y N   M  ISIGN SCALE   AUX1   NAUX1  AUX2   NAUX2
           |    |   |     |   |   |     |   |   |    |     |      |       |     |       |
CALL SCFT(INIT, X , 1  ,  8 , Y , 1  ,  8 , 8 , 4 , -1 , SCALE , AUX1 , 1693 , AUX2 ,   0)
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`SCALE = 0.125`
X = (same as output Y in Example 1)

Output:

Y =(same as input X in Example 1)

Example 3

This example shows an input array X with a set of four short-precision complex sequences

$$e^{2\pi\left(\sqrt{-1}\right)jk/n}$$

for *j* = 0, 1, ..., *n*-1 with *n* = 12, and the single frequencies *k* = 0, 1, 2, and 3. Also, *inc1x* = *inc1y* = *m* and *inc2x* = *inc2y* = 1 to show how the input and output arrays can be stored in the transposed form. The arrays are declared as follows:

```
      COMPLEX*8  X (4,0:11),Y(4,0:11)
      REAL*8     AUX1(10000),AUX2(1)
```

First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
          INIT   X INC1X INC2X Y INC1Y INC2Y  N   M ISIGN SCALE  AUX1   NAUX1   AUX2   NAUX2
           |     |   |     |   |   |     |    |   |   |     |     |       |      |       |
CALL SCFT(INIT, X , 4  ,  1 , Y , 4  ,  1 , 12 , 4 , 1 , SCALE, AUX1 , 10000 , AUX2 ,   0)
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`SCALE = 1.0`

X contains the following four sequences:

```
(1.0000, 0.0000)  ( 1.0000,  0.0000)  ( 1.0000,  0.0000)  ( 1.0000,  0.0000)
(1.0000, 0.0000)  ( 0.8660,  0.5000)  ( 0.5000,  0.8660)  ( 0.0000,  1.0000)
(1.0000, 0.0000)  ( 0.5000,  0.8660)  (-0.5000,  0.8660)  (-1.0000,  0.0000)
(1.0000, 0.0000)  ( 0.0000,  1.0000)  (-1.0000,  0.0000)  ( 0.0000, -1.0000)
(1.0000, 0.0000)  (-0.5000,  0.8660)  (-0.5000, -0.8660)  ( 1.0000,  0.0000)
(1.0000, 0.0000)  (-0.8660,  0.5000)  ( 0.5000, -0.8660)  ( 0.0000,  1.0000)
(1.0000, 0.0000)  (-1.0000,  0.0000)  ( 1.0000,  0.0000)  (-1.0000,  0.0000)
(1.0000, 0.0000)  (-0.8660, -0.5000)  ( 0.5000,  0.8660)  ( 0.0000, -1.0000)
(1.0000, 0.0000)  (-0.5000, -0.8660)  (-0.5000,  0.8660)  ( 1.0000,  0.0000)
(1.0000, 0.0000)  ( 0.0000, -1.0000)  (-1.0000,  0.0000)  ( 0.0000,  1.0000)
(1.0000, 0.0000)  ( 0.5000, -0.8660)  (-0.5000, -0.8660)  (-1.0000,  0.0000)
(1.0000, 0.0000)  ( 0.8660, -0.5000)  ( 0.5000, -0.8660)  ( 0.0000, -1.0000)
```

Output:

Y contains the following four sequences:

```
(12.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)
( 0.0000, 0.0000)  (12.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)
( 0.0000, 0.0000)  ( 0.0000, 0.0000)  (12.0000, 0.0000)  ( 0.0000, 0.0000)
( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)  (12.0000, 0.0000)
( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)
( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)
( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)
( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)
( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)
( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)
( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)
( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)  ( 0.0000, 0.0000)
```

Example 4

This example shows an input array X with a set of four input spike sequences exactly equal to the output of Example 3. This shows how you can compute the inverse of the transform in Example 3 by using a negative *isign*, giving as output the four sequences listed in the input for Example 3. First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
          INIT   X INC1X INC2X Y INC1Y INC2Y  N   M  ISIGN SCALE   AUX1  NAUX1  AUX2  NAUX2
           |     |   |     |   |   |     |    |   |    |     |      |      |     |      |
CALL SCFT(INIT, X , 4  ,  1 , Y , 4  ,  1 , 12 , 4 , -1 , SCALE , AUX1, 10000, AUX2,   0)
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`SCALE = 1.0/12.0`
X = (same as output Y in Example 3)

Output:

Y =(same as input X in Example 3)

Example 5

This example shows how to compute a transform of a single long-precision complex sequence. It uses *isign* = 1 and *scale* = 1.0. The arrays are declared as follows:

```
     COMPLEX*16  X(0:7),Y(0:7)
     REAL*8      AUX1(26),AUX2(1)
```

The input in X is an impulse at zero, and the output in Y is constant for all frequencies. First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
          INIT  X INC1X INC2X Y INC1Y INC2Y N   M ISIGN SCALE   AUX1  NAUX1  AUX2  NAUX2
           |    |   |     |   |   |     |   |   |   |     |      |      |     |      |
CALL DCFT(INIT, X , 1  ,  0 , Y , 1  ,  0 , 8 , 1 , 1 , SCALE , AUX1 , 26  , AUX2 ,  0)
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`SCALE = 1.0`

X contains the following sequence:

```
(1.0000, 0.0000)
(0.0000, 0.0000)
(0.0000, 0.0000)
(0.0000, 0.0000)
(0.0000, 0.0000)
(0.0000, 0.0000)
(0.0000, 0.0000)
(0.0000, 0.0000)
```

Output:

```
(1.0000, 0.0000)
(1.0000, 0.0000)
(1.0000, 0.0000)
(1.0000, 0.0000)
(1.0000, 0.0000)
(1.0000, 0.0000)
(1.0000, 0.0000)
(1.0000, 0.0000)
```

SRCFT and DRCFT (Real-to-Complex Fourier Transform)

Purpose

These subroutines compute a set of *m* complex discrete *n*-point Fourier transforms of real data.

Table 223: Data Types.

Data Types		
X, *scale*	Y	**Subroutine**

Table 223: Data Types. (continued)		
Data Types		
Short-precision real	Short-precision complex	SRCFT
Long-precision real	Long-precision complex	DRCFT

Note:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SRCFT (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*) CALL DRCFT (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)

C and C++	srcft (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*);
	
	drcft (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init
is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transforms of the given sequences are computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: an integer. It can have any value.

x
is the array X, consisting of *m* sequences of length *n*, which are to be transformed. The sequences are assumed to be stored with stride 1.

Specified as: an array of (at least) length *n*+(*m*-1)*inc2x*, containing numbers of the data type indicated in Table 223 on page 993. See "Notes" on page 996 for more details. (It can be declared as X(*inc2x*,*m*).)

inc2x
is the stride between the first elements of the sequences in array X. (If *m* = 1, this argument is ignored.) Specified as: an integer; *inc2x* ≥ *n*.

y
See On Return.

inc2y
is the stride between the first elements of the sequences in array Y. (If *m* = 1, this argument is ignored.) Specified as: an integer; *inc2y* ≥ (*n*/2)+1.

n
is the length of each sequence to be transformed.

Specified as: an integer; $n \leq 37748736$ and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see “Providing a Correct Transform Length to ESSL” on page 56.

m
is the number of sequences to be transformed.

Specified as: an integer; $m > 0$.

isign
controls the direction of the transform, determining the sign *Isign* of the exponent of W_n, where:

If *isign* = positive value, *Isign* = + (transforming time to frequency).

If *isign* = negative value, *Isign* = - (transforming frequency to time).

Specified as: an integer; *isign* > 0 or *isign* < 0.

scale
is the scaling constant *scale*. See “Function” on page 997 for its usage.

Specified as: a number of the data type indicated in Table 223 on page 993, where *scale* > 0.0 or *scale* < 0.0.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, the working storage is computed.

If *init* = 0, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*.

Specified as: an integer; *naux1* > 14 (32-bit integer arguments) or 27 (64-bit integer arguments) and *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 14 (32-bit integer arguments) or 27 (64-bit integer arguments) and the minimum value, you have the option of having the minimum value returned in this argument. For details, see “Using Auxiliary Storage in ESSL” on page 49.

aux2
has the following meaning:

If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*.

Specified as: an integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, SRCFT and DRCFT dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see “Using Auxiliary Storage in ESSL” on page 49.

aux3
this argument is provided for migration purposes only and is ignored.

Specified as: an area of storage, containing *naux3* long-precision real numbers.

naux3
this argument is provided for migration purposes only and is ignored.

Specified as: an integer.

On Return

y
has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of the results of the *m* complex discrete Fourier transforms, each of length *n*. The sequences are stored with the stride 1. Due to complex conjugate symmetry, only the first (n/2) + 1 elements of each sequence are given in the output—that is, y_{ki}, $k = 0, 1, ..., n/2$, $i = 1, 2, ..., m$.

Returned as: an array of (at least) length *n*/2+1+(*m*-1)*inc2y*, containing numbers of the data type indicated in Table 223 on page 993. This array can be declared as Y(*inc2y*,*m*).

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* ≠ 0 and *init* = 0.
3. In these subroutines, the elements in each sequence in ***x*** and ***y*** are assumed to be stored in contiguous storage locations, using a stride of 1; therefore, *inc1x* and *inc1y* values are not a part of the argument list. For optimal performance, the *inc2x* and *inc2y* values should be close to their respective minimum values, which are given below:

   min(*inc2x*) = *n*
   min(*inc2y*) = *n*/2+1

   If you specify the same array for X and Y, then *inc2x* must equal 2(*inc2y*). In this case, output overwrites input. If *m* = 1, the *inc2x* and *inc2y* values are not used by the subroutine. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Formulas

Processor-Independent Formulas for SRCFT for NAUX1 and NAUX2:

NAUX1 Formulas

For 32-bit integer arguments:

If $n \le 16384$, use *naux1* = 25000.
If $n > 16384$, use *naux1* = 20000+0.82*n*.

For 64-bit integer arguments:

If $n \le 16384$, use *naux1* = 35000.
If $n > 16384$, use *naux1* = 30000+0.82n.

NAUX2 Formulas

If $n \le 16384$, use *naux2* = 20000.
If $n > 16384$, use *naux2* = 20000+0.57n.

Processor-Independent Formulas for DRCFT for NAUX1 and NAUX2:

NAUX1 Formulas

For 32-bit integer arguments:

If $n \le 4096$, use *naux1* = 22000.
If $n > 4096$, use *naux1* = 20000+1.64n.

For 64-bit integer arguments:

If $n \le 4096$, use *naux1* = 32000.
If $n > 4096$, use *naux1* = 30000+1.64n.

NAUX2 Formulas

If $n \le 4096$, use *naux2* = 20000.
If $n > 4096$, use *naux2* = 20000+1.14n.

Function

The set of *m* complex conjugate even discrete n-point Fourier transforms of real data in array X, with results going into array Y, is expressed as follows:

$$y_{ki} = scale \sum_{j=0}^{n-1} x_{ji} W_n^{(Isign)jk}$$

for:

$k = 0, 1, \ldots, n-1$
$i = 1, 2, \ldots, m$

where:

$$W_n = e^{-2\pi\left(\sqrt{-1}\right)/n}$$

and where:

x_{ji} are elements of the sequences in array X.
y_{ki} are elements of the sequences in array Y.
Isign is + or - (determined by argument *isign*).
scale is a scalar value.

The output in array Y is complex. For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/*n* and *isign* being negative. See references [“1” on page 1253], [“5” on page 1253], [“27” on page 1254], and [“28” on page 1254].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transforms.

Error conditions

Resource Errors

Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors

None

Input-Argument Errors

1. *n* > 37748736
2. *m* ≤ 0
3. *inc2x* < *n*
4. *inc2y* < *n*/2+1
5. *isign* = 0
6. *scale* = 0.0
7. The subroutine has not been initialized with the present arguments.
8. The length of the transform in *n* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
9. *naux1* ≤ 14
10. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
11. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows an input array X with a set of *m* cosine sequences cos(2π*jk*/*n*), *j* = 0, 1, ..., 15 with the single frequencies *k* = 0, 1, 2, 3. The Fourier transform of the cosine sequence with frequency *k* = 0 or n/2 has 1.0 in the 0 or n/2 position, respectively, and zeros elsewhere. For all other *k*, the Fourier transform has 0.5 in the *k* position and zeros elsewhere. The arrays are declared as follows:

```
     REAL*4     X(0:65535)
     COMPLEX*8  Y(0:32768)
     REAL*8     AUX1(41928), AUX2(1), AUX3(1)
```

First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT   X   INC2X Y INC2Y  N   M ISIGN SCALE  AUX1   NAUX1   AUX2   NAUX2   AUX3
NAUX3
            |     |    |    |   |    |   |   |     |     |       |      |       |      |
|
CALL SRCFT(INIT, X , 16 , Y , 9 , 16 , 4 , 1 , SCALE, AUX1 , 41928 , AUX2 ,   0   , AUX3 ,
0 )
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`SCALE = 1.0/16`

X contains the following four sequences:

```
1.0000   1.0000   1.0000   1.0000
1.0000   0.9239   0.7071   0.3827
1.0000   0.7071   0.0000  -0.7071
1.0000   0.3827  -0.7071  -0.9239
1.0000   0.0000  -1.0000   0.0000
1.0000  -0.3827  -0.7071   0.9239
1.0000  -0.7071   0.0000   0.7071
1.0000  -0.9239   0.7071  -0.3827
1.0000  -1.0000   1.0000  -1.0000
1.0000  -0.9239   0.7071  -0.3827
1.0000  -0.7071   0.0000   0.7071
1.0000  -0.3827  -0.7071   0.9239
1.0000   0.0000  -1.0000   0.0000
1.0000   0.3827  -0.7071  -0.9239
1.0000   0.7071   0.0000  -0.7071
1.0000   0.9239   0.7071   0.3827
```

Output:

Y contains the following four sequences:

```
(1.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.5000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.5000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.5000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
```

Example 2

This example shows another transform computation with different data using the same initialized array AUX1 as in Example 1. The input is also a set of four cosine sequences cos(2πjk/n), j = 0, 1, ..., 15 with the single frequencies k = 8, 9, 10, 11, thus including the middle frequency k = 8. The middle frequency has the value 1.0. For other frequencies, the transform has zeros, except for frequencies k and n-k. Only the values for j = n-k are given in the output.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT X   INC2X Y INC2Y   N   M ISIGN SCALE   AUX1   NAUX1   AUX2   NAUX2   AUX3
NAUX3
             |   |    |   |   |     |   |   |      |     |       |      |       |      |
CALL SRCFT( 0 , X , 16 , Y , 9 , 16 , 4 , 1 , SCALE, AUX1 , 41928 , AUX2 ,   0   , AUX3 ,
0  )
```

```
SCALE    =  1.0/16
```

X contains the following four sequences:

```
 1.0000   1.0000   1.0000   1.0000
-1.0000  -0.9239  -0.7071  -0.3827
 1.0000   0.7071   0.0000  -0.7071
-1.0000  -0.3827   0.7071   0.9239
 1.0000   0.0000  -1.0000   0.0000
-1.0000   0.3827   0.7071  -0.9239
 1.0000  -0.7071   0.0000   0.7071
-1.0000   0.9239  -0.7071   0.3827
 1.0000  -1.0000   1.0000  -1.0000
-1.0000   0.9239  -0.7071   0.3827
 1.0000  -0.7071   0.0000   0.7071
-1.0000   0.3827   0.7071  -0.9239
 1.0000   0.0000  -1.0000   0.0000
-1.0000  -0.3827   0.7071   0.9239
```

```
 1.0000   0.7071   0.0000  -0.7071
-1.0000  -0.9239  -0.7071  -0.3827
```

Output:

Y contains the following four sequences:

```
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.5000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.5000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.5000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(1.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
```

Example 3

This example uses the mixed-radix capability. The arrays are declared as follows:

```
      REAL*8     X(0:11)
      COMPLEX*16 Y(0:6)
      REAL*8     AUX1(50),AUX2(1)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```
      EQUIVALENCE (X,Y)
```

First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT   X INC2X Y INC2Y  N    M ISIGN SCALE    AUX1  NAUX1 AUX2  NAUX2
             |     |   |   |   |    |    |   |      |       |      |    |      |
CALL DRCFT(INIT, X , 0 , Y , 0 , 12 , 1 , 1 , SCALE , AUX1 , 50 , AUX2 ,  0)
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`SCALE = 1.0`
`X = (1.0000 , 1.0000 , 1.0000 , 1.0000 , 1.0000 , 1.0000 ,`
`1.0000 , 1.0000 , 1.0000 , 1.0000 , 1.0000 , 1.0000)`

Output:

Y contains the following sequence:

```
(12.0000 , 0.0000)
 (0.0000 , 0.0000)
 (0.0000 , 0.0000)
 (0.0000 , 0.0000)
 (0.0000 , 0.0000)
 (0.0000 , 0.0000)
 (0.0000 , 0.0000)
```

SCRFT and DCRFT (Complex-to-Real Fourier Transform)

Purpose

These subroutines compute a set of *m* real discrete *n*-point Fourier transforms of complex conjugate even data.

Table 224: Data Types.		
Data Types		
X	Y, *scale*	**Subroutine**
Short-precision complex	Short-precision real	SCRFT
Long-precision complex	Long-precision real	DCRFT

Note:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	**Syntax**
Fortran	CALL SCRFT (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*) CALL DCRFT (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	scrft (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*); dcrft (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init
is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transforms of the given sequences are computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: an integer. It can have any value.

x
is the array X, consisting of *m* sequences. Due to complex conjugate symmetry, the input consists of only the first (*n*/2)+1 elements of each sequence; that is, x_{ji}, *j* = 0, 1, ..., *n*/2, *i* = 1, 2, ..., *m*. The sequences are assumed to be stored with stride 1.

Specified as: an array of (at least) length *n*/2+1+(*m*-1)*inc2x*, containing numbers of the data type indicated in Table 224 on page 1001. This array can be declared as X(*inc2x*,*m*).

inc2x
is the stride between the first elements of the sequences in array X. (If *m* = 1, this argument is ignored.) Specified as: an integer; *inc2x* ≥ (*n*/2)+1.

y
See On Return.

inc2y
is the stride between the first elements of the sequences in array Y. (If *m* = 1, this argument is ignored.) Specified as: an integer; *inc2y* ≥ *n*.

n
is the length of each sequence to be transformed.

Specified as: an integer; $n \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

m
: is the number of sequences to be transformed.

  Specified as: an integer; $m > 0$.

isign
: controls the direction of the transform, determining the sign *Isign* of the exponent of W_n, where:

  If *isign* = positive value, *Isign* = + (transforming time to frequency).

  If *isign* = negative value, *Isign* = - (transforming frequency to time).

  Specified as: an integer; *isign* > 0 or *isign* < 0.

scale
: is the scaling constant *scale*. See "Function" on page 1004 for its usage.

  Specified as: a number of the data type indicated in Table 224 on page 1001, where *scale* > 0.0 or *scale* < 0.0.

aux1
: is the working storage for this subroutine, where:

  If *init* ≠ 0, the working storage is computed.

  If *init* = 0, the working storage is used in the computation of the Fourier transforms.

  Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
: is the number of doublewords in the working storage specified in *aux1*.

  Specified as: an integer; *naux1* > 13 (32-bit integer arguments) or 25 (64-bit integer arguments) and *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 13 (32-bit integer arguments) or 25 (64-bit integer arguments) and the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

aux2
: has the following meaning:

  If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

  Otherwise, it is the working storage used by this subroutine that is available for use by the calling program between calls to this subroutine.

  Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
: is the number of doublewords in the working storage specified in *aux2*.

  Specified as: an integer, where:

  If *naux2* = 0 and error 2015 is unrecoverable, SCRFT and DCRFT dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

  Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

aux3
: this argument is provided for migration purposes only and is ignored.

Specified as: an area of storage, containing *naux3* long-precision real numbers.

naux3
this argument is provided for migration purposes only and is ignored.

Specified as: an integer.

On Return

y
has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of the results of the *m* discrete Fourier transforms of the complex conjugate even data, each of length *n*. The sequences are stored with stride 1.

Returned as: an array of (at least) length *n*+(*m*-1)*inc2y*, containing numbers of the data type indicated in Table 224 on page 1001. See “Notes” on page 1003 for more details. (It can be declared as Y(*inc2y*,*m*).)

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* ≠ 0 and *init* = 0.
3. The elements in each sequence in ***x*** and ***y*** are assumed to be stored in contiguous storage locations—that is, with a stride of 1. Therefore, *inc1x* and *inc1y* values are not a part of the argument list. For optimal performance, the *inc2x* and *inc2y* values should be close to their respective minimum values, which are given below:

   min(*inc2y*) = *n*
   min(*inc2x*) = *n*/2+1

   If you specify the same array for X and Y, then *inc2y* must equal 2(*inc2x*). In this case, output overwrites input. If *m* = 1, the *inc2x* and *inc2y* values are not used by the subroutine. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.

Formulas

Processor-Independent Formulas for SCRFT for NAUX1 and NAUX2:

NAUX1 Formulas

For 32-bit integer arguments:

If *n* ≤ 16384, use *naux1* = 25000.
If *n* > 16384, use *naux1* = 20000+0.82*n*.

For 64-bit integer arguments:

If *n* ≤ 16384, use *naux1* = 35000.
If *n* > 16384, use *naux1* = 30000+0.82*n*.

NAUX2 Formulas

If $n \le 16384$, use *naux2* = 20000.
If $n > 16384$, use *naux2* = 20000+0.57*n*.

Processor-Independent Formulas for DCRFT for NAUX1 and NAUX2:

NAUX1 Formulas

For 32-bit integer arguments:

If $n \le 4096$, use *naux1* = 22000.
If $n > 4096$, use *naux1* = 20000+1.64*n*.

For 64-bit integer arguments:

If $n \le 4096$, use *naux1* = 32000.
If $n > 4096$, use *naux1* = 30000+1.64*n*.

NAUX2 Formulas

If $n \le 4096$, use *naux2* = 20000.
If $n > 4096$, use *naux2* = 20000+1.14*n*.

Function

The set of *m* real discrete *n*-point Fourier transforms of complex conjugate even data in array X, with results going into array Y, is expressed as follows:

$$y_{ki} = scale \sum_{j=0}^{n-1} x_{ji} W_n^{(Isign)jk}$$

for:

$k = 0, 1, ..., n\text{-}1$
$i = 1, 2, ..., m$

where:

$$W_n = e^{-2\pi\left(\sqrt{-1}\right)/n}$$

and where:

x_{ji} are elements of the sequences in array X.
y_{ki} are elements of the sequences in array Y.
Isign is + or - (determined by argument *isign*).
scale is a scalar value.

Because of the symmetry, Y has real data. For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/*n* and *isign* being negative. See references [“1” on page 1253], [“5” on page 1253], [“27” on page 1254], and [“28” on page 1254].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.

2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transforms.

Error conditions

Resource Errors

Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors

None

Input-Argument Errors

1. *n* > 37748736
2. *m* ≤ 0
3. *inc2x* < *n*/2+1
4. *inc2y* < *n*
5. *scale* = 0.0
6. *isign* = 0
7. The subroutine has not been initialized with the present arguments.
8. The length of the transform in *n* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
9. *naux1* ≤ 13
10. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
11. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example uses the mixed-radix capability and shows how to compute a single transform. The arrays are declared as follows:

```
     COMPLEX*8  X(0:6)
     REAL*8     AUX1(50), AUX2(1), AUX3(1)
     REAL*4     Y(0:11)
```

First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note:

1. X shows the *n*/2+1 = 7 elements used in the computation.
2. Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT   X INC2X Y INC2Y   N   M ISIGN SCALE  AUX1  NAUX1 AUX2  NAUX2  AUX3  NAUX3
            |     |   |   |   |     |   |   |     |     |     |     |     |     |     |
CALL SCRFT(INIT, X , 0 , Y , 0 , 12 , 1 , 1 , SCALE, AUX1 , 50 , AUX2 ,  0 ,  AUX3 ,  0  )
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`SCALE = 1.0`

X contains the following sequence:

```
(1.0,  0.0)
(0.0,  0.0)
```

```
(0.0,  0.0)
(0.0,  0.0)
(0.0,  0.0)
(0.0,  0.0)
(0.0,  0.0)
```

Output:

```
Y        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

Example 2

This example shows another transform computation with different data using the same initialized array AUX1 as in Example 1.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
          INIT  X INC2X Y INC2Y  N   M ISIGN SCALE  AUX1 NAUX1  AUX2 NAUX2  AUX3  NAUX3
           |    |   |   |   |    |   |   |     |     |     |     |     |     |      |
CALL SCRFT( 0 , X , 0 , Y , 0 , 12 , 1 , 1 , SCALE, AUX1 , 50 , AUX2 , 0 ,  AUX3 ,  0 )
```

```
SCALE    =  1.0
```

X contains the following sequence:

```
(1.0, 0.0)
(1.0, 0.0)
(1.0, 0.0)
(1.0, 0.0)
(1.0, 0.0)
(1.0, 0.0)
(1.0, 0.0)
```

Output:

```
Y        =  (12.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 ,
             0.0 , 0.0 , 0.0 , 0.0)
```

Example 3

This example shows how to compute many transforms simultaneously. The arrays are declared as follows:

```
     COMPLEX*8  X(0:8,2)
     REAL*8     AUX1(50), AUX2(1), AUX3(1)
     REAL*4     Y(0:15,2)
```

First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT  X INC2X Y INC2Y   N   M ISIGN SCALE  AUX1  NAUX1 AUX2  NAUX2 AUX3  NAUX3
            |    |   |   |   |     |   |   |     |     |      |    |      |    |      |
CALL SCRFT(INIT, X , 9 , Y , 16 , 16 , 2 , 1 , SCALE, AUX1 , 50 , AUX2 ,  0 , AUX3 ,  0 )
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`SCALE = 1.0`

X contains the following two sequences:

```
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (0.0, 0.0)
```

```
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (1.0, 0.0)
```

Output:

Y contains the following two sequences:

```
16.0   1.0
 0.0  -1.0
 0.0   1.0
 0.0  -1.0
 0.0   1.0
 0.0  -1.0
 0.0   1.0
 0.0  -1.0
 0.0   1.0
 0.0  -1.0
 0.0   1.0
 0.0  -1.0
 0.0   1.0
 0.0  -1.0
 0.0   1.0
 0.0  -1.0
```

Example 4

This example shows the same array being used for input and output. The arrays are declared as follows:

```
      COMPLEX*16 X(0:8,2)
      REAL*8     AUX1(50), AUX2(1)
      REAL*8     Y(0:17,2)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```
      EQUIVALENCE (X,Y)
```

This requires INC2Y = 2(INC2X). First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT  X INC2X Y  INC2Y  N   M  ISIGN SCALE  AUX1  NAUX1 AUX2  NAUX2
             |    |   |   |    |    |   |    |     |     |      |    |      |
CALL DCRFT(INIT, X , 9 , Y , 18 , 16 , 2 , -1 , SCALE, AUX1 , 50 , AUX2 ,  0)
```

INIT = 1 (for initialization)
INIT = 0 (for computation)
SCALE = 0.0625

X contains the following two sequences:

```
( 1.0,  0.0)  ( 1.0,  0.0)
( 0.0,  1.0)  ( 0.0, -1.0)
(-1.0,  0.0)  (-1.0,  0.0)
( 0.0, -1.0)  ( 0.0,  1.0)
( 1.0,  0.0)  ( 1.0,  0.0)
( 0.0,  1.0)  ( 0.0, -1.0)
(-1.0,  0.0)  (-1.0,  0.0)
( 0.0, -1.0)  ( 0.0,  1.0)
( 1.0,  0.0)  ( 1.0,  0.0)
```

Output:

Y contains the following two sequences:

```
0.0  0.0
0.0  0.0
0.0  0.0
0.0  0.0
0.0  1.0
0.0  0.0
0.0  0.0
0.0  0.0
0.0  0.0
0.0  0.0
0.0  0.0
0.0  0.0
1.0  0.0
0.0  0.0
0.0  0.0
0.0  0.0
```

SCOSF and DCOSF (Cosine Transform)

Purpose

These subroutines compute a set of *m* real even discrete *n*-point Fourier transforms of cosine sequences of real even data.

Table 225: Data Types.

Data Types	
X, Y, *scale*	**Subroutine**
Short-precision real	SCOSF
Long-precision real	DCOSF

Note:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	**Syntax**
Fortran	CALL SCOSF | DCOSF (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	scosf | dcosf (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init

is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transforms of the given sequences are computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: an integer. It can have any value.

x
: is the array X, consisting of *m* sequences of length *n*/2+1.

  Specified as: an array of (at least) length 1+(*n*/2)*inc1x*+(*m*-1)*inc2x*, containing numbers of the data type indicated in Table 225 on page 1008.

inc1x
: is the stride between the elements within each sequence in array X.

  Specified as: an integer; *inc1x* > 0.

inc2x
: is the stride between the first elements of the sequences in array X. (If *m* = 1, this argument is ignored.) Specified as: an integer; *inc2x* > 0.

y
: See On Return.

inc1y
: is the stride between the elements within each sequence in array Y.

  Specified as: an integer; *inc1y* > 0.

inc2y
: is the stride between the first elements of the sequences in array Y. (If *m* = 1, this argument is ignored.) Specified as: an integer; *inc2y* > 0.

n
: is the transform length. However, due to symmetry, only the first *n*/2+1 values are given in the input and output.

  Specified as: an integer; *n* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

m
: is the number of sequences to be transformed.

  Specified as: an integer; *m* > 0.

scale
: is the scaling constant *scale*. See "Function" on page 1011 for its usage.

  Specified as: a number of the data type indicated in Table 225 on page 1008, where *scale* > 0.0 or *scale* < 0.0.

aux1
: is the working storage for this subroutine, where:

  If *init* ≠ 0, the working storage is computed.

  If *init* = 0, the working storage is used in the computation of the Fourier transforms.

  Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
: is the number of doublewords in the working storage specified in *aux1*.

  Specified as: an integer; *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

aux2
: has the following meaning:

  If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2

is the number of doublewords in the working storage specified in *aux2*.

Specified as: an integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, SCOSF and DCOSF dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

On Return

y

has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of the results of the *m* discrete Fourier transforms, where each Fourier transform is real and of length *n*. However, due to symmetry, only the first *n*/2+1 values are given in the output—that is, y_{ki}, k = 0, 1, ..., *n*/2 for each i = 1, 2, ..., *m*.

Returned as: an array of (at least) length 1+(*n*/2)*inc1y*+(*m*-1)*inc2y*, containing numbers of the data type indicated in Table 225 on page 1008.

aux1

is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* ≠ 0 and *init* = 0.
3. For optimal performance, the preferred value for *inc1x* and *inc1y* is 1. This implies that the sequences are stored with stride 1. In addition, *inc2x* and *inc2y* should be close to *n*/2+1.

   It is possible to specify sequences in the transposed form—that is, as rows of a two-dimensional array. In this case, *inc2x* (or *inc2y*) = 1 and *inc1x* (or *inc1y*) is equal to the leading dimension of the array. One can specify either input, output, or both in the transposed form by specifying appropriate values for the stride parameters. For selecting optimal values of *inc1x* and *inc1y* for _COSF, you should use "STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)" on page 1195. Example 2 in the STRIDE subroutine description explains how it is used for _COSF.

   If you specify the same array for X and Y, then *inc1x* and *inc1y* must be equal, and *inc2x* and *inc2y* must be equal. In this case, output overwrites input. If *m* = 1, the *inc2x* and *inc2y* values are not used by the subroutine. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Formulas

Processor-Independent Formulas for SCOSF for NAUX1 and NAUX2:

NAUX1 Formulas

For 32-bit integer arguments:

If $n \le 16384$, use *naux1* = 40000.
If $n > 16384$, use *naux1* = 20000+.30*n*.

For 64-bit integer arguments:

If $n \le 16384$, use *naux1* = 50000.
If $n > 16384$, use *naux1* = 30000+.30*n*.

NAUX2 Formulas

If $n \le 16384$, use *naux2* = 25000.
If $n > 16384$, use *naux2* = 20000+.32*n*.

For the transposed case, where *inc2x* = 1 or *inc2y* = 1, and where $n \ge 252$, add the following to the above storage requirements:

(*n*/4+257)(min(128, *m*))

Processor-Independent Formulas for DCOSF for NAUX1 and NAUX2:

NAUX1 Formulas

For 32-bit integer arguments:

If $n \le 16384$, use *naux1* = 35000.
If $n > 16384$, use *naux1* = 20000+.60*n*.

For 64-bit integer arguments:

If $n \le 16384$, use *naux1* = 45000.
If $n > 16384$, use *naux1* = 30000+.60*n*.

NAUX2 Formulas

If $n \le 16384$, use *naux2* = 20000.
If $n > 16384$, use *naux2* = 20000+.64*n*.

For the transposed case, where *inc2x* = 1 or *inc2y* = 1, and where $n \ge 252$, add the following to the above storage requirements:

(*n*/2+257)(min(128, *m*))

Function

The set of *m* real even discrete *n*-point Fourier transforms of the cosine sequences of real data in array X, with results going into array Y, is expressed as follows:

$$y_{ki} = scale\left(.5x_{0,i} + .5(-1)^k x_{n/2,i} + \sum_{j=1}^{n/2-1} x_{ji}\ \cos\left(jk\left(2\pi / n\right)\right)\right)$$

for:

$k = 0, 1, ..., n/2$
$i = 1, 2, ..., m$

where:

x_{ji} are elements of the sequences in array X, where each sequence contains the $n/2+1$ real nonredundant data x_{ji}, $j = 0, 1, ..., n/2$.

y_{ki} are elements of the sequences in array Y, where each sequence contains the $n/2+1$ real nonredundant data y_{ki}, $k = 0, 1, ..., n/2$.

scale is a scalar value.

You can obtain the inverse cosine transform by specifying *scale* = 4.0/*n*. Thus, if an X input is used with *scale* = 1.0, and its output is used as input on a subsequent call with *scale* = 4.0/*n*, the original X is obtained. See references [“1” on page 1253], [“5” on page 1253], [“27” on page 1254], and [“28” on page 1254].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transforms.

These subroutines use a Fourier transform method with a mixed-radix capability. This provides maximum performance for your application.

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors
None

Input-Argument Errors

1. $n > 37748736$
2. *inc1x* or *inc1y* ≤ 0
3. *inc2x* or *inc2y* ≤ 0
4. $m \le 0$
5. *scale* = 0.0
6. The subroutine has not been initialized with the present arguments.
7. The length of the transform in *n* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
8. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
9. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows an input array X with a set of *m* cosine sequences of length $n/2+1$, $\cos(jk(2\pi/n))$, $j = 0, 1, ..., n/2$, with the single frequencies $k = 0, 1, 2, 3$. The Fourier transform of the cosine sequence with frequency $k = 0$ or $n/2$ has $n/2$ in the 0-th or $n/2$-th position, respectively, and zeros elsewhere.

For all other *k*, the Fourier transform has *n*/4 in position *k* and zeros elsewhere. The arrays are declared as follows:

```
     REAL*4     X(0:71),Y(0:71)
     REAL*8     AUX1(414),AUX2(1)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT   X INC1X INC2X Y INC1Y INC2Y  N    M   SCALE  AUX1  NAUX1  AUX2  NAUX2
             |    |   |     |   |   |     |    |    |    |       |      |     |      |
CALL SCOSF(INIT, X , 1  , 18 , Y , 1  , 18 , 32 , 4 , SCALE, AUX1 , 414 , AUX2 ,  0)
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`SCALE = 1.0`

X contains the following four sequences:

```
1.0000   1.0000   1.0000   1.0000
1.0000   0.9808   0.9239   0.8315
1.0000   0.9239   0.7071   0.3827
1.0000   0.8315   0.3827  -0.1951
1.0000   0.7071   0.0000  -0.7071
1.0000   0.5556  -0.3827  -0.9808
1.0000   0.3827  -0.7071  -0.9239
1.0000   0.1951  -0.9239  -0.5556
1.0000   0.0000  -1.0000   0.0000
1.0000  -0.1951  -0.9239   0.5556
1.0000  -0.3827  -0.7071   0.9239
1.0000  -0.5556  -0.3827   0.9808
1.0000  -0.7071   0.0000   0.7071
1.0000  -0.8315   0.3827   0.1951
1.0000  -0.9239   0.7071  -0.3827
1.0000  -0.9808   0.9239  -0.8315
1.0000  -1.0000   1.0000  -1.0000
  .        .        .        .
```

Output:

Y contains the following four sequences:

```
16.0000  0.0000  0.0000  0.0000
 0.0000  8.0000  0.0000  0.0000
 0.0000  0.0000  8.0000  0.0000
 0.0000  0.0000  0.0000  8.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
   .       .       .       .
```

Example 2

This example shows an input array X with a set of four input spike sequences equal to the output of Example 1. This shows how you can compute the inverse of the transform in Example 1 by using *scale* = 4.0/*n*, giving as output the four sequences listed in the input for Example 1. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT  X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2
            |    |   |     |   |   |     |    |   |     |     |     |     |     |
CALL SCOSF(INIT, X , 1   , 18 , Y , 1   , 18 , 32 , 4 , SCALE, AUX1 , 414 , AUX2 ,  0)
```

INIT = 1 (for initialization)
INIT = 0 (for computation)
SCALE = 4.0/32
X = (same sequences as in output Y in Example 1)

Output:

Y =(same sequences as in output X in Example 1)

Example 3

This example shows another computation using the same arguments initialized in Example 1 and using different input sequence data. The data for this example has frequencies *k* = 14, 15, 16, 17. Because only the sequence data has changed, initialization does not have to be done again.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2
            |   |   |     |   |   |     |    |   |     |     |     |     |     |
CALL SCOSF( 0 , X , 1   , 18 , Y , 1   , 18 , 32 , 4 , SCALE, AUX1 , 414 , AUX2 ,  0)
```

```
SCALE    =  1.0
```

X contains the following four sequences:

```
 1.0000   1.0000   1.0000   1.0000
-0.9239  -0.9808  -1.0000  -0.9808
 0.7071   0.9239   1.0000   0.9239
-0.3827  -0.8315  -1.0000  -0.8315
 0.0000   0.7071   1.0000   0.7071
 0.3827  -0.5556  -1.0000  -0.5556
-0.7071   0.3827   1.0000   0.3827
 0.9239  -0.1951  -1.0000  -0.1951
-1.0000   0.0000   1.0000   0.0000
 0.9239   0.1951  -1.0000   0.1951
-0.7071  -0.3827   1.0000  -0.3827
 0.3827   0.5556  -1.0000   0.5556
 0.0000  -0.7071   1.0000  -0.7071
-0.3827   0.8315  -1.0000   0.8315
 0.7071  -0.9239   1.0000  -0.9239
-0.9239   0.9808  -1.0000   0.9808
 1.0000  -1.0000   1.0000  -1.0000
   .        .        .        .
```

Output:

Y contains the following four sequences:

```
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
8.0000  0.0000   0.0000  0.0000
```

```
 0.0000  8.0000   0.0000  8.0000
 0.0000  0.0000  16.0000  0.0000
  .       .        .       .
```

SSINF and DSINF (Sine Transform)

Purpose

These subroutines compute a set of *m* real even discrete *n*-point Fourier transforms of sine sequences of real even data.

Table 226: Data Types.

Data Types	
X, Y, *scale*	**Subroutine**
Short-precision real	SSINF
Long-precision real	DSINF

Note:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	**Syntax**
Fortran	CALL SSINF | DSINF (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	ssinf | dsinf (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init

is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transforms of the given sequences are computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: an integer. It can have any value.

x

is the array X, consisting of *m* sequences of length *n*/2.

Specified as: an array of (at least) length 1+(*n* / 2-1)*inc1x*+(*m*-1)*inc2x*, containing numbers of the data type indicated in Table 226 on page 1015. The first element in X must have a value of 0.0 (otherwise, incorrect results may occur).

inc1x

is the stride between the elements within each sequence in array X.

Specified as: an integer; *inc1x* > 0.

inc2x
is the stride between the first elements of the sequences in array X. (If *m* = 1, this argument is ignored.) Specified as: an integer; *inc2x* > 0.

y
See On Return.

inc1y
is the stride between the elements within each sequence in array Y.

Specified as: an integer; *inc1y* > 0.

inc2y
is the stride between the first elements of the sequences in array Y. (If *m* = 1, this argument is ignored.) Specified as: an integer; *inc2y* > 0.

n
is the transform length. However, due to symmetry, only the first *n*/2 values are given in the input and output.

Specified as: an integer; *n* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

m
is the number of sequences to be transformed.

Specified as: an integer; *m* > 0.

scale
is the scaling constant *scale*. See "Function" on page 1018 for its usage.

Specified as: a number of the data type indicated in Table 226 on page 1015, where *scale* > 0.0 or *scale* < 0.0.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, the working storage is computed.

If *init* = 0, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*.

Specified as: an integer; *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

aux2
has the following meaning:

If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*.

Specified as: an integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, SSINF and DSINF dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

On Return

y

has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of the results of the *m* discrete Fourier transforms, where each Fourier transform is real and of length *n*. However, due to symmetry, only the first *n*/2 values are given in the output—that is, y_{ki}, *k* = 0, 1, ..., *n*/2-1 for each *i* = 1, 2, ..., *m*.

Returned as: an array of (at least) length 1+(*n* / 2-1)*inc1y*+(*m*-1)*inc2y*, containing numbers of the data type indicated in Table 226 on page 1015.

aux1

is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* ≠ 0 and *init* = 0.
3. For optimal performance, the preferred value for *inc1x* and *inc1y* is 1. This implies that the sequences are stored with stride 1. In addition, *inc2x* and *inc2y* should be close to *n*/2.

   It is possible to specify sequences in the transposed form—that is, as rows of a two-dimensional array. In this case, *inc2x* (or *inc2y*) = 1 and *inc1x* (or *inc1y*) is equal to the leading dimension of the array. One can specify either input, output, or both in the transposed form by specifying appropriate values for the stride parameters. For selecting optimal values of *inc1x* and *inc1y* for _SINF, you should use "STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)" on page 1195. Example 3 in the STRIDE subroutine description explains how it is used for _SINF.

   If you specify the same array for X and Y, then *inc1x* and *inc1y* must be equal, and *inc2x* and *inc2y* must be equal. In this case, output overwrites input. If *m* = 1, the *inc2x* and *inc2y* values are not used by the subroutine. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Formulas

Processor-Independent Formulas for SSINF for NAUX1 and NAUX2:

NAUX1 Formulas

For 32-bit integer arguments:

If *n* ≤ 16384, use *naux1* = 60000.
If *n* > 16384, use *naux1* = 20000+.30*n*.

For 64-bit integer arguments:

If $n \le 16384$, use *naux1* = 70000.
If $n > 16384$, use *naux1* = 30000+.30n.

NAUX2 Formulas

If $n \le 16384$, use *naux2* = 25000.
If $n > 16384$, use *naux2* = 20000+.32n.

For the transposed case, where *inc2x* = 1 or *inc2y* = 1, and where $n \ge 252$, add the following to the above storage requirements:

$(n/4+257)(\min(128, m))$.

Processor-Independent Formulas for DSINF for NAUX1 and NAUX2:

NAUX1 Formulas

For 32-bit integer arguments:

If $n \le 16384$, use *naux1* = 50000.
If $n > 16384$, use *naux1* = 20000+.60n.

For 64-bit integer arguments:

If $n \le 16384$, use *naux1* = 60000.
If $n > 16384$, use *naux1* = 30000+.60n.

NAUX2 Formulas

If $n \le 16384$, use *naux2* = 20000.
If $n > 16384$, use *naux2* = 20000+.64n.

For the transposed case, where *inc2x* = 1 or *inc2y* = 1, and where $n \ge 252$, add the following to the above storage requirements:

$(n/2+257)(\min(128, m))$

Function

The set of *m* real even discrete *n*-point Fourier transforms of the sine sequences of real data in array X, with results going into array Y, is expressed as follows:

$$y_{ki} = scale \sum_{j=0}^{n/2-1} x_{ji} \sin(jk(2\pi / n))$$

for:

$k = 0, 1, ..., n/2-1$
$i = 1, 2, ..., m$

where:

$x_{0i} = 0.0$

x_{ji} are elements of the sequences in array X, where each sequence contains the $n/2$ real nonredundant data x_{ji}, $j = 0, 1, ..., n/2-1$.

y_{ki} are elements of the sequences in array Y, where each sequence contains the $n/2$ real nonredundant data y_{ki}, $k = 0, 1, ..., n/2-1$.

scale is a scalar value.

You can obtain the inverse sine transform by specifying *scale* = 4.0/*n*. Thus, if an X input is used with *scale* = 1.0, and its output is used as input on a subsequent call with *scale* = 4.0/*n*, the original X is obtained. See references [“1” on page 1253], [“5” on page 1253], [“27” on page 1254], and [“28” on page 1254].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transforms.

These subroutines use a Fourier transform method with a mixed-radix capability. This provides maximum performance for your application.

Error conditions

Resource Errors

Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors

None

Input-Argument Errors

1. $n > 37748736$
2. *inc1x* or *inc1y* ≤ 0
3. *inc2x* or *inc2y* ≤ 0
4. $m \le 0$
5. *scale* = 0.0
6. The subroutine has not been initialized with the present arguments.
7. The length of the transform in *n* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
8. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
9. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows an input array X with a set of *m* sine sequences of length $n/2$, $\sin(jk(2\pi/n))$, $j = 0, 1, ..., n/2-1$, with the single frequencies $k = 1, 2, 3$. The Fourier transform of the sine sequence has $n/4$ in position *k* and zeros elsewhere. The arrays are declared as follows:

```
      REAL*4     X(0:53),Y(0:53)
      REAL*8     AUX1(414),AUX2(1)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT   X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2   NAUX2
              |    |   |     |   |   |     |    |   |     |     |      |     |      |
CALL SSINF(INIT, X , 1  , 18 , Y , 1  , 18 , 32 , 3 , SCALE, AUX1 , 414 , AUX2 ,  0)
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`SCALE = 1.0`

X contains the following three sequences:

```
0.0000   0.0000   0.0000
0.1951   0.3827   0.5556
0.3827   0.7071   0.9239
0.5556   0.9239   0.9808
0.7071   1.0000   0.7071
0.8315   0.9239   0.1951
0.9239   0.7071  -0.3827
0.9808   0.3827  -0.8315
1.0000   0.0000  -1.0000
0.9808  -0.3827  -0.8315
0.9239  -0.7071  -0.3827
0.8315  -0.9239   0.1951
0.7071  -1.0000   0.7071
0.5556  -0.9239   0.9808
0.3827  -0.7071   0.9239
0.1951  -0.3827   0.5556
 .        .        .
 .        .        .
```

Output:

Y contains the following three sequences:

```
0.0000  0.0000  0.0000
8.0000  0.0000  0.0000
0.0000  8.0000  0.0000
0.0000  0.0000  8.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
 .       .       .
 .       .       .
```

Example 2

This example shows an input array X with a set of three input spike sequences equal to the output of Example 1. This shows how you can compute the inverse of the transform in Example 1 by using *scale* = 4.0/*n*, giving as output the three sequences listed in the input for Example 1. First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT   X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2
              |    |   |     |   |   |     |    |   |     |     |      |     |      |
CALL SSINF(INIT, X , 1  , 18 , Y , 1  , 18 , 32 , 3 , SCALE, AUX1 , 414 , AUX2 ,  0)
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)

```
SCALE = 4.0/32
```
X = (same sequences as in output Y in Example 1)

Output:

Y =(same sequences as in output X in Example 1)

Example 3

This example shows another computation using the same arguments initialized in Example 1 and using different input sequence data. The data for this example has frequencies *k* = 14, 15, 17. Because only the sequence data has changed, initialization does not have to be done again.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2
             |   |   |     |   |   |     |    |   |     |     |      |     |      |
CALL SSINF( 0 , X , 1  , 18 , Y , 1  , 18 , 32 , 3 , SCALE, AUX1 , 414 , AUX2 ,   0)
```

```
SCALE    =  1.0
```

X contains the following three sequences:

```
 0.0000   0.0000   0.0000
 0.3827   0.1951  -0.1951
-0.7071  -0.3827   0.3827
 0.9239   0.5556  -0.5556
-1.0000  -0.7071   0.7071
 0.9239   0.8315  -0.8315
-0.7071  -0.9239   0.9239
 0.3827   0.9808  -0.9808
 0.8573  -1.0000   1.0000
-0.3827   0.9808  -0.9808
 0.7071  -0.9239   0.9239
-0.9239   0.8315  -0.8315
 1.0000  -0.7071   0.7071
-0.9239   0.5556  -0.5556
 0.7071  -0.3827   0.3827
-0.3827   0.1951  -0.1951
   .        .        .
   .        .        .
```

Output:

Y contains the following three sequences:

```
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
8.0000  0.0000   0.0000
0.0000  8.0000  -8.0000
0.0000  0.0000   0.0000
  .       .        .
  .       .        .
```

SCFT2 and DCFT2 (Complex Fourier Transform in Two Dimensions)

Purpose

These subroutines compute the two-dimensional discrete Fourier transform of complex data.

Table 227: Data Types.		
Data Types		
X, Y	*scale*	**Subroutine**
Short-precision complex	Short-precision real	SCFT2
Long-precision complex	Long-precision real	DCFT2

Note:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	**Syntax**
Fortran	CALL SCFT2 | DCFT2 (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	scft2 | dcft2 (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init

is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transform of the given array is computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: an integer. It can have any value.

x

is the array X, containing the two-dimensional data to be transformed, where each element $x_{j1,j2}$, using zero-based indexing, is stored in X(*j1*(*inc1x*)+*j2*(*inc2x*)) for *j1* = 0, 1, ..., *n1*-1 and *j2* = 0, 1, ..., *n2*-1.

Specified as: an array of (at least) length 1+(*n1*-1)*inc1x*+(*n2*-1)*inc2x*, containing numbers of the data type indicated in Table 227 on page 1022.

If *inc1x* = 1, the input array is stored in normal form, and *inc2x* ≥ *n1*.

If *inc2x* = 1, the input array is stored in transposed form, and *inc1x* ≥ *n2*.

See "Notes" on page 1024 for more details.

inc1x

is the stride between the elements in array X for the first dimension.

If the array is stored in the normal form, *inc1x* = 1.

If the array is stored in the transposed form, *inc1x* is the leading dimension of the array and *inc1x* ≥ *n2*.

Specified as: an integer; *inc1x* > 0. If *inc2x* = 1, then *inc1x* ≥ *n2*.

inc2x

is the stride between the elements in array X for the second dimension.

If the array is stored in the transposed form, *inc2x* = 1.

If the array is stored in the normal form, *inc2x* is the leading dimension of the array and $inc2x \geq n1$.

Specified as: an integer; $inc2x > 0$. If $inc1x = 1$, then $inc2x \geq n1$.

y

See On Return.

inc1y

is the stride between the elements in array Y for the first dimension.

If the array is stored in the normal form, *inc1y* = 1.

If the array is stored in the transposed form, *inc1y* is the leading dimension of the array and $inc1y \geq n2$.

Specified as: an integer; $inc1y > 0$. If $inc2y = 1$, then $inc1y \geq n2$.

inc2y

is the stride between the elements in array Y for the second dimension.

If the array is stored in the transposed form, *inc2y* = 1.

If the array is stored in the normal form, *inc2y* is the leading dimension of the array and $inc2y \geq n1$.

Specified as: an integer; $inc2y > 0$. If $inc1y = 1$, then $inc2y \geq n1$.

n1

is the length of the first dimension of the two-dimensional data in the array to be transformed.

Specified as: an integer; $n1 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

n2

is the length of the second dimension of the two-dimensional data in the array to be transformed.

Specified as: an integer; $n2 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

isign

controls the direction of the transform, determining the sign *Isign* of the exponents of W_{n1} and W_{n2}, where:

If *isign* = positive value, *Isign* = + (transforming time to frequency).

If *isign* = negative value, *Isign* = - (transforming frequency to time).

Specified as: an integer; $isign > 0$ or $isign < 0$.

scale

is the scaling constant *scale*. See "Function" on page 1026 for its usage.

Specified as: a number of the data type indicated in Table 227 on page 1022, where $scale > 0.0$ or $scale < 0.0$.

aux1

is the working storage for this subroutine, where:

If $init \neq 0$, the working storage is computed.

If *init* = 0, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1

is the number of doublewords in the working storage specified in *aux1*.

Specified as: an integer; *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

aux2

has the following meaning:

If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2

is the number of doublewords in the working storage specified in *aux2*.

Specified as: an integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, SCFT2 and DCFT2 dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

On Return

y

has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, containing the elements resulting from the two-dimensional discrete Fourier transform of the data in X. Each element $y_{k1,k2}$, using zero-based indexing, is stored in Y(*k1*(*inc1y*)+*k2*(*inc2y*)) for *k1* = 0, 1, ..., *n1*-1 and *k2* = 0, 1, ..., *n2*-1.

Returned as: an array of (at least) length 1+(*n1*-1)*inc1y*+(*n2*-1)*inc2y*, containing numbers of the data type indicated in Table 227 on page 1022.

If *inc1y* = 1, the output array is stored in normal form, and *inc2y* ≥ *n1*.

If *inc2y* = 1, the output array is stored in transposed form, and *inc1y* ≥ *n2*.

See "Notes" on page 1024 for more details.

aux1

is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between program calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.

2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* ≠ 0 and *init* = 0.
3. If you specify the same array for X and Y, then *inc1x* must equal *inc1y*, and *inc2x* must equal *inc2y*. In this case, output overwrites input. If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
4. By appropriately specifying the *inc* arguments, this subroutine allows you to specify that it should use one of two forms of its arrays, the normal untransposed form or the transposed form. As a result, you **do not have to move any data**. Instead, the subroutine performs the adjustments for you. Also, either the input array or the output array can be in transposed form. The FFT computation is symmetrical with respect to *n1* and *n2*. They can be interchanged without the loss of generality. If they are interchanged, an array that is stored in the normal form appears as an array stored in the transposed form and vise versa. If, for performance reasons, the forms of the input and output arrays are different, then the input array should be specified in the normal form, and the output array should be specified in the transposed form. This can always be done by interchanging *n1* and *n2*.
5. Although the *inc* arguments for each array can be arbitrary, in most cases, one of the *inc* arguments is 1 for each array. If *inc1* = 1, the array is stored in normal form; that is, the first dimension of the array is along the columns. In this case, *inc2* is the leading dimension of the array and must be at least *n1*. Conversely, if *inc2* = 1, the array is stored in the transposed form; that is, the first dimension of the array is along the rows. In this case, *inc1* is the leading dimension of the array and must be at least *n2*. The rows of the arrays are accessed with a stride that equals the leading dimension of the array. To minimize cache interference in accessing a row, an optimal value should be used for the leading dimension of the array. You should use "STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)" on page 1195 to determine this optimal value. Example 4 in the STRIDE subroutine description explains how it is used to find either *inc1* or *inc2*.

Formulas

Processor-Independent Formulas for SCFT2 for NAUX1 and NAUX2:

The required values of *naux1* and *naux2* depend on *n1* and *n2*.

AUX1 Formulas

For 32-bit integer arguments:

If max(*n1*, *n2*) ≤ 8192, use *naux1* = 40000.

If max(*n1*, *n2*) > 8192, use *naux1* = 40000+1.14(*n1*+*n2*).

For 64-bit integer arguments:

If max(*n1*, *n2*) ≤ 8192, use *naux1* = 60000.

If max(*n1*, *n2*) > 8192, use *naux1* = 60000+1.14(*n1*+*n2*).

NAUX2 Formulas

If max(*n1*, *n2*) < 252, use *naux2* = 20000.

If max(*n1*, *n2*) ≥ 252, use *naux2* = 20000+(*r*+256)(*s*+1.14), where *r* = max(*n1*, *n2*) and *s* = min(64, *n1*, *n2*).

Processor-Independent Formulas for DCFT2 for NAUX1 and NAUX2:

The required values of *naux1* and *naux2* depend on *n1* and *n2*.

NAUX1 Formulas

For 32-bit integer arguments:

If max(*n1*, *n2*) ≤ 2048, use *naux1* = 40000.

If max(*n1*, *n2*) > 2048, use *naux1* = 40000+2.28(*n1*+*n2*).

For 64-bit integer arguments:

If max(*n1*, *n2*) ≤ 2048, use *naux1* = 60000.

If max(*n1*, *n2*) > 2048, use *naux1* = 60000+2.28(*n1*+*n2*).

NAUX2 Formulas

If max(*n1*, *n2*) < 252, use *naux2* = 20000.

If max(*n1*, *n2*) ≥ 252, use *naux2* = 20000+(2*r*+256)(*s*+2.28), where *r* = max(*n1*, *n2*) and *s* = min(64, *n1*, *n2*).

Function

The two-dimensional discrete Fourier transform of complex data in array X, with results going into array Y, is expressed as follows:

$$y_{k1,k2} = scale \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} x_{j1,j2} \; W_{n1}^{(Isign)j1k1} \; W_{n2}^{(Isign)j2k2}$$

for:

k1 = 0, 1, ..., *n1*-1
k2 = 0, 1, ..., *n2*-1

where:

$$W_{n1} = e^{-2\pi\left(\sqrt{-1}\right)/n1}$$
$$W_{n2} = e^{-2\pi\left(\sqrt{-1}\right)/n2}$$

and where:

$x_{j1,j2}$ are elements of array X.
$y_{k1,k2}$ are elements of array Y.
Isign is + or - (determined by argument *isign*).
scale is a scalar value.

For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/((*n1*)(*n2*)) and *isign* being negative. See references [“1” on page 1253], [“5” on page 1253], and [“28” on page 1254].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transform.

Error conditions

Resource Errors

Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors

None

Input-Argument Errors

1. *n1* > 37748736
2. *n2* > 37748736
3. *inc1x*|*inc2x*|*inc1y*|*inc2y* ≤ 0

4. *scale* = 0.0
5. *isign* = 0
6. The subroutine has not been initialized with the present arguments.
7. The length of one of the transforms in *n1* or *n2* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
8. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
9. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows how to compute a two-dimensional transform where both input and output are stored in normal form (*inc1x* = *inc1y* = 1). Also, *inc2x* = *inc2y* so the same array can be used for both input and output. The arrays are declared as follows:

```
      COMPLEX*8  X(6,8),Y(6,8)
      REAL*8     AUX1(20000), AUX2(1)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage: `EQUIVALENCE (X,Y)`. First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT  X INC1X INC2X Y INC1Y INC2Y N1  N2 ISIGN SCALE  AUX1  NAUX1   AUX2  NAUX2
             |    |   |     |   |   |     |   |   |    |     |      |     |      |      |
CALL SCFT2(INIT, X , 1  ,  6 , Y , 1  ,  6 , 6 , 8 ,  1 , SCALE, AUX1, 20000 , AUX2,   0)
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`SCALE = 1.0`
X is an array with 6 rows and 8 columns with (1.0, 0.0) in all locations.

Output:

Y is an array with 6 rows and 8 columns having (48.0, 0.0) in location `Y(1,1)` and (0.0, 0.0) in all others.

Example 2

This example shows how to compute a two-dimensional inverse Fourier transform. For this example, X is stored in normal untransposed form (*inc1x* = 1), and Y is stored in transposed form (*inc2y* = 1). The arrays are declared as follows:

```
      COMPLEX*16  X(6,8),Y(8,6)
      REAL*8      AUX1(20000), AUX2(1)
```

First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT  X INC1X INC2X Y INC1Y INC2Y N1  N2 ISIGN SCALE  AUX1   NAUX1   AUX2   NAUX2
             |    |   |     |   |   |     |   |   |    |     |     |       |       |       |
CALL DCFT2(INIT, X , 1  ,  6 , Y , 8  ,  1 , 6 , 8 , -1 , SCALE, AUX1 , 20000 , AUX2 ,   0)
```

```
INIT = 1 (for initialization)
INIT = 0 (for computation)
SCALE = 1.0/48.0
X  = (same as output Y in Example 1)
```

Output:

Y is an array with 8 rows and 6 columns with (1.0, 0.0) in all locations.

SRCFT2 and DRCFT2 (Real-to-Complex Fourier Transform in Two Dimensions)

Purpose

These subroutines compute the two-dimensional discrete Fourier transform of real data in a two-dimensional array.

Table 228: Data Types.

Data Types		
X, *scale*	Y	**Subroutine**
Short-precision real	Short-precision complex	SRCFT2
Long-precision real	Long-precision complex	DRCFT2

Note:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	**Syntax**
Fortran	CALL SRCFT2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*) CALL DRCFT2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)

C and C++	srcft2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*);
	
	drcft2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init

is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transform of the given array is computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: an integer. It can have any value.

x
: is the array X, containing *n1* rows and *n2* columns of data to be transformed. The data in each column is stored with stride 1. Specified as: an *inc2x* by (at least) *n2* array, containing numbers of the data type indicated in Table 228 on page 1028. See "Notes" on page 1030 for more details.

inc2x
: is the leading dimension (stride between columns) of array X. Specified as: an integer; $inc2x \geq n1$.

y
: See On Return.

inc2y
: is the leading dimension (stride between columns) of array Y. Specified as: an integer; $inc2y \geq ((n1)/2)+1$.

n1
: is the number of rows of data—that is, the length of the columns in array X involved in the computation. The length of the columns in array Y are $(n1)/2+1$.

  Specified as: an integer; $n1 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

n2
: is the number of columns of data—that is, the length of the rows in arrays X and Y involved in the computation.

  Specified as: an integer; $n2 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

isign
: controls the direction of the transform, determining the sign *Isign* of the exponents of W_{n1} and W_{n2}, where:

  If *isign* = positive value, *Isign* = + (transforming time to frequency).

  If *isign* = negative value, *Isign* = - (transforming frequency to time).

  Specified as: an integer; $isign > 0$ or $isign < 0$.

scale
: is the scaling constant *scale*. See "Function" on page 1031 for its usage.

  Specified as: a number of the data type indicated in Table 228 on page 1028, where $scale > 0.0$ or $scale < 0.0$.

aux1
: is the working storage for this subroutine, where:

  If $init \neq 0$, the working storage is computed.

  If $init = 0$, the working storage is used in the computation of the Fourier transforms.

  Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
: is the number of doublewords in the working storage specified in *aux1*.

  Specified as: an integer; $naux1 \geq$ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

aux2
: has the following meaning:

If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*.

Specified as: an integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, SRCFT2 and DRCFT2 dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

aux3
this argument is provided for migration purposes only and is ignored.

Specified as: an area of storage containing *naux3* long-precision real numbers.

naux3
this argument is provided for migration purposes only and is ignored.

Specified as: an integer.

On Return

y
has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, containing the results of the complex discrete Fourier transform of X. The output consists of *n2* columns of data. The data in each column is stored with stride 1. Due to complex conjugate symmetry, the output consists of only the first ((*n1*)/2)+1 rows of the array—that is, $y_{k1,k2}$, where *k1* = 0, 1, ..., (*n1*)/2 and *k2* = 0, 1, ..., *n2*-1.

Returned as: an *inc2y* by (at least) *n2* array, containing numbers of the data type indicated in Table 228 on page 1028.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* ≠ 0 and *init* = 0.
3. If you specify the same array for X and Y, then *inc2x* must equal (2)(*inc2y*). In this case, output overwrites input. If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

4. For selecting optimal strides (or leading dimensions *inc2x* and *inc2y*) for your input and output arrays, you should use "STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)" on page 1195. Example 5 in the STRIDE subroutine description explains how it is used for these subroutines.

Formulas

Processor-Independent Formulas for SRCFT2 for NAUX1 and NAUX2

The required values of *naux1* and *naux2* depend on *n1* and *n2*.

NAUX1 Formulas

For 32-bit integer arguments:

If max($n1/2$, $n2$) ≤ 8192, use *naux1* = 45000.
If max($n1/2$, $n2$) > 8192, use *naux1* = $40000+0.82n1+1.14n2$.

For 64-bit integer arguments:

If max($n1/2$, $n2$) ≤ 8192, use *naux1* = 65000.
If max($n1/2$, $n2$) > 8192, use *naux1* = $60000+0.82n1+1.14n2$.

NAUX2 Formulas

If $n1 \le 16384$ and $n2 < 252$, use *naux2* = 20000.
If $n1 > 16384$ and $n2 < 252$, use *naux2* = $20000+0.57n1$.
If $n2 \ge 252$, add the following to the above storage requirements:

$(n2+256)(1.14+s)$
where $s = \min(64, 1+n1/2)$.

Processor-Independent Formulas for DRCFT2 for NAUX1 and NAUX2

The required values of *naux1* and *naux2* depend on *n1* and *n2*.

NAUX1 Formulas

For 32-bit integer arguments:

If $n \le 2048$, use *naux1* = 42000.
If $n > 2048$, use *naux1* = $40000+1.64n1+2.28n2$,
where $n = \max(n1/2, n2)$.

For 64-bit integer arguments:

If $n \le 2048$, use *naux1* = 62000.
If $n > 2048$, use *naux1* = $60000+1.64n1+2.28n2$,
where $n = \max(n1/2, n2)$.

NAUX2 Formulas

If $n1 \le 4096$ and $n2 < 252$, use *naux2* = 20000.
If $n1 > 4096$ and $n2 < 252$, use *naux2* = $20000+1.14n1$.
If $n2 \ge 252$, add the following to the above storage requirements:

$((2)n2+256)\ (2.28+s)$
where $s = \min(64, 1+n1/2)$.

Function

The two-dimensional complex conjugate even discrete Fourier transform of real data in array X, with results going into array Y, is expressed as follows:

$$y_{k1,k2} = scale \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} x_{j1,j2} \, W_{n1}^{(Isign)\, j1k1} \, W_{n2}^{(Isign)\, j2k2}$$

for:

$k1$ = 0, 1, ..., $n1$-1
$k2$ = 0, 1, ..., $n2$-1

where:

$$W_{n1} = e^{-2\pi\left(\sqrt{-1}\right)/n1}$$
$$W_{n2} = e^{-2\pi\left(\sqrt{-1}\right)/n2}$$

and where:

$x_{j1,j2}$ are elements of array X.
$y_{k1,k2}$ are elements of array Y.
Isign is + or - (determined by argument *isign*).
scale is a scalar value.

The output in array Y is complex. For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/((*n1*)(*n2*)) and *isign* being negative. See references [“1” on page 1253], [“5” on page 1253], [“27” on page 1254], and [“28” on page 1254].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transform.

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors
None

Input-Argument Errors

1. *n1* > 37748736
2. *n2* > 37748736
3. *inc2x* < *n1*
4. *inc2y* < (*n1*)/2+1
5. *scale* = 0.0
6. *isign* = 0
7. The subroutine has not been initialized with the present arguments.
8. The length of one of the transforms in *n1* or *n2* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
9. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

10. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows how to compute a two-dimensional transform. The arrays are declared as follows:

```
     COMPLEX*8  Y(0:6,0:7)
     REAL*4     X(0:11,0:7)
     REAL*8     AUX1(1000), AUX2(1), AUX3(1)
```

First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT   X   INC2X Y INC2Y N1   N2 ISIGN SCALE  AUX1   NAUX1  AUX2  NAUX2  AUX3
NAUX3
             |     |     |   |   |    |    |   |     |      |       |     |      |     |      |
CALL SRCFT2(INIT, X , 12 , Y , 7 , 12 , 8 , 1 , SCALE, AUX1 , 1000 , AUX2 ,  0 ,  AUX3 ,
0 )
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`SCALE = 1.0`

X is an array with 12 rows and 8 columns having 1.0 in location `X(0,0)` and 0.0 in all others.

Output:

Y is an array with 7 rows and 8 columns with (1.0, 0.0) in all locations.

Example 2

This example shows another transform computation with different data using the same initialized array AUX1 in Example 1.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT X   INC2X Y INC2Y N1   N2 ISIGN SCALE  AUX1  NAUX1  AUX2  NAUX2  AUX3  NAUX3
             |   |     |   |   |    |    |    |     |      |     |      |      |     |     |
CALL SRCFT2( 0 , X , 12 , Y , 7 , 12 , 8 ,  1 , SCALE, AUX1, 1000 , AUX2,   0 ,  AUX3 ,  0 )
```

`SCALE = 1.0`
X is an array with 12 rows and 8 columns with 1.0 in all locations.

Output:

Y is an array with 7 rows and 8 columns having (96.0, 0.0) in location `Y(0,0)` and (0.0, 0.0) in all others.

Example 3

This example shows the same array being used for input and output, where *isign* = -1 and *scale* = 1/((N1)(N2)). The arrays are declared as follows:

```
     COMPLEX*16  Y(0:8,0:7)
     REAL*8     X(0:19,0:7)
     REAL*8     AUX1(1000), AUX2(1), AUX3(1)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage.

```
     EQUIVALENCE (X,Y)
```

This requires *inc2x* ≥ 2(*inc2y*). First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT  X  INC2X Y INC2Y N1   N2 ISIGN SCALE  AUX1   NAUX1  AUX2  NAUX2  AUX3   
NAUX3
             |    |    |   |   |   |    |    |      |     |      |     |      |     |      |
CALL DRCFT2(INIT, X , 20 , Y , 9 , 16 , 8 , -1 , SCALE, AUX1 , 1000 , AUX2 ,  0 ,  AUX3 , 
0 )
```

INIT = 1 (for initialization)
INIT = 0 (for computation)
SCALE = 1.0/128.0

```
        ┌                                                ┐
        |  2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0 |
        |  2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0 |
        | -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0 |
        | -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0 |
        |  2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0 |
        |  2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0 |
        | -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0 |
        | -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0 |
        |  2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0 |
X    =  |  2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0 |
        | -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0 |
        | -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0 |
        |  2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0 |
        |  2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0 |
        | -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0 |
        | -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0 |
        |   .     .     .     .     .     .     .     .  |
        |   .     .     .     .     .     .     .     .  |
        |   .     .     .     .     .     .     .     .  |
        |   .     .     .     .     .     .     .     .  |
        └                                                ┘
```

Output:

Y is an array with 9 rows and 8 columns having (1.0, 1.0) in location Y(4,2) and (0.0, 0.0) in all others.

SCRFT2 and DCRFT2 (Complex-to-Real Fourier Transform in Two Dimensions)

Purpose

These subroutines compute the two-dimensional discrete Fourier transform of complex conjugate even data in a two-dimensional array.

Table 229: Data Types.

Data Types		
X	Y, *scale*	**Subroutine**
Short-precision complex	Short-precision real	SCRFT2
Long-precision complex	Long-precision real	DCRFT2

Note:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SCRFT2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*) CALL DCRFT2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)

C and C++	scrft2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*);
	dcrft2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init

is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transform of the given array is computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: an integer. It can have any value.

x

is the array X, containing *n2* columns of data to be transformed. Due to complex conjugate symmetry, the input consists of only the first ((*n1*)/2)+1 rows of the array—that is, $x_{j1,j2}$, *j1* = 0, 1, ..., (*n1*)/2, *j2* = 0, 1, ..., *n2*-1. The data in each column is stored with stride 1.

Specified as: an *inc2x* by (at least) *n2* array, containing numbers of the data type indicated in Table 229 on page 1034.

inc2x

is the leading dimension (stride between columns) of array X. Specified as: an integer; *inc2x* ≥ ((*n1*)/2)+1.

y

See On Return.

inc2y

is the leading dimension (stride between the columns) of array Y.

Specified as: an integer; *inc2y* ≥ *n1*+2.

n1

is the number of rows of data—that is, the length of the columns in array Y involved in the computation. The length of the columns in array X are (*n1*)/2+1.

Specified as: an integer; *n1* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

n2
: is the number of columns of data—that is, the length of the rows in arrays X and Y involved in the computation.

  Specified as: an integer; $n2 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

isign
: controls the direction of the transform, determining the sign *Isign* of the exponents of W_{n1} and W_{n2}, where:

  If *isign* = positive value, *Isign* = + (transforming time to frequency).

  If *isign* = negative value, *Isign* = - (transforming frequency to time).

  Specified as: an integer; *isign* > 0 or *isign* < 0.

scale
: is the scaling constant *scale*. See "Function" on page 1038 for its usage.

  Specified as: a number of the data type indicated in Table 229 on page 1034, where *scale* > 0.0 or *scale* < 0.0.

aux1
: is the working storage for this subroutine, where:

  If *init* ≠ 0, the working storage is computed.

  If *init* = 0, the working storage is used in the computation of the Fourier transforms.

  Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
: is the number of doublewords in the working storage specified in *aux*.

  Specified as: an integer; *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

aux2
: has the following meaning:

  If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

  Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

  Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
: is the number of doublewords in the working storage specified in *aux2*.

  Specified as: an integer, where:

  If *naux2* = 0 and error 2015 is unrecoverable, SCRFT2 and DCRFT2 dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

  Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

aux3
: this argument is provided for migration purposes only and is ignored.

Specified as: an area of storage, containing *naux3* long-precision real numbers.

naux3
this argument is provided for migration purposes only and is ignored.

Specified as: an integer.

On Return

y
has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is the array Y, containing *n1* rows and *n2* columns of results of the real discrete Fourier transform of X. The data in each column of Y is stored with stride 1.

Returned as: an *inc2y* by (at least) *n2* array, containing numbers of the data type indicated in Table 229 on page 1034. See “Notes” on page 1037 for more details.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between program calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* ≠ 0 and *init* = 0.
3. If you specify the same array for X and Y, then (2)(*inc2x*) must equal *inc2y*. In this case, output overwrites input. If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.
4. For selecting optimal strides (or leading dimensions *inc2x* and *inc2y*) for your input and output arrays, you should use “STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1195. Example 6 in the STRIDE subroutine description explains how it is used for these subroutines.

Formulas

Processor-Independent Formulas for SCRFT2 for NAUX1 and NAUX2

The required values of *naux1* and *naux2* depend on *n1* and *n2*.

NAUX1 Formulas
For 32-bit integer arguments:

If max(*n1*/2, *n2*) ≤ 8192, use *naux1* = 45000. If max(*n1*/2, *n2*) > 8192, use *naux1* = 40000+0.82*n1*+1.14*n2*.

For 64-bit integer arguments:

If max(*n1*/2, *n2*) ≤ 8192, use *naux1* = 65000. If max(*n1*/2, *n2*) > 8192, use *naux1* = 60000+0.82*n1*+1.14*n2*.

NAUX2 Formulas
If *n1* ≤ 16384 and *n2* < 252, use *naux2* = 20000.

If *n1* > 16384 and *n2* < 252, use *naux2* = 20000+0.57*n1*.

If *n2* ≥ 252, add the following to the above storage requirements:

(*n2*+256)(1.14+*s*)

where *s* = min(64, 1+*n1*/2).

Processor-Independent Formulas for DCRFT2 for NAUX1 and NAUX2:

The required values of *naux1* and *naux2* depend on *n1* and *n2*.

NAUX1 Formulas

For 32-bit integer arguments:

If *n* ≤ 2048, use *naux1* = 42000. If *n* > 2048, use *naux1* = 40000+1.64*n1*+2.28*n2*, where *n* = max(*n1*/2, *n2*).

For 64-bit integer arguments:

If *n* ≤ 2048, use *naux1* = 62000.

If *n* > 2048, use *naux1* = 60000+1.64*n1*+2.28*n2*,

where *n* = max(*n1*/2, *n2*).

NAUX2 Formulas

If *n1* ≤ 4096 and *n2* < 252, use *naux2* = 20000. If *n1* > 4096 and *n2* < 252, use *naux2* = 20000+1.14*n1*.

If *n2* ≥ 252, add the following to the above storage requirements:

((2)*n2*+256) (2.28+*s*)

where *s* = min(64, 1+*n1*/2).

Function

The two-dimensional discrete Fourier transform of complex conjugate even data in array X, with results going into array Y, is expressed as follows:

$$y_{k1,k2} = scale \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} x_{j1,j2} \; W_{n1}^{(Isign)j1k1} \; W_{n2}^{(Isign)j2k2}$$

for:

k1 = 0, 1, ..., *n1*-1
k2 = 0, 1, ..., *n2*-1

where:

$$W_{n1} = e^{-2\pi\left(\sqrt{-1}\right)/n1}$$
$$W_{n2} = e^{-2\pi\left(\sqrt{-1}\right)/n2}$$

and where:

$x_{j1,j2}$ are elements of array X.
$y_{k1,k2}$ are elements of array Y.
Isign is + or - (determined by argument *isign*).
scale is a scalar value.

Because of the complex conjugate symmetry, the output in array Y is real. For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/((*n1*)(*n2*)) and *isign* being negative. See references ["1" on page 1253], ["5" on page 1253], and ["28" on page 1254].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transform.

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors
None

Input-Argument Errors

1. *n1* > 37748736
2. *n2* > 37748736
3. *inc2x* < (*n1*)/2+1
4. *inc2y* < *n1*+2
5. *scale* = 0.0
6. *isign* = 0
7. The subroutine has not been initialized with the present arguments.
8. The length of one of the transforms in *n1* or *n2* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
9. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
10. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows how to compute a two-dimensional transform. The arrays are declared as follows:

```
      REAL*4     Y(0:13,0:7)
      COMPLEX*8  X(0:6,0:7)
      REAL*8     AUX1(1000), AUX2(1), AUX3(1)
```

First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT  X INC2X Y  INC2Y N1   N2 ISIGN SCALE   AUX1  NAUX1   AUX2  NAUX2  AUX3  
NAUX3
             |    |   |   |    |   |    |    |     |      |      |      |      |     |     
|
CALL SCRFT2(INIT, X , 7 , Y , 14 , 12 , 8 , -1 , SCALE , AUX1 , 1000 , AUX2 ,  0 ,  AUX3 ,  
0 )
```

`INIT = 1` (for initialization)
`INIT = 0` (for computation)
`SCALE = 1.0/96.0`
X is an array with 7 rows and 8 columns with (1.0, 0.0) in all locations.

Output:

```
        | 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
        | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
        | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
        | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
        | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
        | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
Y    =  | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
        | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
        | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
        | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
        | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
        | 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 |
        |  .    .    .    .    .    .    .    .  |
        |  .    .    .    .    .    .    .    .  |
```

Example 2

This example shows another transform computation with different data using the same initialized array AUX1 in Example 1.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT X INC2X Y  INC2Y N1   N2 ISIGN SCALE   AUX1   NAUX1  AUX2   NAUX2  AUX3   
NAUX3
             |   |   |   |    |   |    |    |     |      |       |     |       |     |
|
CALL SCRFT2( 0 , X , 7 , Y , 14 , 12 , 8 , -1 , SCALE , AUX1 , 1000 , AUX2 ,   0  , AUX3 , 
0 )
```

```
SCALE    =  1.0/96.0
```

X is an array with 7 rows and 8 columns having (96.0, 0.0) in location `X(0,0)` and (0.0, 0.0) in all others.

Output:

```
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
Y    =  | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        | 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
        |  .    .    .    .    .    .    .    .  |
        |  .    .    .    .    .    .    .    .  |
```

Example 3

This example shows the same array being used for input and output. The arrays are declared as follows:

```
     REAL*8      Y(0:17,0:7)
     COMPLEX*16  X(0:8,0:7)
     REAL*8      AUX1(1000), AUX2(1), AUX3(1)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage.

```
     EQUIVALENCE (X,Y)
```

This requires *inc2y* = 2(*inc2x*). First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT  X INC2X Y  INC2Y N1  N2 ISIGN SCALE   AUX1   NAUX1  AUX2   NAUX2  AUX3
NAUX3
             |    |   |   |    |    |   |   |     |      |       |      |      |     |
|
CALL DCRFT2(INIT, X , 9 , Y , 18 , 16 , 8 , 1 , SCALE , AUX1 , 1000 , AUX2 ,   0 ,  AUX3 ,
0 )
```

INIT = 1 (for initialization)
INIT = 0 (for computation)
SCALE = 1.0

X is an array with 9 rows and 8 columns having (1.0, 1.0) in location X(4,2) and (0.0, 0.0) in all others.

Output:

```
        |  2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0 |
        |  2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0 |
        | -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0 |
        | -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0 |
        |  2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0 |
        |  2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0 |
        | -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0 |
        | -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0 |
Y    =  |  2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0 |
        |  2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0 |
        | -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0 |
        | -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0 |
        |  2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0 |
        |  2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0 |
        | -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0 |
        | -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0 |
        |   .     .     .     .     .     .     .     .  |
        |   .     .     .     .     .     .     .     .  |
```

SCFT3 and DCFT3 (Complex Fourier Transform in Three Dimensions)

Purpose

These subroutines compute the three-dimensional discrete Fourier transform of complex data.

Table 230: Data Types.

Data Types		
X, Y	*scale*	**Subroutine**
Short-precision complex	Short-precision real	SCFT3
Long-precision complex	Long-precision real	DCFT3

Note:

1. For each use, only one invocation of this subroutine is necessary. The initialization phase, preparing the working storage, is a relatively small part of the total computation, so it is performed on each invocation.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SCFT3 | DCFT3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*)
C and C++	scft3 | dcft3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*);

On Entry

x

is the array X, containing the three-dimensional data to be transformed, where each element $x_{j1,j2,j3}$, using zero-based indexing, is stored in X(*j1*+*j2*(*inc2x*)+*j3*(*inc3x*)) for *j1* = 0, 1, ..., *n1*-1, *j2* = 0, 1, ..., *n2*-1, and *j3* = 0, 1, ..., *n3*-1. The strides for the elements in the first, second, and third dimensions are assumed to be 1, *inc2x*(≥ *n1*), and *inc3x*(≥ (*n2*)(*inc2x*)), respectively.

Specified as: an array, containing numbers of the data type indicated in Table 230 on page 1041. If the array is dimensioned X(LDA1,LDA2,LDA3), then LDA1 = *inc2x*, (LDA1)(LDA2) = *inc3x*, and LDA3 ≥ *n3*. For information on how to set up this array, see "Setting Up Your Data" on page 959. For more details, see "Notes" on page 1043.

inc2x

is the stride between the elements in array X for the second dimension.

Specified as: an integer; *inc2x* ≥ *n1*.

inc3x

is the stride between the elements in array X for the third dimension.

Specified as: an integer; *inc3x* ≥ (*n2*)(*inc2x*).

y

See On Return.

inc2y

is the stride between the elements in array Y for the second dimension.

Specified as: an integer; *inc2y* ≥ *n1*.

inc3y

is the stride between the elements in array Y for the third dimension.

Specified as: an integer; *inc3y* ≥ (*n2*)(*inc2y*).

n1

is the length of the first dimension of the three-dimensional data in the array to be transformed.

Specified as: an integer; *n1* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

n2

is the length of the second dimension of the three-dimensional data in the array to be transformed.

Specified as: an integer; *n2* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

n3
is the length of the third dimension of the three-dimensional data in the array to be transformed.

Specified as: an integer; $n3 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

isign
controls the direction of the transform, determining the sign *Isign* of the exponents of W_{n1}, W_{n2}, and W_{n3}, where:

If *isign* = positive value, *Isign* = + (transforming time to frequency).

If *isign* = negative value, *Isign* = - (transforming frequency to time).

Specified as: an integer; $isign > 0$ or $isign < 0$.

scale
is the scaling constant *scale*. See "Function" on page 1045 for its usage.

Specified as: a number of the data type indicated in Table 230 on page 1041, where $scale > 0.0$ or $scale < 0.0$.

aux
has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is a storage work area used by this subroutine.

Specified as: an area of storage, containing *naux* long-precision real numbers. On output, the contents are overwritten.

naux
is the number of doublewords in the working storage specified in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SCFT3 and DCFT3 dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

On Return

y
is the array Y, containing the elements resulting from the three-dimensional discrete Fourier transform of the data in X. Each element $y_{k1,k2,k3}$, using zero-based indexing, is stored in Y(*k1*+*k2*(*inc2y*)+*k3*(*inc3y*)) for *k1* = 0, 1, ..., *n1*-1, *k2* = 0, 1, ..., *n2*-1, and *k3* = 0, 1, ..., *n3*-1. The strides for the elements in the first, second, and third dimensions are assumed to be 1, *inc2y*(≥ *n1*), and *inc3y*(≥ (*n2*)(*inc2y*)), respectively.

Returned as: an array, containing numbers of the data type indicated in Table 230 on page 1041. If the array is dimensioned `Y(LDA1,LDA2,LDA3)`, then LDA1 = *inc2y*, (LDA1)(LDA2) = *inc3y*, and LDA3 ≥ *n3*. For information on how to set up this array, see "Setting Up Your Data" on page 959. For more details, see "Notes" on page 1043.

Notes

1. If you specify the same array for X and Y, then *inc2x* must be greater than or equal to *inc2y*, and *inc3x* must be greater than or equal to *inc3y*. In this case, output overwrites input. When using the ESSL SMP

Libraries in a multithreaded environment, if *inc2x* > *inc2y* or *inc3x* > *inc3y*, these subroutines run on a single thread and issue an attention message.

If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

2. You should use "STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)" on page 1195 to determine the optimal values for the strides *inc2y* and *inc3y* for your output array. The strides for your input array do not affect performance. Example 7 in the STRIDE subroutine description explains how it is used for these subroutines. For additional information on how to set up your data, see "Setting Up Your Data" on page 959.

Formulas

Processor-Independent Formulas for SCFT3 for NAUX:

Use the following formulas for calculating *naux*:

For 32-bit integer arguments:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 8192, use *naux* = 60000.
   If *n1* > 8192, use *naux* = 60000+2.28*n1*.
2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 8192, use *naux* = 60000+λ.
   If *n1* > 8192, use *naux* = 60000+2.28*n1*+λ,
   where λ = (*n2*+256)(*s*+2.28)
   and *s* = min(64, *n1*).
3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 8192, use *naux* = 60000+ψ.
   If *n1* > 8192, use *naux* = 60000+2.28*n1*+ψ,

   where ψ = (*n3*+256)(*s*+2.28)
   and *s* = min(64, (*n1*)(*n2*)).
4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

For 64-bit integer arguments:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 8192, use *naux* = 90000.
   If *n1* > 8192, use *naux* = 90000+2.28*n1*.
2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 8192, use *naux* = 90000+λ.
   If *n1* > 8192, use *naux* = 90000+2.28*n1*+λ,

   where λ = (*n2*+256)(*s*+2.28)
   and *s* = min(64, *n1*).
3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 8192, use *naux* = 90000+ψ.
   If *n1* > 8192, use *naux* = 90000+2.28*n1*+ψ,

   where ψ = (*n3*+256)(*s*+2.28)
   and *s* = min(64, (*n1*)(*n2*)).
4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

Processor-Independent Formulas for DCFT3 for NAUX:

Use the following formulas for calculating *naux*:

For 32-bit integer arguments:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 2048, use *naux* = 60000.
   If *n1* > 2048, use *naux* = 60000+4.56*n1*.
2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 2048, use *naux* = 60000+λ.
   If *n1* > 2048, use *naux* = 60000+4.56*n1*+λ,

   where λ = ((2)*n2*+256)(*s*+4.56)
   and *s* = min(64, *n1*).
3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 2048, use *naux* = 60000+ψ.
   If *n1* > 2048, use *naux* = 60000+4.56*n1*+ψ,

   where ψ = ((2)*n3*+256)(*s*+4.56)
   and *s* = min(64, (*n1*)(*n2*)).
4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

For 64-bit integer arguments:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 2048, use *naux* = 90000.
   If *n1* > 2048, use *naux* = 90000+4.56*n1*.
2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 2048, use *naux* = 90000+λ.
   If *n1* > 2048, use *naux* = 90000+4.56*n1*+λ,

   where λ = ((2)*n2*+256)(*s*+4.56)
   and *s* = min(64, *n1*).
3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 2048, use *naux* = 90000+ψ.
   If *n1* > 2048, use *naux* = 90000+4.56*n1*+ψ,

   where ψ = ((2)*n3*+256)(*s*+4.56)
   and *s* = min(64, (*n1*)(*n2*)).
4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

Function

The three-dimensional discrete Fourier transform of complex data in array X, with results going into array Y, is expressed as follows:

$$y_{k1,k2,k3} = scale \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \sum_{j3=0}^{n3-1} x_{j1,j2,j3}\; W_{n1}^{(Isign)j1k1}\; W_{n2}^{(Isign)j2k2}\; W_{n3}^{(Isign)j3k3}$$

for:

$k1 = 0, 1, ..., n1-1$
$k2 = 0, 1, ..., n2-1$
$k3 = 0, 1, ..., n3-1$

where:

$$W_{n1} = e^{-2\pi\left(\sqrt{-1}\right)/n1}$$
$$W_{n2} = e^{-2\pi\left(\sqrt{-1}\right)/n2}$$
$$W_{n3} = e^{-2\pi\left(\sqrt{-1}\right)/n3}$$

and where:

$x_{j1,j2,j3}$ are elements of array X.
$y_{k1,k2,k3}$ are elements of array Y.
Isign is + or - (determined by argument *isign*).
scale is a scalar value.

For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/((*n1*)(*n2*)(*n3*)) and *isign* being negative. See references [“1” on page 1253], [“5” on page 1253], [“6” on page 1253], [“27” on page 1254], and [“28” on page 1254].

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors
None

Input-Argument Errors

1. *n1* > 37748736
2. *n2* > 37748736
3. *n3* > 37748736
4. *inc2x* < *n1*
5. *inc3x* < (*n2*)(*inc2x*)
6. *inc2y* < *n1*
7. *inc3y* < (*n2*)(*inc2y*)
8. *scale* = 0.0
9. *isign* = 0
10. The length of one of the transforms in *n1*, *n2*, or *n3* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
11. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example

This example shows how to compute a three-dimensional transform. In this example, INC2X ≥ INC2Y and INC3X ≥ INC3Y, so that the same array can be used for both input and output. The STRIDE subroutine is called to select good values for the INC2Y and INC3Y strides. (As explained below, STRIDE is not called for INC2X and INC3X.) Using the transform lengths (N1 = 32, N2 = 64, and N3 =

40) along with the output data type (short-precision complex: `'C'`), STRIDE is called once for each stride needed. First, it is called for INC2Y:

```
    CALL STRIDE (N2,N1,INC2Y,'C',0)
```

The output value returned for INC2Y is 32. Then STRIDE is called again for INC3Y:

```
    CALL STRIDE (N3,N2*INC2Y,INC3Y,'C',0)
```

The output value returned for INC3Y is 2056. Because INC3Y is not a multiple of INC2Y, Y is not declared as a three-dimensional array. It is declared as a two-dimensional array, `Y(INC3Y,N3)`.

To equivalence the X and Y arrays requires INC2X ≥ INC2Y and INC3X ≥ INC3Y. Therefore, INC2X is set equal to INC2Y(= 32). Also, to declare the X array as a three-dimensional array, INC3X must be a multiple of INC2X. Therefore, its value is set as INC3X = (65)(INC2X) = 2080.

The arrays are declared as follows:

```
      COMPLEX*8  X(32,65,40),Y(2056,40)
      REAL*8     AUX(1)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```
      EQUIVALENCE (X,Y)
```

Note: Because NAUX = 0, this subroutine dynamically allocates the AUX working storage.

Call Statement and Input:

```
              X  INC2X INC3X  Y  INC2Y INC3Y  N1   N2   N3 ISIGN SCALE   AUX  NAUX
              |    |     |    |    |     |     |    |    |   |     |      |     |
 CALL SCFT3( X , 32 , 2080 , Y , 32 , 2056 , 32 , 64 , 40 , 1 , SCALE , AUX ,  0 )
```

`SCALE = 1.0`
X has (1.0,2.0) in location `X(1,1,1)` and (0.0,0.0) in all other locations.

Output:

Y has (1.0,2.0) in locations Y(*ij*,*k*), where *ij* = 1, 2048 and *j* = 1, 40. It remains unchanged elsewhere.

SRCFT3 and DRCFT3 (Real-to-Complex Fourier Transform in Three Dimensions)

Purpose

These subroutines compute the three-dimensional discrete Fourier transform of real data in a three-dimensional array.

Table 231: Data Types.

Data Types		
X, *scale*	Y	**Subroutine**
Short-precision real	Short-precision complex	SRCFT3
Long-precision real	Long-precision complex	DRCFT3

Note:

1. For each use, only one invocation of this subroutine is necessary. The initialization phase, preparing the working storage, is a relatively small part of the total computation, so it is performed on each invocation.

2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	**Syntax**
Fortran	CALL SRCFT3 | DRCFT3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*)
C and C++	srcft3 | drcft3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*);

On Entry

x

is the array X, containing the three-dimensional data to be transformed, where each element $x_{j1,j2,j3}$, using zero-based indexing, is stored in X(*j1*+*j2*(*inc2x*)+*j3*(*inc3x*)) for *j1* = 0, 1, ..., *n1*-1, *j2* = 0, 1, ..., *n2*-1, and *j3* = 0, 1, ..., *n3*-1. The strides for the elements in the first, second, and third dimensions are assumed to be 1, *inc2x*(≥ *n1*), and *inc3x*(≥ (*n2*)(*inc2x*)), respectively.

Specified as: an array, containing numbers of the data type indicated in Table 231 on page 1047. If the array is dimensioned `X(LDA1,LDA2,LDA3)`, then LDA1 = *inc2x*, (LDA1)(LDA2) = *inc3x*, and LDA3 ≥ *n3*. For information on how to set up this array, see "Setting Up Your Data" on page 959. For more details, see "Notes" on page 1049.

inc2x

is the stride between the elements in array X for the second dimension.

Specified as: an integer; *inc2x* ≥ *n1*.

inc3x

is the stride between the elements in array X for the third dimension.

Specified as: an integer; *inc3x* ≥ (*n2*)(*inc2x*).

y

See On Return.

inc2y

is the stride between the elements in array Y for the second dimension.

Specified as: an integer; *inc2y* ≥ *n1*/2+1.

inc3y

is the stride between the elements in array Y for the third dimension.

Specified as: an integer; *inc3y* ≥ (*n2*)(*inc2y*).

n1

is the length of the first dimension of the three-dimensional data in the array to be transformed.

Specified as: an integer; *n1* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

n2

is the length of the second dimension of the three-dimensional data in the array to be transformed.

Specified as: an integer; *n2* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

n3

is the length of the third dimension of the three-dimensional data in the array to be transformed.

Specified as: an integer; *n3* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

isign
controls the direction of the transform, determining the sign *Isign* of the exponents of W_{n1}, W_{n2}, and W_{n3}, where:

If *isign* = positive value, *Isign* = + (transforming time to frequency).

If *isign* = negative value, *Isign* = - (transforming frequency to time).

Specified as: an integer; *isign* > 0 or *isign* < 0.

scale
is the scaling constant *scale*. See "Function" on page 1051 for its usage.

Specified as: a number of the data type indicated in Table 231 on page 1047, where *scale* > 0.0 or *scale* < 0.0.

aux
has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is a storage work area used by this subroutine.

Specified as: an area of storage, containing *naux* long-precision real numbers. On output, the contents are overwritten.

naux
is the number of doublewords in the working storage specified in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SRCFT3 and DRCFT3 dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

On Return

y
is the array Y, containing the elements resulting from the three-dimensional discrete Fourier transform of the data in X. Each element $y_{k1,k2,k3}$, using zero-based indexing, is stored in Y(*k1*+*k2*(*inc2y*)+*k3*(*inc3y*)) for *k1* = 0, 1, ..., *n1*/2, *k2* = 0, 1, ..., *n2*-1, and *k3* = 0, 1, ..., *n3*-1. Due to complex conjugate symmetry, the output consists of only the first *n1*/2+1 values along the first dimension of the array, for *k1* = 0, 1, ..., *n1*/2. The strides for the elements in the first, second, and third dimensions are assumed to be 1, *inc2y*(≥ *n1*/2+1), and *inc3y*(≥ (*n2*)(*inc2y*)), respectively.

Returned as: an array, containing numbers of the data type indicated in Table 231 on page 1047. If the array is dimensioned `Y(LDA1,LDA2,LDA3)`, then LDA1 = *inc2y*, (LDA1)(LDA2) = *inc3y*, and LDA3 ≥ *n3*. For information on how to set up this array, see "Setting Up Your Data" on page 959. For more details, see "Notes" on page 1049.

Notes

1. If you specify the same array for X and Y, then *inc2x* must be greater than or equal to (2)(*inc2y*), and *inc3x* must be greater than or equal to (2)(*inc3y*). In this case, output overwrites input. When using the ESSL SMP Libraries in a multithreaded environment, if *inc2x* > (2)(*inc2y*) or *inc3x* > (2)(*inc3y*), these subroutines run on a single thread and issue an attention message.

If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.

2. The strides for your input array do not affect performance as long as they are even numbers. In addition, you should use “STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1195 to determine the optimal values for the strides *inc2y* and *inc3y* for your output array. Example 8 in the STRIDE subroutine description explains how it is used for these subroutines. For additional information on how to set up your data, see “Setting Up Your Data” on page 959.

Formulas

Processor-Independent Formulas for SRCFT3 for NAUX

Use the following formulas for calculating *naux*:

For 32-bit integer arguments:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 16384, use *naux* = 65000.
   If *n1* > 16384, use *naux* = 60000+1.39*n1*.
2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 16384, use *naux* = 65000+λ.
   If *n1* > 16384, use *naux* = 60000+1.39*n1*+λ,

   where λ = (*n2*+256)(*s*+2.28) and *s* = min(64, 1+*n1*/2).
3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 16384, use *naux* = 65000+ψ.
   If *n1* > 16384, use *naux* = 60000+1.39*n1*+ψ,

   where ψ = (*n3*+256)(s+2.28) and *s* = min(64, (*n2*)(1+*n1*/2)).
4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

For 64-bit integer arguments:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 16384, use *naux* = 95000.
   If *n1* > 16384, use *naux* = 90000+1.39*n1*.
2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 16384, use *naux* = 95000+λ.
   If *n1* > 16384, use *naux* = 90000+1.39*n1*+λ,

   where λ = (*n2*+256)(*s*+2.28) and *s* = min(64, 1+*n1*/2).
3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 16384, use *naux* = 95000+ψ.
   If *n1* > 16384, use *naux* = 90000+1.39*n1*+ψ,

   where ψ = (*n3*+256)(*s*+2.28) and *s* = min(64, (*n2*)(1+*n1*/2)).
4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

If *inc2x* or *inc3x* is an odd number, or if array X is not aligned on a doubleword boundary, you should add the following amount to all the formulas given above:

n2(1+*n1*/2)

Processor-Independent Formulas for DRCFT3 for NAUX

Use the following formulas for calculating *naux*:

For 32-bit integer arguments:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 4096, use *naux* = 62000.
   If *n1* > 4096, use *naux* = 60000+2.78*n1*.
2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 4096, use *naux* = 62000+λ.
   If *n1* > 4096, use *naux* = 60000+2.78*n1*+λ,

   where λ = ((2)*n2*+256)(*s*+4.56)
   and *s* = min(64, *n1*/2).
3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 4096, use *naux* = 62000+ψ.
   If *n1* > 4096, use *naux* = 60000+2.78*n1*+ψ,

   where ψ = ((2)*n3*+256)(*s*+4.56)
   and *s* = min(64, *n2*(1+*n1*/2)).
4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

For 64-bit integer arguments:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 4096, use *naux* = 92000.
   If *n1* > 4096, use *naux* = 90000+2.78*n1*.
2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 4096, use *naux* = 92000+λ.
   If *n1* > 4096, use *naux* = 90000+2.78*n1*+λ,

   where λ = ((2)*n2*+256)(*s*+4.56)
   and *s* = min(64, *n1*/2).
3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 4096, use *naux* = 92000+ψ.
   If *n1* > 4096, use *naux* = 90000+2.78*n1*+ψ,

   where ψ = ((2)*n3*+256)(*s*+4.56)
   and *s* = min(64, *n2*(1+*n1*/2)).
4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

Function

The three-dimensional complex conjugate even discrete Fourier transform of real data in array X, with results going into array Y, is expressed as follows:

$$y_{k1,k2,k3} = scale \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \sum_{j3=0}^{n3-1} x_{j1,j2,j3} \; W_{n1}^{(Isign)j1k1} \; W_{n2}^{(Isign)j2k2} \; W_{n3}^{(Isign)j3k3}$$

for:

k1 = 0, 1, ..., *n1*-1
k2 = 0, 1, ..., *n2*-1
k3 = 0, 1, ..., *n3*-1

where:

$$W_{n1} = e^{-2\pi\left(\sqrt{-1}\right)/n1}$$
$$W_{n2} = e^{-2\pi\left(\sqrt{-1}\right)/n2}$$
$$W_{n3} = e^{-2\pi\left(\sqrt{-1}\right)/n3}$$

and where:

$x_{j1,j2,j3}$ are elements of array X.
$y_{k1,k2,k3}$ are elements of array Y.
Isign is + or - (determined by argument *isign*).
scale is a scalar value.

The output in array Y is complex. For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/((*n1*)(*n2*)(*n3*)) and *isign* being negative. See references [“1” on page 1253], [“5” on page 1253], [“6” on page 1253], [“27” on page 1254], and [“28” on page 1254].

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors
None

Input-Argument Errors

1. *n1* > 37748736
2. *n2* > 37748736
3. *n3* > 37748736
4. *inc2x* < *n1*
5. *inc3x* < (*n2*)(*inc2x*)
6. *inc2y* < *n1*/2+1
7. *inc3y* < (*n2*)(*inc2y*)
8. *scale* = 0.0
9. *isign* = 0
10. The length of one of the transforms in *n1*, *n2*, or *n3* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
11. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example

This example shows how to compute a three-dimensional transform. In this example, `INC2X` ≥ (2)(`INC2Y`) and `INC3X` ≥ (2)(`INC3Y`), so that the same array can be used for both input and output. The STRIDE subroutine is called to select good values for the `INC2Y` and `INC3Y` strides. Using the transform lengths (N1 = 32, N2 = 64, and N2 = 40) along with the output data type (short-precision complex: 'C'), STRIDE is called once for each stride needed. First, it is called for INC2Y:

```
    CALL STRIDE (N2,N1/2+1,INC2Y,'C',0)
```

The output value returned for INC2Y is 17. (This value is equal to N1/2+1.) Then STRIDE is called again for INC3Y:

```
    CALL STRIDE (N3,N2*INC2Y,INC3Y,'C',0)
```

The output value returned for INC3Y is 1088. Because INC3Y is a multiple of INC2Y—that is, INC3Y = (N2)(INC2Y)—Y is declared as a three-dimensional array, Y(17,64,40). (In general, for larger arrays, these types of values for INC2Y and INC3Y are not returned by STRIDE, and you are probably not able to declare Y as a three-dimensional array.)

To equivalence the X and Y arrays requires INC2X ≥ (2)(INC2Y) and INC3X ≥ (2)(INC3Y). Therefore, the values INC2X = (2)(INC2Y) = 34 and INC3X = (2)(INC3Y) = 2176 are set, and X is declared as a three-dimensional array, X(34,64,40).

The arrays are declared as follows:

```
      REAL*4     X(34,64,40)
      COMPLEX*8  Y(17,64,40)
      REAL*8     AUX(1)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```
      EQUIVALENCE (X,Y)
```

Note: Because NAUX= 0, this subroutine dynamically allocates the AX working storage.

Call Statement and Input:

```
              X  INC2X INC3X  Y  INC2Y INC3Y  N1   N2   N3 ISIGN SCALE   AUX   NAUX
              |    |     |    |    |     |     |    |    |   |     |      |     |
 CALL SRCFT3( X , 34 , 2176 , Y , 17 , 1088 , 32 , 64 , 40 , 1 , SCALE , AUX ,  0 )
```

SCALE = 1.0
X has 1.0 in location X(1,1,1) and 0.0 in all other locations.

Output:

Y has (1.0,0.0) in all locations.

SCRFT3 and DCRFT3 (Complex-to-Real Fourier Transform in Three Dimensions)

Purpose

These subroutines compute the three-dimensional discrete Fourier transform of complex conjugate even data in a three-dimensional array.

Table 232: Data Types.

Data Types		
X	Y, *scale*	**Subroutine**
Short-precision complex	Short-precision real	SCRFT2
Long-precision complex	Long-precision real	DCRFT2

Note:

1. For each use, only one invocation of this subroutine is necessary. The initialization phase, preparing the working storage, is a relatively small part of the total computation, so it is performed on each invocation.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SCRFT3 | DCRFT3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*)
C and C++	scrft3 | dcrft3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*);

On Entry

x

is the array X, containing the three-dimensional data to be transformed, where each element $x_{j1,j2,j3}$, using zero-based indexing, is stored in X(*j1*+*j2*(*inc2x*)+*j3*(*inc3x*)) for *j1* = 0, 1, ..., *n1*/2, *j2* = 0, 1, ..., *n2*-1, and *j3* = 0, 1, ..., *n3*-1. Due to complex conjugate symmetry, the input consists of only the first *n1*/2+1 values along the first dimension of the array, for *j1* = 0, 1, ..., *n1*/2. The strides for the elements in the first, second, and third dimensions are assumed to be 1, *inc2x*(≥ *n1*/2+1), and *inc3x*(≥ (*n2*)(*inc2x*)), respectively.

Specified as: an array, containing numbers of the data type indicated in Table 232 on page 1053. If the array is dimensioned X(LDA1,LDA2,LDA3), then LDA1 = *inc2x*, (LDA1)(LDA2) = *inc3x*, and LDA3 ≥ *n3*. For information on how to set up this array, see "Setting Up Your Data" on page 959. For more details, see "Notes" on page 1056.

inc2x

is the stride between the elements in array X for the second dimension.

Specified as: an integer; *inc2x* ≥ *n1*/2+1.

inc3x

is the stride between the elements in array X for the third dimension.

Specified as: an integer; *inc3x* ≥ (*n2*)(*inc2x*).

y

See On Return.

inc2y

is the stride between the elements in array Y for the second dimension.

Specified as: an integer; *inc2y* ≥ *n1*+2.

inc3y

is the stride between the elements in array Y for the third dimension.

Specified as: an integer; *inc3y* ≥ (*n2*)(*inc2y*).

n1

is the length of the first dimension of the three-dimensional data in the array to be transformed.

Specified as: an integer; *n1* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

n2

is the length of the second dimension of the three-dimensional data in the array to be transformed.

Specified as: an integer; $n2 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

n3

is the length of the third dimension of the three-dimensional data in the array to be transformed.

Specified as: an integer; $n3 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 956. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 56.

isign

controls the direction of the transform, determining the sign *Isign* of the exponents of W_{n1}, W_{n2}, and W_{n3}, where:

If *isign* = positive value, *Isign* = + (transforming time to frequency).

If *isign* = negative value, *Isign* = - (transforming frequency to time).

Specified as: an integer; $isign > 0$ or $isign < 0$.

scale

is the scaling constant *scale*. See "Function" on page 1057 for its usage.

Specified as: a number of the data type indicated in Table 232 on page 1053, where $scale > 0.0$ or $scale < 0.0$.

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is a storage work area used by this subroutine.

Specified as: an area of storage, containing *naux* long-precision real numbers. On output, the contents are overwritten.

naux

is the number of doublewords in the working storage specified in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SCRFT3 and DCRFT3 dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

On Return

y

is the array Y, containing the elements resulting from the three-dimensional discrete Fourier transform of the data in X. Each element $y_{k1,k2,k3}$, using zero-based indexing, is stored in $Y(k1+k2(inc2y)+k3(inc3y))$ for $k1 = 0, 1, ..., n1-1$, $k2 = 0, 1, ..., n2-1$, and $k3 = 0, 1, ..., n3-1$. The strides for the elements in the first, second, and third dimensions are assumed to be 1, $inc2y(\geq n1+2)$, and $inc3y(\geq (n2)(inc2y))$, respectively.

Returned as: an array, containing numbers of the data type indicated in Table 232 on page 1053. If the array is dimensioned `Y(LDA1,LDA2,LDA3)`, then LDA1 = *inc2y*, (LDA1)(LDA2) = *inc3y*, and LDA3 ≥ *n3*. For information on how to set up this array, see "Setting Up Your Data" on page 959. For more details, see "Notes" on page 1056.

Notes

1. If you specify the same array for X and Y, then *inc2y* must equal (2)(*inc2x*) and *inc3y* must equal (2)(*inc3x*). In this case, output overwrites input. If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
2. You should use "STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)" on page 1195 to determine the optimal values for the strides *inc2y* and *inc3y* for your output array. To obtain the best performance, you should use *inc2x* = *inc2y*/2 and *inc3x* = *inc3y*/2. Example 9 in the STRIDE subroutine description explains how it is used for these subroutines. For additional information on how to set up your data, see "Setting Up Your Data" on page 959.

Formulas

Processor-Independent Formulas for SCRFT3 for Calculating NAUX

Use the following formulas for calculating *naux*:

For 32-bit integer arguments:

1. If max($n2$, $n3$) < 252 and:

   If $n1 \le 16384$, use *naux* = 65000.
   If $n1 > 16384$, use *naux* = 60000+1.39$n1$.
2. If $n2 \ge 252$, $n3 < 252$, and:

   If $n1 \le 16384$, use *naux* = 65000+λ.
   If $n1 > 16384$, use *naux* = 60000+1.39$n1$+λ,

   where $\lambda = (n2+256)(s+2.28)$
   and $s = \min(64, 1+n1/2)$.
3. If $n2 < 252$, $n3 \ge 252$, and:

   If $n1 \le 16384$, use *naux* = 65000+ψ.
   If $n1 > 16384$, use *naux* = 60000+1.39$n1$+ψ,

   where $\psi = (n3+256)(s+2.28)$
   and $s = \min(64, (n2)(1+n1/2))$.
4. If $n2 \ge 252$ and $n3 \ge 252$, use the larger of the values calculated for cases 2 and 3 above.

For 64-bit integer arguments:

1. If max($n2$, $n3$) < 252 and:

   If $n1 \le 16384$, use *naux* = 95000.
   If $n1 > 16384$, use *naux* = 90000+1.39$n1$.
2. If $n2 \ge 252$, $n3 < 252$, and:

   If $n1 \le 16384$, use *naux* = 95000+λ.
   If $n1 > 16384$, use *naux* = 90000+1.39$n1$+λ,

   where $\lambda = (n2+256)(s+2.28)$
   and $s = \min(64, 1+n1/2)$.
3. If $n2 < 252$, $n3 \ge 252$, and:

   If $n1 \le 16384$, use *naux* = 95000+ψ.
   If $n1 > 16384$, use *naux* = 90000+1.39$n1$+ψ,

   where $\psi = (n3+256)(s+2.28)$
   and $s = \min(64, (n2)(1+n1/2))$.
4. If $n2 \ge 252$ and $n3 \ge 252$, use the larger of the values calculated for cases 2 and 3 above.

If *inc2y* or *inc3y* is an odd number, or if array Y is not aligned on a doubleword boundary, you should add the following amount to all the formulas given above:

(1+*n1*/2)(max(*n2*, *n3*))

Processor-Independent Formulas for DCRFT3 for NAUX

Use the following formulas for calculating *naux*:

For 32-bit integer arguments:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 4096, use *naux* = 62000.
   If *n1* > 4096, use *naux* = 60000+2.78*n1*.
2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 4096, use *naux* = 62000+λ.
   If *n1* > 4096, use *naux* = 60000+2.78*n1*+λ,
   where λ = ((2)*n2*+256)(*s*+4.56)
   and *s* = min(64, *n1*/2).
3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 4096, use *naux* = 62000+ψ.
   If *n1* > 4096, use *naux* = 60000+2.78*n1*+ψ,

   where ψ = ((2)*n3*+256)(*s*+4.56)
   and *s* = min(64, *n2*(1+*n1*/2)).
4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

For 64-bit integer arguments:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 4096, use *naux* = 92000.
   If *n1* > 4096, use *naux* = 90000+2.78*n1*.
2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 4096, use *naux* = 92000+λ.
   If *n1* > 4096, use *naux* = 90000+2.78*n1*+λ,

   where λ = ((2)*n2*+256)(*s*+4.56)
   and *s* = min(64, *n1*/2).
3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 4096, use *naux* = 92000+ψ.
   If *n1* > 4096, use *naux* = 90000+2.78*n1*+ψ,

   where ψ = ((2)*n3*+256)(*s*+4.56)
   and *s* = min(64, *n2*(1+*n1*/2)).
4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

Function

The three-dimensional discrete Fourier transform of complex conjugate even data in array X, with results going into array Y, is expressed as follows:

$$y_{k1,k2,k3} = scale \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \sum_{j3=0}^{n3-1} x_{j1,j2,j3} \; W_{n1}^{(Isign)j1k1} \; W_{n2}^{(Isign)j2k2} \; W_{n3}^{(Isign)j3k3}$$

for:

k1 = 0, 1, ..., *n1*-1
k2 = 0, 1, ..., *n2*-1
k3 = 0, 1, ..., *n3*-1

where:

$$W_{n1} = e^{-2\pi\left(\sqrt{-1}\right)/n1}$$
$$W_{n2} = e^{-2\pi\left(\sqrt{-1}\right)/n2}$$
$$W_{n3} = e^{-2\pi\left(\sqrt{-1}\right)/n3}$$

and where:

$x_{j1,j2,j3}$ are elements of array X.
$y_{k1,k2,k3}$ are elements of array Y.
Isign is + or - (determined by argument *isign*).
scale is a scalar value.

Because of the complex conjugate symmetry, the output in array Y is real. For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/((*n1*)(*n2*)(*n3*)) and *isign* being negative. See references [“1” on page 1253], [“5” on page 1253], [“6” on page 1253], [“27” on page 1254], and [“28” on page 1254].

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors
None

Input-Argument Errors

1. *n1* > 37748736
2. *n2* > 37748736
3. *n3* > 37748736
4. *inc2x* < *n1*/2+1
5. *inc3x* < (*n2*)(*inc2x*)
6. *inc2y* < *n1*+2
7. *inc3y* < (*n2*)(*inc2y*)
8. *scale* = 0.0
9. *isign* = 0
10. The length of one of the transforms in *n1*, *n2*, or *n3* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
11. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example

This example shows how to compute a three-dimensional transform. In this example, INC2Y = (2)(INC2X) and INC3Y = (2)(INC3X), so that the same array can be used for both input and output. The STRIDE subroutine is called to select good values for the INC2Y and INC3Y strides. (As explained below, STRIDE is not called for INC2X and INC3X.) Using the transform lengths (N1 = 32, N2 = 64, and N3 = 40) along with the output data type (short-precision real: 'S'), STRIDE is called once for each stride needed. First, it is called for INC2Y:

```
    CALL STRIDE (N2,N1+2,INC2Y,'S',0)
```

The output value returned for INC2Y is 34. (This value is equal to N1+2.) Then STRIDE is called again for INC3Y:

```
    CALL STRIDE (N3,N2*INC2Y,INC3Y,'S',0)
```

The output value returned for INC3Y is 2176. Because INC3Y is a multiple of INC2Y—that is, INC3Y = (N2)(INC2Y)—Y is declared as a three-dimensional array, Y(34,64,40). (In general, for larger arrays, these types of values for INC2Y and INC3Y are not returned by STRIDE, and you are probably not able to declare Y as a three-dimensional array.)

A good stride value for INC2X is INC2Y/2, and a good stride value for INC3X is INC3Y/2. Also, to equivalence the X and Y arrays requires INC2Y = (2)(INC2X) and INC3Y = (2)(INC3X). Therefore, the values INC2X = INC2Y/2 = 17 and INC3X = INC3Y/2 = 1088 are set, and X is declared as a three-dimensional array, X(17,64,40).

The arrays are declared as follows:

```
      COMPLEX*8  X(17,64,40)
      REAL*4     Y(34,64,40)
      REAL*8     AUX(1)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```
      EQUIVALENCE (X,Y)
```

Note: Because NAUX= 0, this subroutine dynamically allocates the AX working storage.

Call Statement and Input:

```
              X  INC2X INC3X  Y  INC2Y INC3Y  N1   N2   N3 ISIGN SCALE   AUX  NAUX
              |    |     |    |    |     |     |    |    |   |     |      |     |
CALL SCRFT3( X , 17 , 1088 , Y , 34 , 2176 , 32 , 64 , 40 , 1 , SCALE , AUX ,  0 )
```

SCALE = 1.0
X has (1.0,0.0) in location X(1,1,1) and (0.0,0.0) in all other locations.

Output:

Y has 1.0 in all locations.

Convolution and Correlation Subroutines

This contains the convolution and correlation subroutine descriptions.

SCON and SCOR (Convolution or Correlation of One Sequence with One or More Sequences)

Purpose

These subroutines compute the convolutions and correlations of a sequence with one or more sequences using a direct method. The input and output sequences contain short-precision real numbers.

Note: These subroutines are considered obsolete. They are provided in ESSL only for compatibility with earlier releases. You should use SCOND, SCORD, SDCON, SDCOR, SCONF, and SCORF instead, because they provide **better performance**. For further details, see "Convolution and Correlation Considerations" on page 960.

Syntax

Language	**Syntax**
Fortran	CALL SCON | SCOR (*init*, *h*, *inc1h*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nh*, *nx*, *m*, *iy0*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	scon | scor (*init*, *h*, *inc1h*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nh*, *nx*, *m*, *iy0*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init

is a flag, where:

If *init* ≠ 0, no computation is performed, error checking is performed, and the subroutine exits back to the calling program.

If *init* = 0, the convolutions or correlations of the sequence in *h* with the sequences in *x* are computed.

Specified as: an integer. It can have any value.

h

is the array H, consisting of the sequence of length N_h to be convolved or correlated with the sequences in array X.

Specified as: an array of (at least) length $1+(N_h-1)|inc1h|$, containing short-precision real numbers.

inc1h

is the stride between the elements within the sequence in array H.

Specified as: an integer; *inc1h* > 0.

x

is the array X, consisting of *m* input sequences of length N_x, each to be convolved or correlated with the sequence in array H.

Specified as: an array of (at least) length $1 + (m-1)inc2x + (N_x-1)inc1x$, containing short-precision real numbers.

inc1x

is the stride between the elements within each sequence in array X.

Specified as: an integer; *inc1x* > 0.

inc2x

is the stride between the first elements of the sequences in array X.

Specified as: an integer; *inc2x* > 0.

y

See On Return.

inc1y
is the stride between the elements within each sequence in output array Y.

Specified as: an integer; *inc1y* > 0.

inc2y
is the stride between the first elements of each sequence in output array Y.

Specified as: an integer; *inc2y* > 0.

nh
is the number of elements, N_h, in the sequence in array H.

Specified as: an integer; $N_h > 0$.

nx
is the number of elements, N_x, in each sequence in array X.

Specified as: an integer; $N_x > 0$.

m
is the number of sequences in array X to be convolved or correlated.

Specified as: an integer; $m > 0$.

iy0
is the convolution or correlation index of the element to be stored in the first position of each sequence in array Y.

Specified as: an integer. It can have any value.

ny
is the number of elements, N_y, in each sequence in array Y.

Specified as: an integer; $N_y > 0$ for SCON and $N_y \geq -N_h+1$ for SCOR.

aux1
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

naux1
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

aux2
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

naux2
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

On Return

y
is array Y, consisting of *m* output sequences of length N_y that are the result of the convolutions or correlations of the sequence in array H with the sequences in array X. Returned as: an array of (at least) length $1 + (m-1)inc2y + (N_y-1)inc1y$, containing short-precision real numbers.

Notes

1. Output should not overwrite input; that is, input arrays X and H must have no common elements with output array Y. Otherwise, results are unpredictable. See "Vector concepts" on page 71.
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* ≠ 0 and *init* = 0.

3. Auxiliary storage is not needed, but the arguments *aux1*, *naux1*, *aux2*, and *naux2* must still be specified. You can assign any values to these arguments.

Function

The convolutions and correlations of a sequence in array H with one or more sequences in array X are expressed as follows:

Convolutions for SCON:

$$y_{ki} = \sum_{j=\max(0,k-N_x+1)}^{\min(N_h-1,k)} h_j x_{k-j,i}$$

Correlations for SCOR:

$$y_{ki} = \sum_{j=\max(0,-k)}^{\min(N_h-1,\ N_x-1-k)} h_j x_{k+j,i}$$

for:

$k = iy0, iy0+1, \ldots, iy0+N_y-1$
$i = 1, 2, \ldots, m$

where:

y_{ki} are elements of the *m* sequences of length N_y in array Y.

x_{ki} are elements of the *m* sequences of length N_x in array X.

h_j are elements of the sequence of length N_h in array H.

iy0 is the convolution or correlation index of the element to be stored in the first position of each sequence in array Y.

min and max select the minimum and maximum values, respectively.

It is assumed that elements outside the range of definition are zero. See references ["25" on page 1254] and ["104" on page 1258].

Only one invocation of this subroutine is needed:

1. You do not need to invoke the subroutine with *init* ≠ 0. If you do, however, the subroutine performs error checking, exits back to the calling program, and no computation is performed.
2. With *init* = 0, the subroutine performs the calculation of the convolutions or correlations.

Error conditions

Computational Errors
None

Input-Argument Errors

1. *nh*, *nx*, *ny*, or *m* ≤ 0
2. *inc1h*, *inc1x*, *inc2x*, *inc1y*, or *inc2y* ≤ 0

Examples

Example 1

This example shows how to compute a convolution of a sequence in H, which is a ramp function, and three sequences in X, a triangular function and its cyclic translates. It computes the full range of nonzero values of the convolution plus two extra points, which are set to 0. The arrays are declared as follows:

```
     REAL*4   H(0:4999), X(0:49999), Y(0:49999)
     REAL*8   AUX1, AUX2
```

Call Statement and Input:

```
          INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH NX  M IY0 NY  AUX1 NAUX1 AUX2 NAUX2
            |    |   |   |   |     |   |   |     |    |  |  |  |   |   |      |     |     |
CALL SCON(INIT, H , 1 , X , 1 ,  10 , Y , 1 ,  15 , 4, 10, 3, 0, 15, AUX1 , 0 , AUX2 , 0)
```

`INIT = 0`(for computation)
`H = (1.0, 2.0, 3.0, 4.0)`

X contains the following three sequences:

```
1.0  2.0  3.0
2.0  1.0  2.0
3.0  2.0  1.0
4.0  3.0  2.0
5.0  4.0  3.0
6.0  5.0  4.0
5.0  6.0  5.0
4.0  5.0  6.0
3.0  4.0  5.0
2.0  3.0  4.0
```

Output:

Y contains the following three sequences:

```
 1.0   2.0   3.0
 4.0   5.0   8.0
10.0  10.0  14.0
20.0  18.0  22.0
30.0  20.0  18.0
40.0  30.0  20.0
48.0  40.0  30.0
52.0  48.0  40.0
50.0  52.0  48.0
40.0  50.0  52.0
29.0  38.0  47.0
18.0  25.0  32.0
 8.0  12.0  16.0
 0.0   0.0   0.0
 0.0   0.0   0.0
```

Example 2

This example shows how the output from Example 1 differs when the values for NY and *inc2y* are 10 rather than 15. The output is the same except that it consists of only the first 10 values produced in Example 1.

Output:

Y contains the following three sequences:

```
 1.0   2.0   3.0
 4.0   5.0   8.0
10.0  10.0  14.0
20.0  18.0  22.0
30.0  20.0  18.0
40.0  30.0  20.0
48.0  40.0  30.0
52.0  48.0  40.0
```

```
50.0  52.0  48.0
40.0  50.0  52.0
```

Example 3

This example shows how the output from Example 2 differs if the value for `IY0` is 3 rather than 0. The output is the same except it starts at element 3 of the convolution sequences rather than element 0.

Output:

Y contains the following three sequences:

```
20.0  18.0  22.0
30.0  20.0  18.0
40.0  30.0  20.0
48.0  40.0  30.0
52.0  48.0  40.0
50.0  52.0  48.0
40.0  50.0  52.0
29.0  38.0  47.0
18.0  25.0  32.0
 8.0  12.0  16.0
```

Example 4

This example shows how to compute a correlation of a sequence in H, which is a ramp function, and three sequences in X, a triangular function and its cyclic translates. It computes the full range of nonzero values of the correlation plus two extra points, which are set to 0. The arrays are declared as follows:

```
     REAL*4   H(0:4999), X(0:49999), Y(0:49999)
     REAL*8   AUX1, AUX2
```

Call Statement and Input:

```
          INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH NX  M IY0  NY  AUX1  NAUX1 AUX2 NAUX2
           |    |   |   |   |     |   |   |     |   |  |   |   |    |   |       |    |     |
CALL SCOR(INIT, H , 1 , X , 1  , 10 , Y , 1  , 15 , 4, 10, 3, -3, 15, AUX1 ,  0 , AUX2 , 0)
```

`INIT = 0`(for computation)
`H = (1.0, 2.0, 3.0, 4.0)`

X contains the following three sequences:

```
1.0  2.0  3.0
2.0  1.0  2.0
3.0  2.0  1.0
4.0  3.0  2.0
5.0  4.0  3.0
6.0  5.0  4.0
5.0  6.0  5.0
4.0  5.0  6.0
3.0  4.0  5.0
2.0  3.0  4.0
```

Output:

Y contains the following three sequences:

```
 4.0   8.0  12.0
11.0  10.0  17.0
20.0  15.0  16.0
30.0  22.0  18.0
40.0  30.0  22.0
50.0  40.0  30.0
52.0  50.0  40.0
48.0  52.0  50.0
40.0  48.0  52.0
30.0  40.0  48.0
16.0  22.0  28.0
 7.0  10.0  13.0
 2.0   3.0   4.0
```

```
  0.0   0.0   0.0
  0.0   0.0   0.0
```

Example 5

This example shows how the output from Example 4 differs when the values for `NY` and `INC2Y` are 10 rather than 15. The output is the same except that it consists of only the first 10 values produced in Example 4.

Output:

Y contains the following three sequences:

```
  4.0   8.0  12.0
 11.0  10.0  17.0
 20.0  15.0  16.0
 30.0  22.0  18.0
 40.0  30.0  22.0
 50.0  40.0  30.0
 52.0  50.0  40.0
 48.0  52.0  50.0
 40.0  48.0  52.0
 30.0  40.0  48.0
```

Example 6

This example shows how the output from Example 5 differs if the value for `IY0` is 0 rather than -3. The output is the same except it starts at element 0 of the correlation sequences rather than element -3.

Output:

Y contains the following three sequences:

```
 30.0  22.0  18.0
 40.0  30.0  22.0
 50.0  40.0  30.0
 52.0  50.0  40.0
 48.0  52.0  50.0
 40.0  48.0  52.0
 30.0  40.0  48.0
 16.0  22.0  28.0
  7.0  10.0  13.0
  2.0   3.0   4.0
```

SCOND and SCORD (Convolution or Correlation of One Sequence with Another Sequence Using a Direct Method)

Purpose

These subroutines compute the convolution and correlation of a sequence with another sequence using a direct method. The input and output sequences contain short-precision real numbers.

Note:

1. These subroutines compute the convolution and correlation using direct methods. In most cases, these subroutines provide **better performance** than using SCON or SCOR, if you determine that SCON or SCOR would have used a direct method for its computation. For information on how to make this determination, see reference [“5” on page 1253].
2. For long-precision data, you should use DDCON or DDCOR with the decimation rate, *id*, equal to 1.
3. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 34.

Syntax

Language	Syntax
Fortran	CALL SCOND | SCORD (*h*, *inch*, *x*, *incx*, *y*, *incy*, *nh*, *nx*, *iy0*, *ny*)

Language	Syntax
C and C++	scond | scord (*h*, *inch*, *x*, *incx*, *y*, *incy*, *nh*, *nx*, *iy0*, *ny*);

On Entry

h

is the array H, consisting of the sequence of length N_h to be convolved or correlated with the sequence in array X.

Specified as: an array of (at least) length 1+(N_h-1)|*inch*|, containing short-precision real numbers.

inch

is the stride between the elements within the sequence in array H.

Specified as: an integer; *inch* > 0 or *inch* < 0.

x

is the array X, consisting of the input sequence of length N_x, to be convolved or correlated with the sequence in array H.

Specified as: an array of (at least) length 1+(N_x-1)|*incx*|, containing short-precision real numbers.

incx

is the stride between the elements within the sequence in array X.

Specified as: an integer; *incx* > 0 or *incx* < 0.

y

See On Return.

incy

is the stride between the elements within the sequence in output array Y.

Specified as: an integer; *incy* > 0 or *incy* < 0.

nh

is the number of elements, N_h, in the sequence in array H.

Specified as: an integer; N_h > 0.

nx

is the number of elements, N_x, in the sequence in array X.

Specified as: an integer; N_x > 0.

iy0

is the convolution or correlation index of the element to be stored in the first position of the sequence in array Y.

Specified as: an integer. It can have any value.

ny

is the number of elements, N_y, in the sequence in array Y.

Specified as: an integer; N_y > 0.

On Return

y

is the array Y of length N_y, consisting of the output sequence that is the result of the convolution or correlation of the sequence in array H with the sequence in array X. Returned as: an array of (at least) length 1+(N_y-1)|*incy*|, containing short-precision real numbers.

Notes

1. Output should not overwrite input—that is, input arrays X and H must have no common elements with output array Y. Otherwise, results are unpredictable. See “Vector concepts” on page 71.

2. If *iy0* and *ny* are such that output outside the basic range is needed, where the basic range is 0 ≤ *k* ≤ (*nh*+*nx*-2) for SCOND and (-*nh*+1) ≤ *k* ≤ (*nx*-1) for SCORD, the subroutine stores zeros using scalar code. It is not efficient to store many zeros in this manner. It is more efficient to set *iy0* and *ny* so that the output is produced within the above range of *k* values.

Function

The convolution and correlation of a sequence in array H with a sequence in array X are expressed as follows:

Convolution for SCOND:

$$y_k = \sum_{j=\max(0,k-N_x+1)}^{\min(N_h-1,k)} h_j x_{k-j}$$

Correlation for SCORD:

$$y_k = \sum_{j=\max(0,-k)}^{\min(N_h-1,\ N_x-1-k)} h_j x_{k+j}$$

for k = *iy0*, *iy0*+1, ..., *iy0*+N_y-1

where:

y_k are elements of the sequence of length N_y in array Y.

x_k are elements of the sequence of length N_x in array X.h_j are elements of the sequence of length N_h in array H.

iy0 is the convolution or correlation index of the element to be stored in the first position of each sequence in array Y.

min and max select the minimum and maximum values, respectively.

It is assumed that elements outside the range of definition are zero. See reference [“5” on page 1253].

Special Usage

SCORD can also perform the functions of SCON and SACOR; that is, it can compute convolutions and autocorrelations. To compute a convolution, you must specify a negative stride for H (see Example 9). To compute the autocorrelation, you must specify the two input sequences to be the same (see Example 10). In fact, you can also compute the autoconvolution by using both of these techniques together, letting the two input sequences be the same, and specifying a negative stride for the first input sequence.

Error conditions

Computational Errors
None

Input-Argument Errors

1. *nh*, *nx*, or *ny* ≤ 0
2. *inch*, *incx*, or *incy* = 0

Examples

Example 1

This example shows how to compute a convolution of a sequence in H with a sequence in X, where both sequences are ramp functions.

Call Statement and Input:

```
            H  INCH  X  INCX  Y  INCY  NH  NX IY0  NY
            |   |    |   |    |   |    |   |   |   |
CALL SCOND( H , 1  , X , 1  , Y , 1  , 4 , 8 , 0 , 11 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output:

```
Y        =  (11.0, 34.0, 70.0, 120.0, 130.0, 140.0, 150.0, 160.0,
             151.0, 122.0, 72.0)
```

Example 2

This example shows how the output from Example 1 differs when the value for IY0 is -2 rather than 0, and NY is 15 rather than 11. The output has two zeros at the beginning and end of the sequence, for points outside the range of nonzero output.

Call Statement and Input:

```
            H  INCH  X  INCX  Y  INCY  NH  NX  IY0  NY
            |   |    |   |    |   |    |   |    |   |
CALL SCOND( H , 1  , X , 1  , Y , 1  , 4 , 8 , -2 , 15 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output:

```
Y        =  (0.0, 0.0, 11.0, 34.0, 70.0, 120.0, 130.0, 140.0, 150.0,
             160.0, 151.0, 122.0, 72.0, 0.0, 0.0)
```

Example 3

This example shows how the same output as Example 1 can be obtained when H and X are interchanged, because the convolution is symmetric in H and X. (The arguments are switched in the calling sequence.)

Call Statement and Input:

```
            H  INCH  X  INCX  Y  INCY  NH  NX IY0  NY
            |   |    |   |    |   |    |   |   |   |
CALL SCOND( X , 1  , H , 1  , Y , 1  , 4 , 8 , 0 , 11 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output:

```
Y        =  (11.0, 34.0, 70.0, 120.0, 130.0, 140.0, 150.0, 160.0,
             151.0, 122.0, 72.0)
```

Example 4

This example shows how the output from Example 1 differs when a negative stride is specified for the sequence in H. By reversing the H sequence, the correlation is computed.

Call Statement and Input:

```
            H  INCH  X  INCX  Y  INCY  NH  NX IY0  NY
            |    |   |   |    |   |    |   |   |   |
CALL SCOND( H , -1 , X , 1  , Y , 1  , 4 , 8 , 0 , 11 )
```

```
H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output:

```
Y        =  (44.0, 81.0, 110.0, 130.0, 140.0, 150.0, 160.0, 170.0,
             104.0, 53.0, 18.0)
```

Example 5

This example shows how to compute the autoconvolution of a sequence by letting the two input sequences for H and X be the same. (X is specified for both arguments in the calling sequence.)

Call Statement and Input:

```
            H  INCH  X  INCX  Y  INCY NH  NX IY0  NY
            |   |   |   |    |   |    |   |   |   |
CALL SCOND( X , 1 , X , 1  , Y , 1  , 4 , 4 , 0 , 7 )

X        =  (11.0, 12.0, 13.0, 14.0)
```

Output:

```
Y        =  (121.0, 264.0, 430.0, 620.0, 505.0, 364.0, 196.0)
```

Example 6

This example shows how to compute a correlation of a sequence in H with a sequence in X, where both sequences are ramp functions.

Call Statement and Input:

```
            H  INCH  X  INCX  Y  INCY NH  NX   IY0  NY
            |   |    |   |    |   |    |   |    |   |
CALL SCORD( H , 1  , X , 1  , Y , 1  , 4 , 8 , -3 , 11 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output:

```
Y        =  (44.0, 81.0, 110.0, 130.0, 140.0, 150.0, 160.0, 170.0,
             104.0, 53.0, 18.0)
```

Example 7

This example shows how the output from Example 6 differs when the value for `IY0` is -5 rather than -3 and NY is 15 rather than 11. The output has two zeros at the beginning and end of the sequence, for points outside the range of nonzero output.

Call Statement and Input:

```
            H  INCH  X  INCX  Y  INCY NH  NX   IY0  NY
            |   |    |   |    |   |    |   |    |   |
CALL SCORD( H , 1  , X , 1  , Y , 1  , 4 , 8 , -5 , 15 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output:

```
Y        =  (0.0, 0.0, 44.0, 81.0, 110.0, 130.0, 140.0, 150.0, 160.0,
             170.0, 104.0, 53.0, 18.0, 0.0, 0.0)
```

Example 8

This example shows how the output from Example 6 differs when H and X are interchanged (in the calling sequence). The output sequence is the reverse of that in Example 6. To get the full range of output, `IY0` is set to -NX+1.

Call Statement and Input:

```
             H  INCH  X  INCX  Y  INCY NH  NX   IY0  NY
             |   |    |   |    |   |    |   |    |   |
CALL SCORD( X , 1  , H , 1  , Y , 1  , 4 , 8 , -7 , 11 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output:

```
Y        =  (18.0, 53.0, 104.0, 170.0, 160.0, 150.0, 140.0, 130.0,
             110.0, 81.0, 44.0)
```

Example 9

This example shows how the output from Example 6 differs when a negative stride is specified for the sequence in H. By reversing the H sequence, the convolution is computed.

Call Statement and Input:

```
             H  INCH  X  INCX  Y  INCY  NH  NX  IY0  NY
             |    |   |   |    |   |    |   |    |   |
CALL SCORD( H , -1 , X , 1  , Y , 1  , 4 , 8 , -3 , 11 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output:

```
Y        =  (11.0, 34.0, 70.0, 120.0, 130.0, 140.0, 150.0, 160.0,
             151.0, 122.0, 72.0)
```

Example 10

This example shows how to compute the autocorrelation of a sequence by letting the two input sequences for H and X be the same. (X is specified for both arguments in the calling sequence.)

Call Statement and Input:

```
             H  INCH  X  INCX  Y  INCY  NH  NX  IY0  NY
             |   |    |   |    |   |    |   |    |   |
CALL SCORD( X , 1  , X , 1  , Y , 1  , 4 , 4 , -3 , 7 )

X        =  (11.0, 12.0, 13.0, 14.0)
```

Output:

```
Y        =  (154.0, 311.0, 470.0, 630.0, 470.0, 311.0, 154.0)
```

SCONF and SCORF (Convolution or Correlation of One Sequence with One or More Sequences Using the Mixed-Radix Fourier Method)

Purpose

These subroutines compute the convolutions and correlations, respectively, of a sequence with one or more sequences using the mixed-radix Fourier method. The input and output sequences contain short-precision real numbers.

Note:

1. Two invocations of these subroutines are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 34.

Syntax

Language	Syntax
Fortran	CALL SCONF | SCORF (*init*, *h*, *inc1h*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nh*, *nx*, *m*, *iy0*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	sconf | scorf (*init*, *h*, *inc1h*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nh*, *nx*, *m*, *iy0*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init

is a flag, where:

If *init* ≠ 0, trigonometric functions, the transform of the sequence in *h*, and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the convolutions or correlations of the sequence that was in *h* at initialization with the sequences in *x* are computed. *h* is not used or changed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: an integer. It can have any value.

h

is the array H, consisting of the sequence of length N_h to be convolved or correlated with the sequences in array X.

Specified as: an array of (at least) length 1+(N_h-1)|*inc1h*|, containing short-precision real numbers.

inc1h

is the stride between the elements within the sequence in array H.

Specified as: an integer; *inc1h* > 0.

x

is the array X, consisting of *m* input sequences of length N_x, each to be convolved or correlated with the sequence in array H.

Specified as: an array of (at least) length 1+(N_x-1)*inc1x*+(*m*-1)*inc2x*, containing short-precision real numbers.

inc1x

is the stride between the elements within each sequence in array X.

Specified as: an integer; *inc1x* > 0.

inc2x

is the stride between the first elements of the sequences in array X.

Specified as: an integer; *inc2x* > 0.

y

See On Return.

inc1y

is the stride between the elements within each sequence in output array Y.

Specified as: an integer; *inc1y* > 0.

inc2y

is the stride between the first elements of each sequence in output array Y.

Specified as: an integer; *inc2y* > 0.

nh

is the number of elements, N_h, in the sequence in array H.

Specified as: an integer; $N_h > 0$.

nx
is the number of elements, N_x, in each sequence in array X.

Specified as: an integer; $N_x > 0$.

m
is the number of sequences in array X to be convolved or correlated.

Specified as: an integer; $m > 0$.

iy0
is the convolution or correlation index of the element to be stored in the first position of each sequence in array Y.

Specified as: an integer. It can have any value.

ny
is the number of elements, N_y, in each sequence in array Y.

Specified as: an integer; $N_y > 0$.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, the working storage is computed.

If *init* = 0, the working storage is used in the computation of the convolutions.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*.

Specified as: an integer; *naux1* > 23 (32-bit integer arguments) or 45 (64-bit integer arguments) and *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 23 (32-bit integer arguments) or 45 (64-bit integer arguments) and the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

aux2
has the following meaning:

If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*.

Specified as: an integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, SCONF and SCORF dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 49.

On Return

y
has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of *m* output sequences of length N_y that are the result of the convolutions or correlations of the sequence in array H with the sequences in array X.

Returned as: an array of (at least) length 1+(N_y-1)*inc1y*+(*m*-1)*inc2y*, containing short-precision real numbers.

aux1

is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* ≠ 0 and *init* = 0.
3. If you specify the same array for X and Y, then *inc1x* and *inc1y* must be equal, and *inc2x* and *inc2y* must be equal. In this case, output overwrites input.
4. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.
5. If *iy0* and *ny* are such that output outside the basic range is needed, the subroutine stores zeros. These ranges are: $0 \le k \le N_x+N_h-2$ for SCONF and $1-N_h \le k \le N_x-1$ for SCORF.

Formulas

Formulas for the Length of the Fourier Transform

Before calculating the necessary sizes of *naux1* and *naux2*, you must determine the length *n* of the Fourier transform. The value of *n* is based on *nf*. You can use one of two techniques to determine *nf*:

- Use the simple overestimate of *nf* = *nx*+*nh*-1. (If *iy0* = 0 and *ny* > *nh*+*nx*, this is the actual value, not an overestimate.)
- Use the values of the arguments *iy0*, *nh*, *nx*, and *ny* inserted into the following formulas to get a value for the variable *nf*:

  iy0p = max(*iy0*, 0)
  ix0 = max((*iy0p*+1)-*nh*, 0)
  ih0 = max((*iy0p*+1)-*nx*, 0)
  nd = *ix0*+*ih0*
  n1 = *iy0*+*ny*
  nxx = min(*n1*, *nx*)-*ix0*
  nhh = min(*n1*, *nh*)-*ih0*
  ntt = *nxx*+*nhh*-1
  nn1 = *n1*-*nd*
  iyy0 = *iy0p*-*nd*
  nzleft = max(0, *nhh*-*iyy0*-1)
  nzrt = min(*nn1*, *ntt*)-*nxx*
  nf = max(12, *nxx*+max(*nzleft*, *nzrt*))

After calculating the value for *nf*, using one of these two techniques, refer to the formula or table of allowable values of *n* in "Acceptable Lengths for the Transforms" on page 956, selecting the value equal to or greater than *nf*.

Processor-Independent Formulas for NAUX1 and NAUX2

The required values of *naux1* and *naux2* depend on the value determined for *n* in Formulas for the Length of the Fourier Transform.

NAUX1 Formulas

For 32-bit integer arguments:

If $n \le 16384$, use $naux1 = 58000$.
If $n > 16384$, use $naux1 = 40000+2.14n$.

For 64-bit integer arguments:

If $n \le 16384$, use $naux1 = 78000$.
If $n > 16384$, use $naux1 = 60000+2.14n$.

NAUX2 Formulas

If $n \le 16384$, use $naux2 = 30000$.
If $n > 16384$, use $naux2 = 20000+1.07n$.

Function

The convolutions and correlations of a sequence in array H with one or more sequences in array X are expressed as follows.

Convolutions for SCONF:

$$y_{ki} = \sum_{j=\max(0,k-N_x+1)}^{\min(N_h-1,k)} h_j x_{k-j,i}$$

Correlations for SCORF:

$$y_{ki} = \sum_{j=\max(0,-k)}^{\min(N_h-1,\ N_x-1-k)} h_j x_{k+j,i}$$

for:

$k = iy0, iy0+1, \ldots, iy0+N_y-1$
$i = 1, 2, \ldots, m$

where:

y_{ki} are elements of the *m* sequences of length N_y in array Y.

x_{ki} are elements of the *m* sequences of length N_x in array X.

h_j are elements of the sequence of length N_h in array H.

iy0 is the convolution or correlation index of the element to be stored in the first position of each sequence in array Y.

min and max select the minimum and maximum values, respectively.

These subroutines use a Fourier transform method with a mixed-radix capability. This provides maximum performance for your application. The length of the transform, *n*, that you must calculate to determine the correct sizes for *naux1* and *naux2* is the same length used by the Fourier transform subroutines called by this subroutine. It is assumed that elements outside the range of definition are zero. See references [“25” on page 1254] and [“104” on page 1258].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the convolutions.

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors
None

Input-Argument Errors

1. *nh*, *nx*, *ny*, or *m* ≤ 0
2. *inc1h*, *inc1x*, *inc2x*, *inc1y*, or *inc2y* ≤ 0
3. The resulting internal Fourier transform length *n*, is too large. See "Convolutions and Correlations by Fourier Methods" on page 961.
4. The subroutine has not been initialized with the present arguments.
5. *naux1* ≤ 23
6. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
7. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows how to compute a convolution of a sequence in H, where H and X are ramp functions. It calculates all nonzero values of the convolution of the sequences in H and X. The arrays are declared as follows:

```
     REAL*4   H(8), X(10,1), Y(17)
```

Because this convolution is symmetric in H and X, you can interchange the H and X sequences, leaving all other arguments the same, and you get the same output shown below. First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT   H INC1H X INC1X INC2X Y INC1Y INC2Y NH   NX   M IY0 NY   AUX1 NAUX1 AUX2 NAUX2
             |    |   |   |   |     |   |   |     |    |    |   |  |   |    |     |     |     |
CALL SCONF(INIT, H , 1 , X , 1   ,   1 , Y,  1   ,  1   , 8, 10, 1, 0, 17, AUX1, 128, AUX2,  0)
```

`INIT = 1`(for initialization)
`INIT = 0`(for computation)
`H = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0)`
`X = (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 20.0)`

Output:

```
Y        =  (11.0, 34.0, 70.0, 120.0, 185.0, 266.0, 364.0, 480.0,
             516.0, 552.0, 567.0, 560.0, 530.0, 476.0, 397.0, 292.0,
             160.0)
```

Example 2

This example shows how the output from Example 1 differs when the value for NY is 21 rather than 17, and the value for `IY0` is -2 rather than 0. This yields two zeros on each end of the convolution.

Output:

```
Y        =  (0.0, 0.0, 11.0, 34.0, 70.0, 120.0, 185.0, 266.0, 364.0,
             480.0, 516.0, 552.0, 567.0, 560.0, 530.0, 476.0, 397.0,
             292.0, 160.0, 0.0, 0.0)
```

Example 3

This example shows how to compute the autoconvolution by letting the two input sequences be the same for Example 2. First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH  NX  M  IY0  NY  AUX1 NAUX1 AUX2
NAUX2
            |    |   |   |   |     |   |   |     |    |   |  |   |    |   |     |    |     |
CALL SCONF(INIT, H , 1 , H , 1  ,  1 , Y,  1  ,  1  , 8, 10, 1, -2, 21, AUX1, 128, AUX2,
0)
```

`INIT` = `1`(for initialization)
`INIT` = `0`(for computation)

Output:

```
Y        =  (1.0, 4.0, 10.0, 20.0, 35.0, 56.0, 84.0, 120.0, 147.0,
             164.0, 170.0, 164.0, 145.0, 112.0, 64.0)
```

Example 4

This example shows how to compute all nonzero values of the convolution of the sequence in H with the two sequences in X. First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH  NX  M IY0 NY  AUX1 NAUX1 AUX2 NAUX2
            |    |   |   |   |     |   |   |     |    |   |  |  |   |   |     |    |     |
CALL SCONF(INIT, H , 1 , X,  1  , 10 , Y,  1  , 17  , 8, 10, 2, 0, 17, AUX1, 148, AUX2,  0)
```

`INIT` = `1`(for initialization)
`INIT` = `0`(for computation)
`H = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0)`

X contains the following two sequences:

```
11.0  12.0
12.0  13.0
13.0  14.0
14.0  15.0
15.0  16.0
16.0  17.0
17.0  18.0
18.0  19.0
19.0  20.0
20.0  11.0
```

Output:

Y contains the following two sequences:

```
 11.0   12.0
 34.0   37.0
 70.0   76.0
120.0  130.0
185.0  200.0
266.0  287.0
364.0  392.0
480.0  516.0
516.0  552.0
552.0  578.0
567.0  582.0
560.0  563.0
530.0  520.0
476.0  452.0
397.0  358.0
292.0  237.0
160.0   88.0
```

Example 5

This example shows how to compute a correlation of a sequence in H, where H and X are ramp functions. It calculates all nonzero values of the correlation of the sequences in H and X. The arrays are declared as follows:

```
      REAL*4   H(8), X(10,1)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH  NX  M  IY0 NY  AUX1 NAUX1 AUX2
NAUX2
             |    |   |   |   |     |   |   |     |    |   |  |   |   |   |     |     |     |
CALL SCORF(INIT, H , 1 , X,  1  ,  1 , Y,  1  ,  1  , 8, 10, 1, -7, 17, AUX1, 128, AUX2,  0)
```

INIT = 1(for initialization)
INIT = 0(for computation)
H =(same as input H in Example 1)
X =(same as input X in Example 1)

Output:

```
Y        =  (88.0, 173.0, 254.0, 330.0, 400.0, 463.0, 518.0, 564.0,
             600.0, 636.0, 504.0, 385.0, 280.0, 190.0, 116.0,
             59.0, 20.0)
```

Example 6

This example shows how the output from Example 5 differs when the value for NY is 21 rather than 17, and the value for IY0 is -9 rather than 0. This yields two zeros on each end of the correlation.

Output:

```
Y        =  (0.0, 0.0, 88.0, 173.0, 254.0, 330.0, 400.0, 463.0, 518.0,
             564.0, 600.0, 636.0, 504.0, 385.0, 280.0, 190.0, 116.0,
             59.0, 20.0, 0.0, 0.0)
```

Example 7

This example shows the effect of interchanging H and X. It uses the same input as Example 5, with H and X switched in the calling sequence, and with IY0 with a value of -9. Unlike convolution, as noted in Example 1, the correlation is not symmetric in H and X. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH  NX  M  IY0 NY  AUX1 NAUX1 AUX2
NAUX2
            |    |   |   |   |     |   |   |     |    |   |  |   |   |   |     |    |     |
CALL SCONF(INIT, X , 1 , H,  1  ,  1 , Y,  1  ,  1  , 8, 10, 1, -9, 17, AUX1, 128, AUX2,  0)
```

INIT = 1(for initialization)
INIT = 0(for computation)

Output:

```
Y        =  (20.0, 59.0, 116.0, 190.0, 280.0, 385.0, 504.0, 636.0,
             600.0, 564.0, 518.0, 463.0, 400.0, 330.0, 254.0, 173.0,
             88.0)
```

Example 8

This example shows how to compute the autocorrelation by letting the two input sequences be the same. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation. Because there is only one H input sequence, only one autocorrelation can be computed. Furthermore, this usage does not take advantage of the fact that the output is symmetric. Therefore, you should use SACORF to compute autocorrelations, because it does not have either of these problems.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH NX M  IY0 NY  AUX1 NAUX1 AUX2 NAUX2
            |    |   |   |   |     |   |   |     |   |  |  |   |   |   |     |    |     |
CALL SCONF(INIT, H , 1 , H,  1  ,  1 , Y,  1  ,  1 , 8, 8, 1, -7, 15, AUX1, 148, AUX2,  0)
```

INIT = 1(for initialization)
INIT = 0(for computation)

Output:

```
Y        =  (8.0, 23.0, 44.0, 70.0, 100.0, 133.0, 168.0, 204.0, 168.0,
             133.0, 100.0 , 70.0, 44.0, 23.0, 8.0)
```

Example 9

This example shows how to compute all nonzero values of the correlation of the sequence in H with the two sequences in X. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
           INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH  NX  M  IY0 NY  AUX1 NAUX1 AUX2
NAUX2
            |    |   |   |   |     |   |   |     |    |   |  |   |   |   |     |    |     |
CALL SCONF(INIT, H , 1 , X,  1  , 10 , Y,  1  , 17  , 8, 10, 2, -7, 17, AUX1, 148, AUX2,  0)
```

INIT = 1(for initialization)
INIT = 0(for computation)
H =(same as input H in Example 4)
X =(same as input X in Example 4)

Output:

Y contains the following two sequences:

```
 88.0   96.0
173.0  188.0
254.0  275.0
330.0  356.0
400.0  430.0
463.0  496.0
518.0  553.0
564.0  600.0
600.0  636.0
636.0  592.0
504.0  462.0
385.0  346.0
280.0  245.0
190.0  160.0
116.0   92.0
 59.0   42.0
 20.0   11.0
```

SDCON, DDCON, SDCOR, and DDCOR (Convolution or Correlation with Decimated Output Using a Direct Method)

Purpose

These subroutines compute the convolution and correlation of a sequence with another sequence, with decimated output, using a direct method.

Table 233: Data Types.

Data Types	
h*, *x*, *y	**Subroutine**
Short-precision real	SDCON
Long-precision real	DDCON
Short-precision real	SDCOR
Long-precision real	DDCOR

Note:

1. These subroutines are the short- and long-precision equivalents of SCOND and SCORD when the decimation interval *id* is equal to 1. Because there is no long-precision version of SCOND and SCORD, you can use DDCON and DDCOR, respectively, with decimation interval *id* = 1 to perform the same function.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SDCON | DDCON | SDCOR | DDCOR (*h*, *inch*, *x*, *incx*, *y*, *incy*, *nh*, *nx*, *iy0*, *ny*, *id*)
C and C++	sdcon | ddcon | sdcor | ddcor (*h*, *inch*, *x*, *incx*, *y*, *incy*, *nh*, *nx*, *iy0*, *ny*, *id*);

On Entry

h

is the array H, consisting of the sequence of length N_h to be convolved or correlated with the sequence in array X.

Specified as: an array of (at least) length 1+(N_h-1)|*inch*|, containing numbers of the data type indicated in Table 233 on page 1079.

inch
is the stride between the elements within the sequence in array H.

Specified as: an integer; *inch* > 0 or *inch* < 0.

x
is the array X, consisting of the input sequence of length N_x, to be convolved or correlated with the sequence in array H.

Specified as: an array of (at least) length 1+(N_x-1)|*incx*|, containing numbers of the data type indicated in Table 233 on page 1079.

incx
is the stride between the elements within the sequence in array X.

Specified as: an integer; *incx* > 0 or *incx* < 0.

y
See On Return.

incy
is the stride between the elements within the sequence in output array Y.

Specified as: an integer; *incy* > 0 or *incy* < 0.

nh
is the number of elements, N_h, in the sequence in array H.

Specified as: an integer; $N_h > 0$.

nx
is the number of elements, N_x, in the sequence in array X.

Specified as: an integer; $N_x > 0$.

iy0
is the convolution or correlation index of the element to be stored in the first position of the sequence in array Y.

Specified as: an integer. It can have any value.

ny
is the number of elements, N_y, in the sequence in array Y.

Specified as: an integer; $N_y > 0$.

id
is the decimation interval *id* for the output sequence in array Y; that is, every *id*-th value of the convolution or correlation is produced.

Specified as: an integer; *id* > 0.

On Return

y
is the array Y of length N_y, consisting of the output sequence that is the result of the convolution or correlation of the sequence in array H with the sequence in array X, given for every *id*-th value in the convolution or correlation.

Returned as: an array of (at least) length 1+(N_y-1)|*incy*|, containing numbers of the data type indicated in Table 233 on page 1079.

Notes

1. If you specify the same array for X and Y, the following conditions must be true: *incx* = *incy*, *incx* > 0, *incy* > 0, *id* = 1, and *iy0* ≥ N_h-1 for _DCON and *iy0* ≥ 0 for _DCOR. In this case, output overwrites input.

In all other cases, output should not overwrite input; that is, input arrays X and H must have no common elements with output array Y. Otherwise, results are unpredictable. See "Vector concepts" on page 71.

2. If *iy0* and *ny* are such that output outside the basic range is needed, where the basic range is $0 \le k \le (nh+nx-2)$ for SDCON and DDCON and is $(-nh+1) \le k \le (nx-1)$ for SDCOR and DDCOR, the subroutine stores zeros using scalar code. It is not efficient to store many zeros in this manner. If you anticipate that this will happen, you may want to adjust *iy0* and *ny*, so the subroutine computes only for *k* in the above range, or use the ESSL subroutine SSCAL or DSCAL to store the zeros, so you achieve better performance.

Function

The convolution and correlation of a sequence in array H with a sequence in array X, with decimated output, are expressed as follows:

Convolution for SDCON and DDCON:

$$y_k = \sum_{j=\max(0,k-N_x+1)}^{\min(N_h-1,k)} h_j x_{k-j}$$

Correlation for SDCOR and DDCOR:

$$y_k = \sum_{j=\max(0,-k)}^{\min(N_h-1,\ N_x-1-k)} h_j x_{k+j}$$

for $k = iy0, iy0+id, iy0+(2)id, \ldots, iy0+(N_y-1)id$

where:

y_k are elements of the sequence of length N_y in array Y.

x_k are elements of the sequence of length N_x in array X.

h_j are elements of the sequence of length N_h in array H.

iy0 is the convolution or correlation index of the element to be stored in the first position of the sequence in array Y.

min and max select the minimum and maximum values, respectively.

It is assumed that elements outside the range of definition are zero. See reference ["5" on page 1253].

Special Usage

SDCON and DDCON can also perform a correlation, autoconvolution, or autocorrelation. To compute a correlation, you must specify a negative stride for H. To compute the autoconvolution, you must specify the two input sequences to be the same. You can also compute the autocorrelation by using both of these techniques together, letting the two input sequences be the same, and specifying a negative stride for the first input sequence. (See SCOND Example 1.) Because SCOND and SDCON are functionally the same, their results are the same as long as the decimation interval *id* = 1 for SDCON.

SDCOR and DDCOR can also perform a convolution, autocorrelation, or autoconvolution. To compute a convolution, you must specify a negative stride for H. To compute the autocorrelation, you must specify the two input sequences to be the same. You can also compute the autoconvolution by using both of these techniques together, letting the two input sequences be the same and specifying a negative stride for the first input sequence. For examples of these, see SCORD Example 6. Because SCORD and SDCOR are functionally the same, their results are the same as long as the decimation interval *id* = 1 for SDCOR.

Error conditions

Computational Errors

None

Input-Argument Errors

1. *nh*, *nx*, or *ny* ≤ 0
2. *inch*, *incx*, or *incy* = 0
3. *id* ≤ 0

Examples

Example 1

This example shows how to compute a convolution of a sequence in H with a sequence in X, where both sequences are ramp functions. It shows how a decimated output can be obtained, using the same input as Example 1 for SCOND and using a decimation interval ID = 2.

Note: For further examples of use, see SCOND Example 1. Because SCOND and SDCON are functionally the same, their results are the same as long as the decimation interval ID = 1 for SDCON.

Call Statement and Input:

```
            H  INCH  X  INCX  Y  INCY  NH  NX IY0  NY  ID
            |   |    |   |    |   |    |   |   |   |   |
CALL SDCON( H , 1  , X , 1  , Y , 1  , 4 , 8 , 0 , 6 , 2 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output:

```
Y        =  (11.0, 70.0, 130.0, 150.0, 151.0, 72.0)
```

Example 2

This example shows how to compute a correlation of a sequence in H with a sequence in X, where both sequences are ramp functions. It shows how a decimated output can be obtained, using the same input as Example 6 for SCORD and using a decimation interval ID = 2.

Note: For further examples of use, see SCORD Example 6. Because SCORD and SDCOR are functionally the same, their results are the same as long as the decimation interval ID = 1 for SDCOR.

Call Statement and Input:

```
            H  INCH  X  INCX  Y  INCY  NH  NX  IY0  NY  ID
            |   |    |   |    |   |    |   |    |   |   |
CALL SDCOR( H , 1  , X , 1  , Y , 1  , 4 , 8 , -3 , 6 , 2 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output:

```
Y        =  (44.0, 110.0, 140.0, 160.0, 104.0, 18.0)
```

Example 3

This example shows how to compute the same function as computed in Example 1 for SCOND. The input sequences and arguments are the same as that example, except a decimation interval ID = 1 is specified here for SDCON.

Call Statement and Input:

```
            H  INCH  X  INCX  Y  INCY  NH  NX IY0  NY   ID
            |   |    |   |    |   |    |   |   |    |   |
CALL SDCON( H , 1  , X , 1  , Y , 1  , 4 , 8 , 0 , 11 , 1 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output:

```
Y        =  (11.0, 34.0, 70.0, 120.0, 130.0, 140.0, 150.0, 160.0,
             151.0, 122.0, 72.0)
```

SACOR (Autocorrelation of One or More Sequences)

Purpose

This subroutine computes the autocorrelations of one or more sequences using a direct method. The input and output sequences contain short-precision real numbers.

Note: This subroutine is considered obsolete. It is provided in ESSL only for compatibility with earlier releases. You should use SCORD, SDCOR, SCORF and SACORF instead, because they provide **better performance**. For further details, see reference [“5” on page 1253].

Syntax

Language	Syntax
Fortran	CALL SACOR (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nx*, *m*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	sacor (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nx*, *m*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init

is a flag, where:

If *init* ≠ 0, no computation is performed, error checking is performed, and the subroutine exits back to the calling program.

If *init* = 0, the autocorrelations of the sequence in *x* are computed.

Specified as: an integer. It can have any value.

x

is the array X, consisting of *m* input sequences of length N_x, to be autocorrelated. Specified as: an array of (at least) length $1+(N_x-1)inc1x+(m-1)inc2x$, containing short-precision real numbers.

inc1x

is the stride between the elements within each sequence in array X.

Specified as: an integer; *inc1x* > 0.

inc2x

is the stride between the first elements of the sequences in array X.

Specified as: an integer; *inc2x* > 0.

y

See On Return.

inc1y

is the stride between the elements within each sequence in output array Y.

Specified as: an integer; *inc1y* > 0.

inc2y
is the stride between the first elements of each sequence in output array Y.

Specified as: an integer; *inc2y* > 0.

nx
is the number of elements, N_x, in each sequence in array X.

Specified as: an integer; $N_x > 0$.

m
is the number of sequences in array X to be correlated.

Specified as: an integer; $m > 0$.

ny
is the number of elements, N_y, in each sequence in array Y.

Specified as: an integer; $N_y > 0$.

aux1
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

naux1
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

aux2
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

naux2
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

On Return

y
is array Y, consisting of *m* output sequences of length N_y that are the autocorrelation functions of the sequences in array X. Returned as: an array of (at least) length $1 + (N_y\text{-}1)inc1y + (m\text{-}1)inc2y$, containing short-precision real numbers.

Notes

1. Output should not overwrite input; that is, input arrays X and H must have no common elements with output array Y. Otherwise, results are unpredictable. See "Vector concepts" on page 71.
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* ≠ 0 and *init* = 0.
3. Auxiliary storage is not needed, but the arguments *aux1*, *naux1*, *aux2*, and *naux2* must still be specified. You can assign any values to these arguments.

Function

The autocorrelations of the sequences in array X are expressed as follows:

$$y_{ki} = \sum_{j=0}^{N_x-1-k} x_{ji} x_{j+k,i}$$

for:

$k = 0, 1, ..., N_y-1$
$i = 1, 2, ..., m$

where:

y_{ki} are elements of the *m* sequences of length N_y in array Y.
x_{ji} and $x_{j+k,i}$ are elements of the *m* sequences of length N_x in array X.

See references [“25” on page 1254] and [“104” on page 1258].

Only one invocation of this subroutine is needed:

1. You do not need to invoke the subroutine with *init* ≠ 0. If you do, however, the subroutine performs error checking, exits back to the calling program, and no computation is performed.
2. With *init* = 0, the subroutine performs the calculation of the convolutions or correlations.

Error conditions

Computational Errors

None

Input-Argument Errors

1. *nx*, *ny*, or *m* ≤ 0
2. *inc1x*, *inc2x*, *inc1y*, or *inc2y* ≤ 0 (or incompatible)

Examples

Example 1

This example shows how to compute an autocorrelation for three short sequences in array X, where the input sequence length NX is equal to the output sequence length NY. This gives all nonzero autocorrelation values.

The arrays are declared as follows:

```
     REAL*4   X(0:49999), Y(0:49999)
     REAL*8   AUX1, AUX2
```

Call Statement and Input:

```
           INIT  X INC1X INC2X Y INC1Y INC2Y NX  M   NY  AUX1 NAUX1 AUX2   NAUX2
            |    |   |     |   |   |     |   |   |   |    |     |    |       |
CALL SACOR(INIT, X , 1  ,  7 , Y , 1  ,  7 , 7 , 3 , 7 , AUX1 , 0 , AUX2 ,   0)
```

`INIT = 0`(for computation)

X contains the following three sequences:

```
1.0  2.0  3.0
2.0  1.0  2.0
3.0  2.0  1.0
4.0  3.0  2.0
4.0  4.0  3.0
3.0  4.0  4.0
2.0  3.0  4.0
```

Output:

Y contains the following three sequences:

```
59.0  59.0  59.0
54.0  50.0  44.0
43.0  39.0  30.0
29.0  27.0  24.0
16.0  18.0  21.0
 7.0  11.0  20.0
 2.0   6.0  12.0
```

Example 2

This example shows how the output from Example 1 differs when the values for NY and INC2Y are 9 rather than 7. This shows that when NY is greater than NX, the output array is longer, and that part is filled with zeros.

Output:

Y contains the following three sequences:

```
59.0  59.0  59.0
54.0  50.0  44.0
43.0  39.0  30.0
29.0  27.0  24.0
16.0  18.0  21.0
 7.0  11.0  20.0
 2.0   6.0  12.0
 0.0   0.0   0.0
 0.0   0.0   0.0
```

Example 3

This example shows how the output from Example 1 differs when the value for NY is 5 rather than 7. Also, the values for INC1X and INC1Y are 3, and the values for INC2X and INC2Y are 1 rather than 7. This shows that when NY is less than NX, the output array is shortened.

Output:

Y contains the following three sequences:

```
59.0  59.0  59.0
54.0  50.0  44.0
43.0  39.0  30.0
29.0  27.0  24.0
16.0  18.0  21.0
```

SACORF (Autocorrelation of One or More Sequences Using the Mixed-Radix Fourier Method)

Purpose

This subroutine computes the autocorrelations of one or more sequences using the mixed-radix Fourier method. The input and output sequences contain short-precision real numbers.

Note:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 34.

Syntax

Language	Syntax
Fortran	CALL SACORF (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nx*, *m*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	sacorf (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nx*, *m*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init

is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the autocorrelations of the sequence in *x* are computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: an integer. It can have any value.

x
is the array X, consisting of *m* input sequences of length N_x, to be autocorrelated. Specified as: an array of (at least) length $1+(N_x-1)inc1x+(m-1)inc2x$, containing short-precision real numbers.

inc1x
is the stride between the elements within each sequence in array X.

Specified as: an integer; *inc1x* > 0.

inc2x
is the stride between the first elements of the sequences in array X.

Specified as: an integer; *inc2x* > 0.

y
See On Return.

inc1y
is the stride between the elements within each sequence in output array Y.

Specified as: an integer; *inc1y* > 0.

inc2y
is the stride between the first elements of each sequence in output array Y.

Specified as: an integer; *inc2y* > 0.

nx
is the number of elements, N_x, in each sequence in array X.

Specified as: an integer; $N_x > 0$.

m
is the number of sequences in array X to be correlated.

Specified as: an integer; *m* > 0.

ny
is the number of elements, N_y, in each sequence in array Y.

Specified as: an integer; $N_y > 0$.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, the working storage is computed.

If *init* = 0, the working storage is used in the computation of the autocorrelations.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*.

Specified as: an integer; *naux1* > 21 (32-bit integer arguments) or 43 (64-bit integer arguments) and *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 21 (32-bit integer arguments) or 43 (64-bit integer arguments) and the minimum value, you have the option of having the minimum value returned in this argument. For details, see “Using Auxiliary Storage in ESSL” on page 49.

aux2
has the following meaning:

If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2

is the number of doublewords in the working storage specified in *aux2*.

Specified as: an integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, SACORF dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see “Using Auxiliary Storage in ESSL” on page 49.

On Return

y

has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of *m* output sequences of length N_y that are the autocorrelation functions of the sequences in array X.

Returned as: an array of (at least) length 1+(N_y-1)*inc1y*+(*m*-1)*inc2y*, containing short-precision real numbers.

aux1

is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for *init* ≠ 0 and *init* = 0.
3. If you specify the same array for X and Y, then *inc1x* and *inc1y* must be equal and *inc2x* and *inc2y* must be equal. In this case, output overwrites input.
4. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.
5. If *ny* is such that output outside the basic range is needed, the subroutine stores zeros. This range is: 0 ≤ *k* ≤ *nx*-1.

Formulas

Formula for Calculating the Length of the Fourier Transform

Before calculating the necessary sizes of *naux1* and *naux2*, you must determine the length *n* of the Fourier transform. To do this, you use the values of the arguments *nx* and *ny*, inserted into the following formula, to get a value for the variable *nf*. After calculating *nf*, reference the formula or table of allowable values of *n* in “Acceptable Lengths for the Transforms” on page 956, selecting the value equal to or greater than *nf*. Following is the formula for determining *nf*:

$nf = \min(ny, nx)+nx+1$

Processor-Independent Formulas for NAUX1 and NAUX2

The required values of *naux1* and *naux2* depend on the value determined for *n* in Formula for Calculating the Length of the Fourier Transform and the argument *m*.

NAUX1 Formulas:

For 32-bit integer arguments:

If $n \le 16384$, use *naux1* = 55000.
If $n > 16384$, use *naux1* = 40000+1.89*n*.

For 64-bit integer arguments:

If $n \le 16384$, use *naux1* = 75000.
If $n > 16384$, use *naux1* = 60000+1.89*n*.

NAUX2 Formulas:

If $n \le 16384$, use *naux2* = 50000.
If $n > 16384$, use *naux2* = 40000+1.64*n*.

Function

The autocorrelations of the sequences in array X are expressed as follows:

$$y_{ki} = \sum_{j=0}^{N_x-1-k} x_{ji}x_{j+k,i}$$

for:

$k = 0, 1, \ldots, N_y-1$
$i = 1, 2, \ldots, m$

where:

y_{ki} are elements of the *m* sequences of length N_y in array Y.
x_{ji} and $x_{j+k,i}$ are elements of the *m* sequences of length N_x in array X.

This subroutine uses a Fourier transform method with a mixed-radix capability. This provides maximum performance for your application. The length of the transform, *n*, that you must calculate to determine the correct sizes for *naux1* and *naux2* is the same length used by the Fourier transform subroutines called by this subroutine. See references [“25” on page 1254] and [“104” on page 1258].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the autocorrelations.

Error conditions

Resource Errors

Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors

None

Input-Argument Errors

1. *nx*, *ny*, or *m* ≤ 0
2. *inc1x*, *inc2x*, *inc1y*, or *inc2y* ≤ 0 (or incompatible)
3. The resulting correlation is too long.
4. The subroutine has not been initialized with the present arguments.
5. *naux1* ≤ 21
6. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
7. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows how to compute an autocorrelation for three short sequences in array X, where the input sequence length NX is equal to the output sequence length NY. This gives all nonzero autocorrelation values. The arrays are declared as follows:

```
     REAL*4   X(0:49999), Y(0:49999)
     REAL*8   AUX1(2959), AUX2(1)
```

First, initialize AUX1 using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
            INIT  X INC1X INC2X Y INC1Y INC2Y NX  M   NY  AUX1  NAUX1 AUX2  NAUX2
             |    |   |     |   |   |     |   |   |   |    |      |    |      |
CALL SACORF(INIT, X , 1  ,  7 , Y , 1  ,  7 , 7 , 3 , 7 , AUX1, 2959, AUX2,   0)
```

`INIT = 1`(for initialization)
`INIT = 0`(for computation)

X contains the following three sequences:

```
1.0  2.0  3.0
2.0  1.0  2.0
3.0  2.0  1.0
4.0  3.0  2.0
4.0  4.0  3.0
3.0  4.0  4.0
2.0  3.0  4.0
```

Output:

Y contains the following three sequences:

```
59.0  59.0  59.0
54.0  50.0  44.0
43.0  39.0  30.0
29.0  27.0  24.0
16.0  18.0  21.0
 7.0  11.0  20.0
 2.0   6.0  12.0
```

Example 2

This example shows how the output from Example 1 differs when the value for NY and `INC2Y` are 9 rather than 7. This shows that when NY is greater than NX, the output array is longer and that part is filled with zeros.

Output:

Y contains the following three sequences:

```
59.0  59.0  59.0
54.0  50.0  44.0
43.0  39.0  30.0
29.0  27.0  24.0
16.0  18.0  21.0
 7.0  11.0  20.0
 2.0   6.0  12.0
 0.0   0.0   0.0
 0.0   0.0   0.0
```

Example 3

This example shows how the output from Example 1 differs when the value for NY is 5 rather than 7. Also, the values for `INC1X` and `INC1Y` are 3 rather than 1, and the values for `INC2X` and `INC2Y` are 1 rather than 7. This shows that when NY is less than NX, the output array is shortened.

Output:

Y contains the following three sequences:

```
59.0  59.0  59.0
54.0  50.0  44.0
43.0  39.0  30.0
29.0  27.0  24.0
16.0  18.0  21.0
```

Related-Computation Subroutines

This contains the related-computation subroutine descriptions.

SPOLY and DPOLY (Polynomial Evaluation)

Purpose

These subroutines evaluate a polynomial of degree *k*, using coefficient vector ***u***, input vector ***x***, and output vector ***y***:

$$y_i = u_0 + u_1 x_i + u_2 x_i^2 + \ldots + u_k x_i^k \qquad \text{for } i = 1,2,\ldots,n$$

where u_k, x_i, and y_i are elements of ***u***, ***x***, and ***y***, respectively.

Table 234: Data Types.

u, x, y	**Subroutine**
Short-precision real	SPOLY
Long-precision real	DPOLY

Syntax

Language	**Syntax**
Fortran	CALL SPOLY | DPOLY (*u*, *incu*, *k*, *x*, *incx*, *y*, *incy*, *n*)
C and C++	spoly | dpoly (*u*, *incu*, *k*, *x*, *incx*, *y*, *incy*, *n*);

On Entry

u

is the coefficient vector ***u*** of length *k*+1. It contains elements u_0, u_1, u_0, u_1, u_2, ..., u_k, which are stored in this order. Specified as: a one-dimensional array of (at least) length 1+*k*|*incu*|, containing numbers of the data type indicated in Table 234 on page 1091.

incu

is the stride for vector ***u***.

Specified as: an integer. It can have any value.

k

is the degree *k* of the polynomial.

Specified as: an integer; $k \geq 0$.

x

is the input vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 234 on page 1091.

incx

is the stride for vector ***x***.

Specified as: an integer. It can have any value.

y

See On Return.

incy

is the stride for the output vector ***y***. Specified as: an integer. It can have any value.

n

is the number of elements in input vector ***x*** and the number of resulting elements in output vector ***y***.

Specified as: an integer; $n \geq 0$.

On Return

y

is the output vector ***y*** of length *n*, containing the results of the polynomial evaluation. Returned as: a one-dimensional array of (at least) length 1+(*n*-1)|*incy*|, containing numbers of the data type indicated in Table 234 on page 1091.

Notes

Vectors ***u***, ***x***, and ***y*** must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

The evaluation of the polynomial:

$$y_i = u_0 + u_1 x_i + u_2 x_i^2 + \ldots + u_k x_i^k \qquad \text{for } i = 1,2,\ldots,n$$

is expressed as follows:

$y_i = u_0 + x_i\,(u_1 + x_i\,(u_2 + \ldots + x_i\,(u_{k-1} + x_i u_k)\ \ldots)$ for $i = 1, 2, \ldots, n$

See reference ["100" on page 1258] for Horner's Rule. If *n* is 0, no computation is performed. For SPOLY, intermediate results are accumulated in long precision.

SPOLY provides the same function as the IBM 3838 function POLY, with restrictions removed. DPOLY provides a long-precision computation that is not included in the IBM 3838 functions. See the *IBM 3838 Array Processor Functional Characteristics* manual.

Error conditions

Computational Errors

None

Input-Argument Errors

1. $k < 0$
2. $n < 0$

Examples

Example 1

This example shows a polynomial evaluation with the degree, K, equal to 0.

Call Statement and Input:

```
            U   INCU   K   X   INCX   Y  INCY  N
            |    |     |   |    |     |   |    |
CALL SPOLY( U , INCU , 0 , X , INCX , Y , 1  , 3 )
```

`U = (4.0)`
`INCU` =(not relevant)
`X` =(not relevant)
`INCX` =(not relevant)

Output:

```
Y        =  (4.0, 4.0, 4.0)
```

Example 2

This example shows a polynomial evaluation, using a negative stride `INCU` for vector ***u***. For ***u***, processing begins at element `U(4)` which is 1.0.

Call Statement and Input:

```
            U  INCU  K   X  INCX  Y  INCY  N
            |    |   |   |   |    |   |    |
CALL SPOLY( U , -1 , 3 , X , 1  , Y , 1  , 3 )

U        =  (4.0, 3.0, 2.0, 1.0)
X        =  (2.0, 1.0, -3.0)
```

Output:

```
Y        =  (49.0, 10.0, -86.0)
```

Example 3

This example shows a polynomial evaluation, using a stride `INCX` of 0 for input vector ***x***.

Call Statement and Input:

```
            U  INCU  K   X  INCX  Y  INCY  N
            |   |    |   |   |    |   |    |
CALL SPOLY( U , 1  , 3 , X , 0  , Y , 1  , 3 )

U        =  (4.0, 3.0, 2.0, 1.0)
X        =  (2.0, . , . )
```

Output:

```
Y        =  (26.0, 26.0, 26.0)
```

Example 4

This example shows a polynomial evaluation, using a stride INCX greater than 1 for input vector ***x***, and a negative stride INCY for output vector ***y***. For ***y***, results are stored beginning at element Y(5).

Call Statement and Input:

```
            U  INCU  K   X  INCX  Y  INCY  N
            |   |    |   |   |    |   |    |
CALL SPOLY( U , 1  , 3 , X , 2  , Y , -2 , 3 )

U        =  (4.0, 3.0, 2.0, 1.0)
X        =  (2.0, . , -3.0, . , 1.0)
```

Output:

```
Y        =  (10.0, . , -14.0, . , 26.0)
```

SIZC and DIZC (I-th Zero Crossing)

Purpose

These subroutines find the position of the *i*-th zero crossing in vector ***x***. This is the *i*-th transition between positive and negative or negative and positive, where 0 is considered a positive value. It returns the position of the element in vector ***x*** where the *i*-th zero crossing is detected. The direction of the scan is either from the first element to the last or from the last element to the first, depending on the value you specify for the scan direction argument.

Table 235: Data Types.

x	**Subroutine**
Short-precision real	SIZC
Long-precision real	DIZC

Syntax

Language	**Syntax**
Fortran	CALL SIZC | DIZC (*x*, *idrx*, *n*, *i*, *ky*)
C and C++	sizc | dizc (*x*, *idrx*, *n*, *i*, *ky*);

On Entry

x

is the target vector ***x*** of length *n*.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 235 on page 1094.

idrx

indicates the scan direction. If it is positive or 0, ***x*** is scanned from the first element to the last (1, *n*). If it is negative, ***x*** is scanned from the last element to the first (*n*, 1).

Specified as: an integer. It can have any value.

n

is the number of elements in vector ***x***. Specified as: an integer; *n* > 1.

i

is the number of the zero crossing to be identified.

Specified as: an integer; *i* > 0.

ky
See On Return.

On Return

ky
is the integer vector ***ky*** of length 2, containing elements ky_1 and ky_2, where:

If the *i*-th zero crossing is found:

- $ky_1 = j$, where *j* is the position of the element x_j at the point that the *i*-th zero crossing is found. The position is always relative to the beginning of the vector regardless of the scan direction.
- $ky_2 = i$

If the *i*-th zero crossing is not found:

- $ky_1 = 0$
- ky_2 = the total number of zero crossings encountered in the scan.

Returned as: an array of (at least) length 2, containing integers.

Notes

The *aux* and *naux* arguments, required in some earlier releases of ESSL, are no longer required by these subroutines. If your program still includes them, you do not have to change your program; it continues to run normally. It ignores these arguments. However, if you did any program checking for error code 2015, you may want to remove it, because this error no longer occurs. (You must not code these arguments in your C program.)

Function

The *i*-th zero crossing in vector ***x*** is found by scanning vector ***x*** for *i* occurrences of TRUE for the following logical expressions. A zero crossing is defined here as a crossing either from a positive value to a negative value or from a negative value to a positive value, where 0 is considered a positive value. If the *i*-th zero crossing is found, the value of *j* at that point is returned in ky_1 as the position of the *i*-th zero crossing, and *i* is returned in ky_2.

If $idrx \geq 0$:

TRUE = ($x_{j-1} < 0$ and $x_j \geq 0$) or ($x_{j-1} \geq 0$ and $x_j < 0$) for $j = 2, n$

If $idrx < 0$:

TRUE = ($x_{j+1} < 0$ and $x_j \geq 0$) or ($x_{j+1} \geq 0$ and $x_j < 0$) for $j = n\text{-}1, 1$

If the position of the *i*-th zero crossing is not found, 0 is returned in y_1 and the number of zero crossings encountered in the scan is returned in y_2.

SIZC provides the same functions as the IBM 3838 functions NZCP and NZCN, with restrictions removed. It combines these functions into one ESSL subroutine. DIZC provides a long-precision computation that is not included in the IBM 3838 functions. See the *IBM 3838 Array Processor Functional Characteristics* manual.

Error conditions

Computational Errors
None

Input-Argument Errors

1. $n \leq 1$
2. $i \leq 0$

Examples

Example 1

This example shows a scan of a vector ***x*** from the first element to the last. It is looking for the fifth zero crossing, which is encountered at position 9.

Call Statement and Input:

```
           X  IDRX  N    I   KY
           |   |    |    |   |
CALL SIZC( X , 1  , 12 , 5 , KY )

X        =  (2.0, -1.0, -3.0, 3.0, 0.0, 8.0, -2.0, 0.0, -5.0, -3.0,
             2.0, -9.0)
```

Output:

```
KY       =  (9, 5)
```

Example 2

This example shows a scan of a vector ***x*** from the last element to the first. It is looking for the seventh zero crossing, which is encountered at position 3. Because IDRX is negative, X is scanned from the last element, X(12), to the first element, X(1).

Call Statement and Input:

```
           X  IDRX  N    I   KY
           |    |   |    |   |
CALL SIZC( X , -1 , 12 , 7 , KY )

X        =  (2.0, -1.0, 3.0, -3.0, 0.0, -8.0, -2.0, 0.0, -5.0, -3.0,
             2.0, -9.0)
```

Output:

```
KY       =  (3, 7)
```

Example 3

This example shows a scan of a vector ***x*** when the *i*-th zero crossing is not found. It encounters seven zero crossings and returns this value in KY(2).

Call Statement and Input:

```
           X  IDRX  N    I    KY
           |   |    |    |    |
CALL SIZC( X , 1  , 12 , 10 , KY )

X        =  (2.0, -1.0, -3.0, 3.0, 0.0, 8.0, -2.0, 0.0, -5.0, -3.0,
             2.0, -9.0)
```

Output:

```
KY       =  (0, 7)
```

STREC and DTREC (Time-Varying Recursive Filter)

Purpose

These subroutines implement the first-order time-varying recursive equation, using initial value *s*, target vectors ***u*** and ***x***, and output vector ***y***.

Table 236: Data Types.	
s, u, x, y	**Subroutine**
Short-precision real	STREC
Long-precision real	DTREC

Syntax

Language	**Syntax**
Fortran	CALL STREC | DTREC (*s*, *u*, *incu*, *x*, *incx*, *y*, *incy*, *n* , *iopt*)
C and C++	strec | dtrec (*s*, *u*, *incu*, *x*, *incx*, *y*, *incy*, *n*, *iopt*);

On Entry

s

is the scalar *s* used in the initial computation for y_1.

Specified as: a number of the data type indicated in Table 236 on page 1097.

u

is the target vector ***u*** of length *n*.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incu*|, containing numbers of the data type indicated in Table 236 on page 1097.

incu

is the stride for target vector ***u***.

Specified as: an integer. It can have any value.

x

is the target vector ***x*** of length *n*.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 236 on page 1097.

incx

is the stride for target vector ***x.*** Specified as: an integer. It can have any value.

y

See On Return.

incy

is the stride for output vector ***y.*** Specified as: an integer; *incy* > 0 or *incy* < 0.

n

is the number of elements in vectors ***u*** and ***x*** and the number of resulting elements in output vector ***y***.

Specified as: an integer; *n* ≥ 0.

iopt

this argument has no effect on the performance of the computation, but still must be Specified as: an integer; *iopt* = 0 or 1.

On Return

y

is the vector ***y*** of length *n*, containing the results of the implementation of the first-order time-varying recursive equation. Returned as: a one-dimensional array of (at least) length 1+(*n*-1)|*incy*|, containing numbers of the data type indicated in Table 236 on page 1097.

Notes

Vectors ***u***, ***x***, and ***y*** must have no common elements; otherwise, results are unpredictable. See "Vector concepts" on page 71.

Function

The first-order time-varying recursive equation is expressed as follows:

$y_1 = s+u_1x_1$
$y_2 = u_2y_1+u_1x_2$
.
.
.
$y_i = u_iy_{i-1}+u_1x_i$ for $i = 3, 4, ..., n$

STREC provides the same function as the IBM 3838 function REC, with restrictions removed. DTREC provides a long-precision computation that is not included in the IBM 3838 functions. See the *IBM 3838 Array Processor Functional Characteristics* manual.

Error conditions

Computational Errors
None

Input-Argument Errors

1. *incy* = 0
2. *n* < 0
3. *iopt* ≠ 0 or 1

Examples

Example 1

This example shows all strides INCU, INCX, and INCY equal to 1 for vectors ***u***, ***x***, and ***y***, respectively.

Call Statement and Input:

```
              S    U  INCU  X  INCX  Y  INCY  N  IOPT
              |    |   |    |   |    |   |    |   |
CALL STREC( 1.0 , U , 1   , X , 1   , Y , 1   , 8 , 0 )

U        =  (1.0, 2.0, 3.0, 3.0, 2.0, 1.0, 1.0, 2.0)
X        =  (3.0, 2.0, 1.0, 1.0, 2.0, 3.0, 3.0, 2.0)
```

Output:

```
Y        =  (4.0, 10.0, 31.0, 94.0, 190.0, 193.0, 196.0, 394.0)
```

Example 2

This example shows a stride, INCU, that is greater than 1 for vector ***u***. The strides INCX and INCY for vectors ***x*** and ***y***, respectively, are 1.

Call Statement and Input:

```
              S    U  INCU  X  INCX  Y  INCY  N  IOPT
              |    |   |    |   |    |   |    |   |
CALL STREC( 1.0 , U , 2   , X , 1   , Y , 1   , 4 , 0 )

U        =  (1.0, . , 3.0, . , 2.0, . , 1.0, . )
X        =  (3.0, 2.0, 1.0, 1.0, 2.0, 3.0, 3.0, 2.0)
```

Output:

```
Y        =  (4.0, 14.0, 29.0, 30.0)
```

Example 3

This example shows a stride, `INCU`, of 1 for vector ***u***, a stride, `INCX`, that is greater than 1 for vector ***x***, and a negative stride, `INCY`, for vector ***y***. For ***y***, results are stored beginning at element `Y(4)`.

Call Statement and Input:

```
              S    U  INCU  X  INCX  Y  INCY  N  IOPT
              |    |   |    |   |    |    |   |   |
CALL STREC( 1.0 , U , 1  , X , 2  , Y , -1 , 4 , 1 )

U        =  (1.0, 2.0, 3.0, 3.0, 2.0, 1.0, 1.0, 2.0)
X        =  (3.0, . , 1.0, . , 2.0, . , 3.0)
```

Output:

```
Y        =  (90.0, 29.0, 9.0, 4.0)
```

SQINT and DQINT (Quadratic Interpolation)

Purpose

These subroutines perform a quadratic interpolation at specified points in the vector ***x***, using initial linear displacement in the samples *s*, sample interval *g*, output scaling parameter Ω, and sample reflection times in vector ***t***. The result is returned in vector ***y***.

Table 237: Data Types.	
x*, *s*, *g*, Ω, *t*, *y	**Subroutine**
Short-precision real	SQINT
Long-precision real	DQINT

Syntax

Language	**Syntax**
Fortran	CALL SQINT | DQINT (*s*, *g*, *omega*, *x*, *incx*, *n*, *t*, *inct*, *y*, *incy*, *m*)
C and C++	sqint | dqint (*s*, *g*, *omega*, *x*, *incx*, *n*, *t*, *inct*, *y*, *incy*, *m*);

On Entry

s

is the scalar *s*, containing the initial linear displacement in samples.

Specified as: a number of the data type indicated in Table 237 on page 1099.

g

is the scalar *g*, containing the sample interval.

Specified as: a number of the data type indicated in Table 237 on page 1099; *g* > 0.0.

omega

is the output scaling parameter Ω.

Specified as: a number of the data type indicated in Table 237 on page 1099.

x

is the vector ***x*** of length *n*, containing the trace data.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 237 on page 1099.

incx
is the stride for vector ***x***.

Specified as: an integer; *incx* > 0 or *incx* < 0.

n
is the number of elements in vector ***x***.

Specified as: an integer; $n \geq 3$.

t
is the vector ***t*** of length *m*, containing the sample reflection times to be processed.

Specified as: a one-dimensional array of (at least) length 1+(*m*-1)|*inct*|, containing numbers of the data type indicated in Table 237 on page 1099.

inct
is the stride for vector ***t***.

Specified as: an integer; *inct* > 0 or *inct* < 0.

y
See On Return.

incy
is the stride for output vector ***y***.

Specified as: an integer; *incy* > 0 or *incy* < 0.

m
is the number of elements in vector ***t*** and the number of elements in output vector ***y***.

Specified as: an integer; $m \geq 0$.

On Return

y
is the vector ***y*** of length *m*, containing the results of the quadratic interpolation. Returned as: a one-dimensional array of (at least) length 1+(*m*-1)|*incy*|, containing numbers of the data type indicated in Table 237 on page 1099.

Function

The quadratic interpolation, which is expressed as follows:

$$y_i = \Omega \left(trace_{k_i}\left(f_i^{\,2} - f_i \right) + 2\ trace_{k_i+1}\left(1 - f_i^{\,2} \right) + trace_{k_i+2}\left(f_i^{\,2} + f_i \right) \right)$$

for *i* = 1, 2, ..., *m*

uses the following values:

x is the vector containing the specified points.
s is the initial linear displacement in the samples.
g is a sample interval.
Ω is the output scaling parameter.
t is the vector containing the sample reflection times.

and where ***trace***, ***k***, ***f***, and ***w*** are four working vectors, and *so* is a working scalar defined as:

$trace_1 = 3x_1 - 3x_2 + x_3$
$trace_{i+1} = x_i$ for *i* = 1, 2, ..., *n*

$so = s+2.0$
$w_i = so+t_i / g$ for $i = 1, 2, ..., m$
f_i = fraction part of w_i
k_i+1 = integer part of w_i

Note: Allowing k_i+1 to have a value of 2 results in performance degradation. If possible, avoid specifying a point at which this occurs.

If *n* or *m* is 0, no computation is performed.

SQINT provides the same function as the IBM 3838 function INT, with restrictions removed. DQINT provides a long-precision computation that is not included in the IBM 3838 functions. See the *IBM 3838 Array Processor Functional Characteristics* manual.

Error conditions

Computational Errors

The condition ($k_i+1 > n$) or ($k_i+1 \leq 2$) has occurred, where *n* is the number of elements in vector ***x***. See "Function" on page 1100 for how to calculate k_i.

- The lower range *l* and the upper range *j* of the vector are identified in the computational error message.
- The return code is set to 1.
- The ranges *l* and *j* of the vector can be determined at run time by using the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2100 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 65.

Input-Argument Errors

1. $n < 3$
2. $m < 0$
3. $g \leq 0$
4. *incx* = 0
5. *inct* = 0
6. *incy* = 0

Examples

Example 1

This example shows a quadratic interpolation, using vectors with strides of 1.

Call Statement and Input:

```
              S     G   OMEGA  X  INCX  N   T  INCT  Y  INCY  M
              |     |     |    |   |    |   |   |    |   |    |
CALL SQINT( 2.0 , 1.0 , 1.0 , X , 1  , 8 , T , 1  , Y , 1  , 4 )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0)
T        =  (1.5, 2.5, 3.5, 4.5)
```

Output:

```
Y        =  (9.0, 11.0, 13.0, 15.0)
```

Example 2

This example shows a quadratic interpolation, using vectors with a positive stride of 1 and negative strides of -1.

Call Statement and Input:

```
             S     G   OMEGA  X  INCX  N   T  INCT  Y  INCY  M
             |     |     |    |    |   |   |    |   |   |    |
CALL SQINT( 2.0 , 1.0 , 1.0 , X , -1 , 8 , T , -1 , Y , 1  , 4 )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0)
T        =  (1.5, 2.5, 3.5, 4.5)
```

Output:

```
Y        =  (3.0, 5.0, 7.0, 9.0)
```

Example 3

This example shows a quadratic interpolation, using vectors with a positive stride greater than 1 and negative strides less than -1.

Call Statement and Input:

```
             S     G   OMEGA  X  INCX  N   T  INCT  Y  INCY  M
             |     |     |    |    |   |   |    |   |   |    |
CALL SQINT( 2.0 , 1.0 , 1.0 , X , -2 , 8 , T , -1 , Y , 2  , 4 )

X        =  (1.0, . , 3.0, . , 5.0, . , 7.0, . , 9.0, . , 11.0, . ,
             13.0, . , 15.0)
T        =  (1.36, 2.36, 3.36, 4.36)
```

Output:

```
Y        =  (4.56, . , 8.56, . , 12.56, . , 16.56)
```

Example 4

This example shows a quadratic interpolation, using vectors with positive strides and larger values for S and G than shown in the previous examples.

Call Statement and Input:

```
             S      G   OMEGA  X  INCX  N   T  INCT  Y  INCY  M
             |      |     |    |   |    |   |   |    |   |    |
CALL SQINT( 3.0 , 10.0 , 1.0 , X , 1  , 8 , T , 2  , Y , 3  , 4 )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0)
T        =  (1.5, . , 2.5, . , 3.5, . , 4.5)
```

Output:

```
Y        =  (8.3, . , . , 8.5, . , . , 8.7, . , . , 8.9)
```

SWLEV, DWLEV, CWLEV, and ZWLEV (Wiener-Levinson Filter Coefficients)

Purpose

These subroutines compute the coefficients of an n-point Wiener-Levinson filter, using vector ***x***, the trace for which the filter is to be designed, and vector ***u***, the right-hand side of the system, chosen to remove reverberations or sharpen the wavelet. The result is returned in vector ***y***.

Table 238: Data Types.

x, u, y	***aux***	**Subroutine**
Short-precision real	Long-precision real	SWLEV
Long-precision real	Long-precision real	DWLEV

Table 238: Data Types. (continued)		
x, u, y	***aux***	**Subroutine**
Short-precision complex	Long-precision complex	CWLEV
Long-precision complex	Long-precision complex	ZWLEV

Syntax

Language	Syntax
Fortran	CALL SWLEV | DWLEV | CWLEV | ZWLEV | (*x*, *incx*, *u*, *incu*, *y*, *incy*, *n*, *aux*, *naux*)
C and C++	swlev | dwlev | cwlev | zwlev (*x*, *incx*, *u*, *incu*, *y*, *incy*, *n*, *aux*, *naux*);

On Entry

x

is the vector ***x*** of length *n*, containing the trace data for which the filter is to be designed.

For SWLEV and DWLEV, *x* represents the first row (or the first column) of a positive definite or negative definite symmetric Toeplitz matrix, which is the autocorrelation matrix for which the filter is designed.

For CWLEV and ZWLEV, *x* represents the first row of a positive definite or negative definite complex Hermitian Toeplitz matrix, which is the autocorrelation matrix for which the filter is designed.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 238 on page 1102.

incx

is the stride for vector ***x***.

Specified as: an integer; *incx* > 0.

u

is the vector ***u*** of length *n*, containing the right-hand side of the system to be solved.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incu*|, containing numbers of the data type indicated in Table 238 on page 1102.

incu

is the stride for vector ***u***.

Specified as: an integer. It can have any value.

y

See On Return.

incy

is the stride for vector ***y***.

Specified as: an integer; *incy* > 0 or *incy* < 0.

n

is the number of elements in vectors ***x***, ***u***, and ***y***.

Specified as: an integer; *n* ≥ 0.

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by these subroutines.

Specified as: an area of storage of length *naux*, containing numbers of the data type indicated in Table 238 on page 1102.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SWLEV, DWLEV, CWLEV, and ZWLEV dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux* ≥ 3*n*.

You cannot use dynamic allocation if you need the information returned in AUX(1).

On Return

y

is the vector ***y*** of length *n*, containing the solution vector—that is, the coefficients of the n-point Wiener-Levinson filter. Returned as: a one-dimensional array of (at least) length 1+(*n*-1)|*incy*|, containing numbers of the data type indicated in Table 238 on page 1102.

aux

is the storage work area used by these subroutines, where if *naux* ≠ 0:

If AUX(1) = 0.0, the input Toeplitz matrix is positive definite or negative definite.

If AUX(1) > 0.0, the input Toeplitz matrix is indefinite (that is, it is not positive definite and it is not negative definite). The value returned in AUX(1) is the order of the first submatrix of ***A*** that is indefinite. The subroutine continues processing. See reference [“75” on page 1257] for information about under what circumstances your solution vector *y* would be valid.

All other values in *aux* are overwritten and are not significant.

Returned as: an area of storage of length *naux*, containing numbers of the data type indicated in Table 238 on page 1102, where AUX(1) ≥ 0.0.

Notes

1. For a description of a positive definite or negative definite symmetric Toeplitz matrix, see “Positive Definite or Negative Definite Symmetric Toeplitz Matrix” on page 86.
2. For a description of a positive definite or negative definite complex Hermitian Toeplitz matrix, see “Positive Definite or Negative Definite Complex Hermitian Toeplitz Matrix” on page 87.
3. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 49.

Function

The computation of the coefficients of an n-point Wiener-Levinson filter in vector ***y*** is expressed as solving the following system:

$\boldsymbol{Ay} = \boldsymbol{u}$

where:

- For SWLEV and DWLEV, matrix ***A*** is a real symmetric Toeplitz matrix whose first row (or first column) is represented by vector ***x***.

  For CWLEV and ZWLEV, matrix ***A*** is a complex Hermitian Toeplitz matrix whose first row is represented by vector ***x***.
- ***u*** is the vector specifying the right side of the system, chosen to remove reverberations or to sharpen the wavelet.
- ***y*** is the solution vector.

See reference ["75" on page 1257], ["36" on page 1255], and the *IBM 3838 Array Processor Functional Characteristics*.

If *n* is 0, no computation is performed. For SWLEV and CWLEV, intermediate results are accumulated in long precision.

SWLEV provides the same function as the IBM 3838 function WLEV, with restrictions removed. See the *IBM 3838 Array Processor Functional Characteristics* manual.

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors
None

Input-Argument Errors

1. *n* < 0
2. *incx* ≤ 0
3. *incy* = 0
4. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value specified in the syntax for this argument. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows how to compute filter coefficients in vector ***y*** by solving the system ***Ay*** = ***u***. Matrix ***A*** is:

```
                         ┌                         ┐
                         | 50.0  -8.0   7.0  -5.0 |
                         | -8.0  50.0  -8.0   7.0 |
                         |  7.0  -8.0  50.0  -8.0 |
                         | -5.0   7.0  -8.0  50.0 |
                         └                         ┘
```

This input Toeplitz matrix is positive definite, as indicated by the zero value in AUX(1) on output.

Call Statement and Input:

```
             X  INCX  U  INCU  Y  INCY  N   AUX  NAUX
             |   |    |   |    |   |    |    |    |
CALL SWLEV( X , 1  , U , 1  , Y , 1  , 4 , AUX , 12 )
```

X = (50.0, -8.0, 7.0, -5.0)
U = (40.0, -10.0, 30.0, 20.0)
AUX =(not relevant)

Output:

```
Y        =  (0.7667, -0.0663, 0.5745, 0.5778)
AUX      =  (0.0, . , . , . , . , . , . , . , . , . , . , . )
```

Example 2

This example shows how to compute filter coefficients in vector ***y*** by solving the system ***Ay*** = ***u***. Matrix ***A*** is:

```
                         ┌                         ┐
                         | 10.0  -8.0   7.0  -5.0 |
                         | -8.0  10.0  -8.0   7.0 |
                         |  7.0  -8.0  10.0  -8.0 |
                          | -5.0   7.0  -8.0  10.0 |
                                                   ┘
```

This input Toeplitz matrix is not positive definite, as indicated by the zero value in AUX(1) on output.

Call Statement and Input:

```
             X  INCX  U  INCU  Y  INCY  N   AUX  NAUX
             |   |    |   |    |   |    |    |    |
CALL SWLEV( X , 1  , U , 1  , Y , 1  , 4 , AUX , 12 )
```

X = (10.0, -8.0, 7.0, -5.0)
U = (40.0, -10.0, 30.0, 20.0)
AUX =(not relevant)

Output:

```
Y        =  (5.1111, 5.5555, 12.2222, 10.4444)
AUX      =  (0.0, . , . , . , . , . , . , . , . , . , . , . )
```

Example 3

This example shows a vector ***x*** with a stride greater than 1, a vector ***u*** with a negative stride, and a vector ***y*** with a stride of 1. It uses the same input Toeplitz matrix as in Example 2, which is not positive definite.

Call Statement and Input:

```
             X  INCX  U  INCU  Y  INCY  N   AUX  NAUX
             |   |    |    |   |   |    |    |    |
CALL SWLEV( X , 2  , U , -2 , Y , 1  , 4 , AUX , 12 )
```

X = (10.0, . , -8.0, . , 7.0, . , -5.0)
U = (20.0, . , 30.0, . , -10.0, . , 40.0)
AUX =(not relevant)

Output:

```
Y        =  (5.1111, 5.5555, 12.2222, 10.4444)
AUX      =  (0.0, . , . , . , . , . , . , . , . , . , . , . )
```

Example 4

This example shows how to compute filter coefficients in vector ***y*** by solving the system ***Ay*** = ***u***. Matrix ***A*** is:

```
            ┌                                                 ┐
            |  (10.0, 0.0)   (2.0, -3.0)  (-3.0, 1.0)   (1.0, 1.0) |
            |   (2.0, 3.0)   (10.0, 0.0)  (2.0, -3.0)  (-3.0, 1.0) |
            | (-3.0, -1.0)    (2.0, 3.0)  (10.0, 0.0)  (2.0, -3.0) |
            |  (1.0, -1.0)  (-3.0, -1.0)   (2.0, 3.0)  (10.0, 0.0) |
                                                                   ┘
```

This input complex Hermitian Toeplitz matrix is positive definite, as indicated by the zero value in AUX(1) on output.

Call Statement and Input:

```
             X  INCX  U  INCU  Y  INCY  N   AUX  NAUX
             |   |    |   |    |   |    |    |    |
CALL ZWLEV( X , 1  , U , 1  , Y , 1  , 4 , AUX , 12 )
```

X = ((10.0, 0.0), (2.0, -3.0), (-3.0, 1.0), (1.0, 1.0))
U = ((8.0, 3.0), (21.0, -5.0), (67.0, -13.0), (72.0, 11.0))
AUX =(not relevant)

Output:

```
Y        =  ((1.0, 0.0), (3.0, 0.0), (5.0, 0.0), (7.0, 0.0))
AUX      =  ((0.0, 0.0), . , . , . , . , . , . , . , . , . , . , . )
```

Example 5

This example shows a vector ***x*** with a stride greater than 1, a vector ***u*** with a negative stride, and a vector ***y*** with a stride of 1. It uses the same input complex Hermitian Toeplitz matrix as in Example 4.

This input complex Hermitian Toeplitz matrix is positive definite, as indicated by the zero value in `AUX(1)` on output.

Call Statement and Input:

```
            X  INCX  U  INCU   Y  INCY  N   AUX  NAUX
            |   |    |    |    |   |    |    |    |
CALL ZWLEV( X , 2  , U , -2  , Y , 1  , 4 , AUX , 12 )
```

```
X = ((10.0, 0.0), . , (2.0, -3.0), . , (-3.0, 1.0), .
,
(1.0, 1.0))
U = ((72.0, 11.0), . , (67.0, -13.0), . , (21.0, -5.0), . ,
(8.0, 3.0), . )
```
AUX =(not relevant)

Output:

```
Y        =  ((1.0, 0.0), (3.0, 0.0), (5.0, 0.0), (7.0, 0.0))
AUX      =  ((0.0, 0.0), . , . , . , . , . , . , . , . , . , . , . )
```

Chapter 13. Sorting and searching subroutines

The sorting and searching subroutines are described here.

Overview of the Sorting and Searching Subroutines

The sorting and searching subroutines operate on three types of data: integer, short-precision real, and long-precision-real. The sorting subroutines perform sorts with or without index designations. The searching subroutines perform either a binary or sequential search.

Table 239: List of Sorting and Searching Subroutines.

Integer Subroutine	**Short-Precision Subroutine**	**Long-Precision Subroutine**	**Descriptive Name and Location**
ISORT	SSORT	DSORT	"ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)" on page 1110
ISORTX	SSORTX	DSORTX	"ISORTX, SSORTX, and DSORTX (Sort the Elements of a Sequence and Note the Original Element Positions)" on page 1111
ISORTS	SSORTS	DSORTS	"ISORTS, SSORTS, and DSORTS (Sort the Elements of a Sequence Using a Stable Sort and Note the Original Element Positions)" on page 1114
IBSRCH	SBSRCH	DBSRCH	"IBSRCH, SBSRCH, and DBSRCH (Binary Search for Elements of a Sequence X in a Sorted Sequence Y)" on page 1117
ISSRCH	SSSRCH	DSSRCH	"ISSRCH, SSSRCH, and DSSRCH (Sequential Search for Elements of a Sequence X in the Sequence Y)" on page 1120

Use Considerations

It is important to understand the concept of stride for sequences when using these subroutines. For example, in the sort subroutines, a negative stride causes a sequence to be sorted into descending order in an array. In the search subroutines, a negative stride reverses the direction of the search. See "How Stride Is Used for Vectors" on page 73.

Performance and Accuracy Considerations

1. The binary search subroutines provide better performance than the sequential search subroutines because of the nature of the searching algorithms. However, the binary search subroutines require that, before the subroutine is called, the sequence to be searched is sorted into ascending order. Therefore, if your data is already sorted, a binary search subroutine is faster. On the other hand, if your data is in random order and the number of elements being searched for is small, a sequential search subroutine is faster than doing a sort and binary search.
2. When doing multiple invocations of the binary search subroutines, you get better overall performance from the searching algorithms by doing fewer invocations and specifying larger search element arrays for argument *x*.
3. If you do not need the results provided in array RC by these subroutine, you get better performance if you do not request it. That is, specify 0 for the *iopt* argument.

Sorting and Searching Subroutines

This contains the sorting and searching subroutine descriptions.

ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)

Purpose

These subroutines sort the elements of sequence ***x***.

Table 240: Data Types.

Data Types	
x	**Subroutine**
Integer	ISORT
Short-precision real	SSORT
Long-precision real	DSORT

Syntax

Language	**Syntax**
Fortran	CALL ISORT | SSORT | DSORT (*x*, *incx*, *n*)
C and C++	isort | ssort | dsort (*x*, *incx*, *n*);

On Entry

x

is the sequence ***x*** of length *n*, to be sorted.

Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 240 on page 1110.

incx

is the stride for both the input sequence ***x*** and the output sequence ***x***. If it is positive, elements are sorted into ascending order in the array, and if it is negative, elements are sorted into descending order in the array.

Specified as: an integer. It can have any value.

n

is the number of elements in sequence ***x***. Specified as: an integer; $n \geq 0$.

On Return

x

is the sequence ***x*** of length *n*, with its elements sorted into designated order in the array. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 240 on page 1110.

Function

The elements of input sequence ***x*** are sorted into ascending order, in place and using a partition sort. The elements of output sequence ***x*** can be expressed as follows:

$x_1 \leq x_2 \leq x_3 \leq \dots \leq x_n$

By specifying a negative stride for sequence ***x***, the elements of sequence ***x*** are assumed to be reversed in the array, (x_n, x_{n-1}, ... , x_1), thus producing a sort into descending order within the array. If *n* is 0 or 1 or if *incx* is 0, no sort is performed. See reference [“91” on page 1258].

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors
$n < 0$

Examples

Example 1

This example shows a sequence ***x*** with a positive stride.

Call Statement and Input:

```
            X  INCX  N
            |   |    |
CALL ISORT( X , 2  , 5 )

X        =  (2, . , -1, . , 5, . , 4, . , -2)
```

Output:

```
X        =  (-2, . , -1, . , 2, . , 4, . , 5)
```

Example 2

This example shows a sequence ***x*** with a negative stride.

Call Statement and Input:

```
            X   INCX  N
            |    |    |
CALL ISORT( X , -1  , 5 )

X        =  (2, -1, 5, 4, -2)
```

Output:

```
X        =  (5, 4, 2, -1, -2)
```

ISORTX, SSORTX, and DSORTX (Sort the Elements of a Sequence and Note the Original Element Positions)

Purpose

These subroutines sort the elements of sequence ***x***. The original positions of the elements in sequence ***x*** are returned in the indices array, `INDX`. Where equal elements occur in the input sequence, they do not necessarily remain in the same relative order in the output sequence.

Note: If you need a stable sort, you should use ISORTS, SSORTS, or DSORTS rather than these subroutines.

Table 241: Data Types.	
Data Types	
x	**Subroutine**
Integer	ISORTX
Short-precision real	SSORTX
Long-precision real	DSORTX

Syntax

Language	**Syntax**
Fortran	CALL ISORTX | SSORTX | DSORTX (*x*, *incx*, *n*, *indx*)
C and C++	isortx | ssortx | dsortx (*x*, *incx*, *n*, *indx*);

On Entry

x

is the sequence ***x*** of length *n*, to be sorted.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*| elements, containing numbers of the data type indicated in Table 241 on page 1112.

incx

is the stride for both the input sequence ***x*** and the output sequence ***x***. If it is positive, elements are sorted into ascending order in the array, and if it is negative, elements are sorted into descending order in the array.

Specified as: an integer. It can have any value.

n

is the number of elements in sequence ***x***. Specified as: an integer; $n \geq 0$.

indx

See On Return.

On Return

x

is the sequence ***x*** of length *n*, with its elements sorted into designated order in the array. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 241 on page 1112.

indx

is the array, referred to as `INDX`, containing the *n* indices that indicate, for the elements in the sorted output sequence, the original positions of those elements in input sequence ***x***.

Note: It is important to remember that when you specify a negative stride, ESSL assumes that the order of the input and output sequence elements in the X array is reversed; however, the elements in `INDX` are not reversed. See "Function" on page 1112.

Returned as: a one-dimensional array of length *n*, containing integers; 1 ≤ (`INDX` elements) ≤ *n*.

Function

The elements of input sequence ***x*** are sorted into ascending order, in place and using a partition sort. The elements of output sequence ***x*** can be expressed as follows:

$x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_n$

Where equal elements occur in the input sequence, they do not necessarily remain in the same relative order in the output sequence.

By specifying a negative stride for ***x***, the elements of input sequence ***x*** are assumed to be reversed in the array, (x_n, x_{n-1}, ... , x_1), thus producing a sort into descending order within the array.

In addition, the `INDX` array contains the *n* indices that indicate, for the elements in the sorted output sequence, the original positions of those elements in input sequence ***x***. (These are not the positions in the array, but rather the positions in the sequence.) For each element x_j in the input sequence, becoming element xx_k in the output sequence, the elements in `INDX` are defined as follows:

`INDX`(k) = j for j = 1, n and k = 1, n
where $xx_k = x_j$

To understand `INDX` when you specify a negative stride, you should remember that both the input and output sequences, ***x***, are assumed to be in reverse order in array X, but `INDX` is not affected by stride. The sequence elements of ***x*** are assumed to be stored in your input array as follows:

X = (x_n, x_{n-1}, ... , x_1)

The sequence elements of ***x*** are stored in your output array by ESSL as follows:

X = (xx_n, xx_{n-1}, ... , xx_1)

where the elements xx_k are the elements x_j, sorted into descending order in X. As an example of how `INDX` is calculated, if $xx_1 = x_{n-1}$, then `INDX`(1) = n-1.

If *n* is 0, no computation is performed. See reference [“91” on page 1258].

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors
$n < 0$

Examples

Example 1

This example shows how to sort a sequence ***x*** into ascending order by specifying a positive stride.

Call Statement and Input:

```
              X  INCX  N   INDX
              |   |    |    |
CALL ISORTX( X , 2  , 5 , INDX )

X        =  (2, . , -1, . , 5, . , 1, . , -2)
```

Output:

```
X        =  (-2, . , -1, . , 1, . , 2, . , 5)
INDX     =  (5, 2, 4, 1, 3)
```

Example 2

This example shows how to sort a sequence ***x*** into descending order by specifying a negative stride. Therefore, both the input and output sequences are assumed to be reversed in the array X. The input sequence is assumed to be stored as follows:

X = (x_5, x_4, x_3, x_2, x_1) = `(2, -1, 5, 1, -2)`

The output sequence is stored by ESSL as follows:

X = (xx_5, xx_4, xx_3, xx_2, xx_1) = (5, 2, 1, -1, -2)

As a result, INDX is defined as follows:

INDX = ($indx_1$, $indx_2$, $indx_3$, $indx_4$, $indx_5$) = (1, 4, 2, 5, 3)

For example, because output sequence element xx_4 = 2 is input sequence element x_5, then INDX(4) = 5.

Call Statement and Input:

```
              X   INCX  N   INDX
              |    |    |    |
CALL ISORTX( X , -1  , 5 , INDX )

X        =  (2, -1, 5, 1, -2)
```

Output:

```
X        =  (5, 2, 1, -1, -2)
INDX     =  (1, 4, 2, 5, 3)
```

ISORTS, SSORTS, and DSORTS (Sort the Elements of a Sequence Using a Stable Sort and Note the Original Element Positions)

Purpose

These subroutines sort the elements of sequence ***x*** using a stable sort; that is, where equal elements occur in the input sequence, they remain in the same relative order in the output sequence. The original positions of the elements in sequence ***x*** are returned in the indices array INDX.

Note: If you need a stable sort, then you should use these subroutines rather than ISORTX, SSORTX, or DSORTX.

Table 242: Data Types.

Data Types	
x, *work*	**Subroutine**
Integer	ISORTS
Short-precision real	SSORTS
Long-precision real	DSORTS

Syntax

Language	**Syntax**
Fortran	CALL ISORTS | SSORTS | DSORTS (*x*, *incx*, *n*, *indx*, *work*, *lwork*)
C and C++	isorts | ssorts | dsorts (*x*, *incx*, *n*, *indx*, *work*, *lwork*);

On Entry

x

is the sequence ***x*** of length *n*, to be sorted.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*| elements, containing numbers of the data type indicated in Table 242 on page 1114.

incx
is the stride for both the input sequence ***x*** and the output sequence ***x***. If it is positive, elements are sorted into ascending order in the array, and if it is negative, elements are sorted into descending order in the array.

Specified as: an integer. It can have any value.

n
is the number of elements in sequence ***x***. Specified as: an integer; $n \geq 0$.

indx
See On Return.

work
is the storage work area used by this subroutine. Its size is specified by *lwork*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 242 on page 1114.

lwork
is the size of the work area specified by *work*— that is, the number of elements in *work*.

Specified as: an integer; $lwork \geq n/2$.

Note: This is the value to achieve optimal performance. The sort is performed regardless of the value you specify for *lwork*, but you may receive an attention message.

On Return

x
is the sequence ***x*** of length *n*, with its elements sorted into designated order in the array. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 242 on page 1114.

indx
is the array, referred to as `INDX`, containing the *n* indices that indicate, for the elements in the sorted output sequence, the original positions of those elements in input sequence ***x***.

Note: It is important to remember that when you specify a negative stride, ESSL assumes that the order of the input and output sequence elements in the X array is reversed; however, the elements in `INDX` are not reversed. See "Function" on page 1115.

Returned as: a one-dimensional array of length *n*, containing integers; $1 \leq$ (`INDX` elements) $\leq n$.

Function

The elements of input sequence ***x*** are sorted into ascending order using a partition sort. The sorting is stable; that is, where equal elements occur in the input sequence, they remain in the same relative order in the output sequence. The elements of output sequence ***x*** can be expressed as follows:

$x_1 \leq x_2 \leq x_3 \leq \dots \leq x_n$

By specifying a negative stride for ***x***, the elements of input sequence ***x*** are assumed to be reversed in the array, $(x_n, x_{n-1}, \dots, x_1)$, thus producing a sort into descending order within the array.

In addition, the `INDX` array contains the *n* indices that indicate, for the elements in the sorted output sequence, the original positions of those elements in input sequence ***x***. (These are not the positions in the array, but rather the positions in the sequence.) For each element x_j in the input sequence, becoming element xx_k in the output sequence, the elements in `INDX` are defined as follows:

`INDX`$(k) = j$ for $j = 1, n$ and $k = 1, n$
where $xx_k = x_j$

To understand `INDX` when you specify a negative stride, you should remember that both the input and output sequences, ***x***, are assumed to be in reverse order in array X, but `INDX` is not affected by stride. The sequence elements of ***x*** are assumed to be stored in your input array as follows:

X = (x_n, x_{n-1}, ... , x_1)

The sequence elements of ***x*** are stored in your output array by ESSL as follows:

X = (xx_n, xx_{n-1}, ... , xx_1)

where the elements xx_k are the elements x_j, sorted into descending order in X. As an example of how `INDX` is calculated, if $xx_1 = x_{n-1}$, then `INDX`(1) = n-1.

If n is 0, no computation is performed. See references [“38” on page 1255] and [“91” on page 1258].

Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors

$n < 0$

Examples

Example 1

This example shows how to sort a sequence ***x*** into ascending order by specifying a positive stride. Because this is a stable sort, the -1 elements remain in the same relative order in the output sequence, indicated by `INDX(2)` = 2 and `INDX(3)` = 4.

Call Statement and Input:

```
              X  INCX  N   INDX   WORK  LWORK
              |   |    |    |      |      |
CALL ISORTS( X , 2   , 5 , INDX , WORK ,  5  )

X        =  (2, . , -1, . , 5, . , -1, . , -2)
```

Output:

```
X        =  (-2, . , -1, . , -1, . , 2, . , 5)
INDX     =  (5, 2, 4, 1, 3)
```

Example 2

This example shows how to sort a sequence ***x*** into descending order by specifying a negative stride. Therefore, both the input and output sequences are assumed to be reversed in the array X. The input sequence is assumed to be stored as follows:

X = (x_5, x_4, x_3, x_2, x_1) = `(2, -1, 5, -1, -2)`

The output sequence is stored by ESSL as follows:

X = (xx_5, xx_4, xx_3, xx_2, xx_1) = `(5, 2, -1, -1, -2)`

As a result, `INDX` is defined as follows:

`INDX` = ($indx_1$, $indx_2$, $indx_3$, $indx_4$, $indx_5$) = `(1, 2, 4, 5, 3)`

For example, because output sequence element xx_4 = 2 is input sequence element x_5, then `INDX`(4) = 5. Also, because this is a stable sort, the -1 elements remain in the same relative order in the output sequence, indicated by `INDX(2)` = 2 and `INDX(3)` = 4.

Call Statement and Input:

```
              X   INCX  N   INDX   WORK  LWORK
              |    |    |    |      |      |
CALL ISORTS( X , -1  , 5 , INDX , WORK ,  5  )
```

```
X        =  (2, -1, 5, -1, -2)
```

Output:

```
X        =  (5, 2, -1, -1, -2)
INDX     =  (1, 2, 4, 5, 3)
```

IBSRCH, SBSRCH, and DBSRCH (Binary Search for Elements of a Sequence X in a Sorted Sequence Y)

Purpose

These subroutines perform a binary search for the locations of the elements of sequence ***x*** in another sequence ***y***, where ***y*** has been sorted into ascending order. The first occurrence of each element is found. When an exact match is not found, the position of the next larger element in ***y*** is indicated. The locations are returned in the indices array `INDX`, and, optionally, return codes indicating whether the exact elements were found are returned in array RC.

Table 243: Data Types.

Data Types	
x, ***y***	**Subroutine**
Integer	IBSRCH
Short-precision real	SBSRCH
Long-precision real	DBSRCH

Syntax

Language	Syntax
Fortran	CALL IBSRCH | SBSRCH | DBSRCH (*x*, *incx*, *n*, *y*, *incy*, *m*, *indx*, *rc*, *iopt*)
C and C++	ibsrch | sbsrch | dbsrch (*x*, *incx*, *n*, *y*, *incy*, *m*, *indx*, *rc*, *iopt*);

On Entry

x

is the sequence ***x*** of length *n*, containing the elements for which sequence ***y*** is searched.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 243 on page 1117. It must have at least 1+(*n*-1)|*incx*| elements.

incx

is the stride for sequence ***x***.

Specified as: an integer. It can have any value.

n

is the number of elements in sequence ***x*** and arrays `INDX` and RC.

Specified as: an integer; $n \geq 0$.

y

is the sequence ***y*** of length *m*, to be searched, where ***y*** must be sorted into ascending order.

Note: Be careful in specifying the stride for sequence ***y***. A negative stride reverses the direction of the search, because the order of the sequence elements is reversed in the array.

Specified as: a one-dimensional array of (at least) length 1+(*m*-1)|*incy*|, containing numbers of the data type indicated in Table 243 on page 1117.

incy

is the stride for sequence ***y***.

Specified as: an integer. It can have any value.

m

is the number of elements in sequence ***y***. Specified as: an integer; *m* ≥ 0.

indx

See On Return.

rc

See On Return.

iopt

has the following meaning, where:

If *iopt* = 0, the *rc* argument is not used in the computation.

If *iopt* = 1, the *rc* argument is used in the computation.

Specified as: an integer; *iopt* = 0 or 1.

On Return

indx

is the array, referred to as `INDX`, containing the *n* indices that indicate the positions of the elements of sequence ***x*** in sequence ***y***. The first occurrence of the element found in sequence ***y*** is indicated in array `INDX`. When an exact match between an element of sequence ***x*** and an element of sequence ***y*** is not found, the position of the next larger element in sequence ***y*** is indicated. When the element in sequence ***x*** is larger than all the elements in sequence ***y***, then *m*+1 is indicated in array `INDX`.

Returned as: a one-dimensional array of length *n*, containing integers; 1 ≤ (`INDX` elements) ≤ *m*+1.

rc

has the following meaning, where:

If *iopt* = 0, then *rc* is not used, and its contents remain unchanged.

If *iopt* = 1, it is the array, referred to as RC, containing the *n* return codes that indicate whether the elements in sequence ***x*** were found in sequence ***y***. For *i* = 1, *n*, elements RC(*i*) = 0 if x_i matches an element in sequence ***y***, and RC(*i*) = 1 if an exact match is not found in sequence ***y***.

Returned as: a one-dimensional array of length *n*, containing integers; RC(*i*) = 0 or 1.

Notes

1. The elements of ***y*** must be sorted into ascending order; otherwise, results are unpredictable. For details on how to do this, see "ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)" on page 1110.
2. If you do not need the results provided in array RC by these subroutines, you get better performance if you do not request it. That is, specify 0 for the *iopt* argument.

Function

These subroutines perform a binary search for the first occurrence (or last occurrence, using negative stride) of the locations of the elements of sequence ***x*** in another sequence ***y***, where ***y*** must be sorted into ascending order before calling this subroutine. The first occurrence of each element is found. Two arrays are returned, containing the results of the binary searches:

- `INDX`, the indices array, contains the positions of the elements of sequence ***x*** in sequence ***y***. When an exact match between values of elements in sequences ***x*** and ***y*** is not found, the location of the next larger element in sequence ***y*** is indicated in array `INDX`.

- RC, the return codes array, indicates for each element in sequence ***x*** whether the exact element was found in sequence ***y***. If you do not need these results, you get better performance if you set *iopt* = 0.

The results returned for the INDX and RC arrays are expressed as follows:

```
For i = 1, n

for all $y_j \geq x_i$, j = 1, m , INDX(i) = min(j)
if all $y_j < x_i$, j = 1, m , INDX(i) = m+1

And for i = 1, n

if $x_i = y_{INDX(i)}$, RC(i) = 0
if $x_i \neq y_{INDX(i)}$, RC(i) = 1
```

where:

x is a sequence of length *n*, containing the search elements
y is a sequence of length *m* to be searched. It must be sorted into ascending order
INDX is the array of length *n* of indices
RC is the array of length *n* of return codes

See reference [“91” on page 1258]. If *n* is 0, no search is performed. If *m* is 0, then:

INDX(*i*) = 1 and RC(*i*) = 1 for *i* = 1, *n*

It is important to note that a negative stride for sequence ***y*** reverses the direction of the search, because the order of the sequence elements is reversed in the array. For more details on sorting sequences, see “Function” on page 1110.

Error conditions

Computational Errors
None

Input-Argument Errors

1. *n* < 0
2. *m* < 0
3. *iopt* ≠ 0 or 1

Examples

Example 1

This example shows a search where sequences ***x*** and ***y*** have positive strides, and where the optional return codes are returned as part of the output.

Call Statement and Input:

```
              X  INCX  N   Y  INCY  M    INDX   RC  IOPT
              |   |    |   |   |    |     |     |    |
CALL IBSRCH( X , 2  , 5 , Y , 1  , 10 , INDX , RC , 1 )

X        =  (-3, . , 125, . , 30, . , 20, . , 70)
Y        =  (10, 20, 30, 30, 40, 50, 60, 80, 90, 100)
```

Output:

```
INDX     =  (1, 11, 3, 2, 8)
RC       =  (1, 1, 0, 0, 1)
```

Example 2

This example shows the same calling sequence as in Example 1, except that it includes the IOPT argument, specified as 1. This is equivalent to using the calling sequence in Example 1 and gives the same results.

Call Statement and Input:

```
              X  INCX  N   Y  INCY  M    INDX   RC  IOPT
              |   |    |   |   |    |     |     |    |
CALL IBSRCH( X , 2  , 5 , Y , 1  , 10 , INDX , RC , 1  )
```

Example 3

This example shows a search where sequence ***x*** has a negative stride, and sequence ***y*** has a positive stride. The optional return codes are not requested, because IOPT is specified as 0.

Call Statement and Input:

```
              X   INCX  N   Y  INCY  M    INDX   RC  IOPT
              |    |    |   |   |    |     |     |    |
CALL IBSRCH( X , -2  , 5 , Y , 1  , 10 , INDX , RC , 0 )

X        =  (-3, . , 125, . , 30, . , 20, . , 70)
Y        =  (10, 20, 30, 30, 40, 50, 60, 80, 90, 100)
```

Output:

```
INDX     =  (8, 2, 3, 11, 1)
RC       =(not relevant)
```

Example 4

This example shows a search where sequence ***x*** has a positive stride, and sequence ***y*** has a negative stride. As shown below, elements of ***y*** are in descending order in array Y. The optional return codes are not requested, because IOPT is specified as 0.

Call Statement and Input:

```
              X  INCX  N   Y   INCY  M    INDX   RC  IOPT
              |   |    |   |    |    |     |     |    |
CALL IBSRCH( X , 2  , 5 , Y , -1  , 10 , INDX , RC , 0  )
```

X = (-3, . , 125, . , 30, . , 20, . , 70)
Y = (100, 90, 80, 60, 50, 40, 30, 30, 20, 10)
RC =(not relevant)

Output:

```
INDX     =  (1, 11, 3, 2, 8)
```

ISSRCH, SSSRCH, and DSSRCH (Sequential Search for Elements of a Sequence X in the Sequence Y)

Purpose

These subroutines perform a sequential search for the locations of the elements of sequence ***x*** in another sequence ***y***. Depending on the sign of the *idir* argument, the search direction indicator, the location of either the first or last occurrence of each element is indicated in the resulting indices array INDX. When an exact match between elements is not found, the position is indicated as 0.

Table 244: Data Types.

Data Types	
x, ***y***	**Subroutine**
Integer	ISSRCH
Short-precision real	SSSRCH

Table 244: Data Types. (continued)

Data Types	
Long-precision real	DSSRCH

Syntax

Language	Syntax
Fortran	CALL ISSRCH | SSSRCH | DSSRCH (*x*, *incx*, *n*, *y*, *incy*, *m*, *idir*, *indx*)
C and C++	issrch | sssrch | dssrch (*x*, *incx*, *n*, *y*, *incy*, *m*, *idir*, *indx*);

On Entry

x

is the sequence ***x*** of length *n*, containing the elements for which sequence ***y*** is searched.

Specified as: a one-dimensional array of (at least) length 1+(*n*-1)|*incx*|, containing numbers of the data type indicated in Table 244 on page 1120.

incx

is the stride for sequence ***x***.

Specified as: an integer. It can have any value.

n

is the number of elements in sequence ***x*** and array `INDX`.

Specified as: an integer; $n \ge 0$.

y

is the sequence ***y*** of length *m* to be searched.

Note: Be careful in specifying the stride for sequence ***y***. A negative stride reverses the direction of the search, because the order of the sequence elements is reversed in the array.

Specified as: a one-dimensional array of (at least) length 1+(*m*-1)|*incy*|, containing numbers of the data type indicated in Table 244 on page 1120.

incy

is the stride for sequence ***y***.

Specified as: an integer. It can have any value.

m

is the number of elements in sequence ***y***. Specified as: an integer; $m \ge 0$.

idir

indicates the search direction, where:

If $idir \ge 0$, sequence ***y*** is searched from the first element to the last (1, *n*), thus finding the first occurrence of the element in the sequence.

If $idir < 0$, sequence ***y*** is searched from the last element to the first (*n*, 1), thus finding the last occurrence of the element in the sequence.

Specified as: an integer. It can have any value.

indx

See On Return.

On Return

indx

is the array, referred to as `INDX`, containing the *n* indices that indicate the positions of the elements of sequence ***x*** in sequence ***y***, where:

If *idir* ≥ 0, the first occurrence of the element found in sequence ***y*** is indicated in array INDX.

If *idir* < 0, the last occurrence of the element found in sequence ***y*** is indicated in array INDX.

In all cases, if no match is found, 0 is indicated in array INDX.

Returned as: a one-dimensional array of length *n*, containing integers; 0 ≤ (INDX elements) ≤ *m*.

Function

These subroutines perform a sequential search for the first occurrence (or last occurrence, using a negative *idir*) of the locations of the elements of sequence ***x*** in another sequence ***y***. The results of the sequential searches are returned in the indices array INDX, indicating the positions of the elements of sequence ***x*** in sequence ***y***. The positions indicated in array INDX are calculated relative to the first sequence element position—that is, the position of y_1. When an exact match between values of elements in sequences ***x*** and ***y*** is not found, 0 is indicated in array INDX for that position.

The results returned in array INDX are expressed as follows:

For $i = 1, n$

for all $y_j = x_i$, $j = 1, m$
INDX(i) = min(j), if $idir \geq 0$
INDX(i) = max(j), if $idir < 0$

if all $y_j \neq x_i$, $j = 1, m$
INDX(i) = 0

where:

x is a sequence of length *n*, containing the search elements.
y is a sequence of length *m* to be searched.
INDX is the array of length *n* of indices.

See reference [“91” on page 1258]. If *n* is 0, no search is performed.

It is important to note that a negative stride for sequence ***y*** reverses the direction of the search, because the order of the sequence elements is reversed in the array.

Error conditions

Computational Errors
None

Input-Argument Errors

1. $n < 0$
2. $m < 0$

Examples

Example 1

This example shows a search where sequences ***x*** and ***y*** have positive strides, and the search direction indicator, *idir*, is positive.

Call Statement and Input:

```
              X  INCX  N   Y  INCY  M  IDIR  INDX
              |   |    |   |   |    |   |     |
CALL ISSRCH( X , 1  , 3 , Y , 2  , 8 , 1  , INDX )

X        =  (0, 12, 3)
Y        =  (0, . , 8, . , 12, . , 0, . , 1, . , 4, . , 0, . , 2)
```

Output:

```
INDX     =  (1, 3, 0)
```

Example 2

This example shows a search where sequences ***x*** and ***y*** have positive strides, and the search direction indicator, *idir*, is negative.

Call Statement and Input:

```
             X  INCX  N   Y  INCY  M   IDIR  INDX
             |   |    |   |   |    |    |     |
CALL ISSRCH( X , 2  , 3 , Y , 2  , 8 , -1  , INDX )

X        =  (0, . , 12, . , 3)
Y        =  (0, . , 8, . , 12, . , 0, . , 1, . , 4, . , 0, . , 2)
```

Output:

```
INDX     =  (7, 3, 0)
```

Example 3

This example shows a search where sequences ***x*** and ***y*** have negative strides, and the search direction indicator, *idir*, is positive.

Call Statement and Input:

```
             X   INCX  N   Y   INCY  M  IDIR  INDX
             |    |    |   |    |    |   |     |
CALL ISSRCH( X , -1  , 3 , Y , -2  , 8 , 1  , INDX )

X        =  (0, 12, 3)
Y        =  (0, . , 8, . , 12, . , 0, . , 1, . , 4, . , 0, . , 2)
```

Output:

```
INDX     =  (0, 6, 2)
```

Example 4

This example shows a search where sequences ***x*** and ***y*** have negative strides, and the search direction indicator, *idir*, is negative.

Call Statement and Input:

```
             X   INCX  N   Y   INCY  M   IDIR  INDX
             |    |    |   |    |    |    |     |
CALL ISSRCH( X , -2  , 3 , Y , -1  , 8 , -1  , INDX )

X        =  (0, . , 12, . , 3)
Y        =  (0, 8, 12, 0, 1, 4, 0, 2)
```

Output:

```
INDX     =  (0, 6, 8)
```

Chapter 14. Interpolation subroutines

The interpolation subroutines are described here.

Overview of the Interpolation Subroutines

The interpolation subroutines provide the capabilities of doing polynomial interpolation, local polynomial interpolation, and one- and two-dimensional cubic spline interpolation (Table 245 on page 1125).

Table 245: List of Interpolation Subroutines.

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Location
Polynomial Interpolation	SPINT	DPINT	"SPINT and DPINT (Polynomial Interpolation)" on page 1126
Local Polynomial Interpolation	STPINT	DTPINT	"STPINT and DTPINT (Local Polynomial Interpolation)" on page 1130
Cubic Spline Interpolation	SCSINT	DCSINT	"SCSINT and DCSINT (Cubic Spline Interpolation)" on page 1133
Two-Dimensional Cubic Spline Interpolation	SCSIN2	DCSIN2	"SCSIN2 and DCSIN2 (Two-Dimensional Cubic Spline Interpolation)" on page 1137

Use Considerations

Polynomial interpolation (SPINT and DPINT) is a global scheme. As the number of data points increases, the degree of the interpolating polynomial is raised; therefore, the graph of the interpolating polynomial tends to be oscillatory.

Local polynomial interpolation (STPINT and DTPINT) is a local scheme. The data generated is affected only by locally grouped data points. The degree of the local interpolating polynomial is usually lower than a global interpolating polynomial.

Performance and Accuracy Considerations

1. Doing extrapolation with SPINT and DPINT is not encouraged unless you know the consequences of doing polynomial extrapolation.
2. If performance is the overriding consideration, you should investigate using the general signal processing subroutines, DQINT and SQINT.
3. There are some ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 61.

Interpolation Subroutines

This contains the interpolation subroutine descriptions.

SPINT and DPINT (Polynomial Interpolation)

Purpose

These subroutines compute the Newton divided difference coefficients and perform a polynomial interpolation through a set of data points at specified abscissas.

Table 246: Data Types.

Data Types	
x, ***y***, ***c***, ***t***, ***s***	**Subroutine**
Short-precision real	SPINT
Long-precision real	DPINT

Syntax

Language	Syntax
Fortran	CALL SPINT | DPINT (*x*, *y*, *n*, *c*, *ninit*, *t*, *s*, *m*)
C and C++	spint | dpint (*x*, *y*, *n*, *c*, *ninit*, *t*, *s*, *m*);

On Entry

x

is the vector ***x*** of length *n*, containing the abscissas of the data points used in the interpolations. The elements of ***x*** must be distinct.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 246 on page 1126.

y

is the vector ***y*** of length *n*, containing the ordinates of the data points used in the interpolations.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 246 on page 1126.

n

is the number of elements in vectors ***x, y,*** and ***c***—that is, the number of data points. Specified as: an integer; $n \geq 0$.

c

is the vector ***c*** of length *n*, where:

If $ninit \leq 0$, all elements of ***c*** are undefined on entry.

If $ninit > 0$, ***c*** contains the Newton divided difference coefficients, c_j for $j = 1, ninit$, for the interpolating polynomial through the data points (x_j,y_j) for $j = 1, ninit$. If $ninit < n$, the values of c_j for $j = ninit+1, n$ are undefined.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 246 on page 1126.

ninit

indicates the following:

If $ninit \leq 0$, this is the first call to this subroutine with the data in ***x*** and ***y***; therefore, none of the Newton divided difference coefficients in ***c*** have been initialized.

If $ninit > 0$, a previous call to this subroutine was made with the data points (x_j, y_j) for $j = 1, ninit$, where:

- If *ninit* = *n*, all the Newton divided difference coefficients in ***c*** were computed for the data points. No additional coefficients are computed on this entry.
- If *ninit* < *n*, the first *ninit* Newton divided difference coefficients in ***c*** were computed for the data points (x_j, y_j) for *j* = 1, *ninit*. The coefficients are updated for the additional data points (x_j, y_j) for *j* = *ninit*+1, *n* on this entry.

Specified as: an integer; *ninit* ≤ *n*.

t

is the vector ***t*** of length *m*, containing the abscissas at which interpolation is to be done.

Specified as: a one-dimensional array of (at least) length *m*, containing numbers of the data type indicated in Table 246 on page 1126.

s

See On Return.

m

is the number of elements in vectors ***t*** and ***s***—that is, the number of interpolations to be performed.

Specified as: an integer; *m* ≥ 0.

On Return

c

is the vector ***c*** of length *n*, containing the coefficients of the Newton divided difference form of the interpolating polynomial through the data points (x_j,y_j) for *j* = 1, *n*. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 246 on page 1126.

ninit

is the number of coefficients, *n*, in output vector ***c***. (If you call this subroutine again with the same data, this value should be specified for *ninit*.) Returned as: an integer; *ninit* = *n*.

s

is the vector ***s*** of length *m*, containing the resulting interpolated values; that is, each s_i is the value of the interpolating polynomial evaluated at t_i. Returned as: a one-dimensional array of (at least) length *m*, containing numbers of the data type indicated in Table 246 on page 1126.

Notes

1. In your C program, argument *ninit* must be passed by reference.
2. Vectors ***x, y,*** and ***t*** must have no common elements with vectors ***c*** and ***s***, and vector ***c*** must have no common element with vector ***s***; otherwise, results are unpredictable.
3. The elements of vector ***x*** must be distinct; that is, $x_i \neq x_j$ if $i \neq j$ for *i*, *j* = 1, *n*.

Function

Polynomial interpolation is performed at specified abscissas, t_i for *i* = 1, *m*, in vector ***t***, using the method of Newton divided differences through the data points:

(x_j, y_j) for *j* = 1, *n*

where:

x_j are elements of vector ***x***.
y_j are elements of vector ***y***.

The interpolated value at each t_i is returned in s_i for *i* = 1, *m*. See references [“23” on page 1254] and [“65” on page 1256]. The interpolating values returned in ***s*** are computed using the Newton divided difference coefficients, as defined here.

The divided difference coefficients, c_j for *j* = 1, *n*, are returned in vector ***c***. These coefficients can then be reused on subsequent calls to this subroutine, using the same data points (x_j, y_j), but with new values of t_i.

If the number of data points is increased from one call this subroutine to the next, the new coefficients are computed, and the existing coefficients are updated (not recomputed). This feature can be used to test for the convergence of the interpolations through a sequence of an increasingly larger set of points.

The values specified for *ninit* and *m* indicate which combination of functions are performed by this subroutine: computing the coefficients, performing the interpolation, or both. If $m = 0$, only the divided difference coefficients are computed. No interpolation is performed. If $n = 0$, no computation or interpolation is performed.

For SPINT, the Newton divided differences and interpolating values are accumulated in long precision.

Newton Divided Differences and Interpolating Values:

The Newton divided differences of the following data points:

(x_j, y_j) for $j = 1, n$
where $x_j \neq x_l$ if $j \neq l$ for $j, l = 1, n$

are denoted by $\delta_k y_j$ for $k = 0, 1, 2, ..., n-1$ and $j = 1, 2, ..., n-k$, and are defined as follows:

For $k = 0$ and 1:
$\delta_0 y_j = y_j$ for $j = 1, 2, ..., n$
$\delta_1 y_j = (y_{j+1} - y_j) / (x_{j+1} - x_j)$ for $j = 1, 2, ..., n-1$

For $k = 2, 3, ..., n-1$:
$\delta_k y_j = (\delta_{k-1} y_{j+1} - \delta_{k-1} y_j) / (x_{j+k} - x_j)$ for $j = 1, 2, ..., n-k$

The value *s* of the Newton divided difference form of the interpolating polynomial evaluated at an abscissa *t* is given by:

$s = y_n + (t-x_n)\ \delta_1 y_{n-1}$
$+ (t-x_{n-1})\ (t-x_n)\ \delta_2 y_{n-2}$
$+ ...+(t-x_2)\ (t-x_3)\ ...\ (t-x_n)\ \delta_{n-1} y_1$

Therefore, on output, the coefficients in vector ***c*** are as follows:

$c_n = y_n$
$c_{n-1} = \delta_1 y_{n-1}$
$c_{n-2} = \delta_2 y_{n-2}$
.
.
.
$c_1 = \delta_{n-1} y_1$

Also, the interpolating values in ***s***, in terms of ***c***, are as follows for $i = 1, m$:

$s_i = c_n + (t_i-x_n)\ c_{n-1}$
$+ (t_i-x_{n-1})\ (t_i-x_n)\ c_{n-2}$
$+ ...$
$+ (t_i-x_2)\ (t_i-x_3)\ ...\ (t_i-x_n)\ c_1$

Error conditions

Computational Errors
None

Input-Argument Errors

1. $n < 0$
2. $ninit > n$
3. $m < 0$

Examples

Example 1

This example shows a quadratic polynomial interpolation on the initial call with the specified data points; that is, NINIT = 0, and C contains all undefined values. On output, NINIT and C are updated with new values.

Call Statement and Input:

```
            X   Y   N   C  NINIT  T   S   M
            |   |   |   |    |    |   |   |
CALL SPINT( X , Y , 3 , C ,  0  , T , S , 2 )
```

```
X        =  (-0.50, 0.00, 1.00)
Y        =  (0.25, 0.00, 1.00)
C        =  ( . , . , . )
T        =  (-0.2, 0.2)
```

Output:

```
C        =  (1.00, 1.00, 1.00)
NINIT    =  3
S        =  (0.04, 0.04)
```

Example 2

This example shows a quadratic polynomial interpolation on a subsequent call with the same data points specified in Example 1, but using a different set of abscissas in T. In this case, NINIT = N = 3, and C contains the values defined on output in Example 1. On output here, the values in NINIT and C are unchanged.

Call Statement and Input:

```
            X   Y   N   C  NINIT  T   S   M
            |   |   |   |    |    |   |   |
CALL SPINT( X , Y , 3 , C ,  3  , T , S , 2 )
```

```
X        =  (-0.50, 0.00, 1.00)
Y        =  (0.25, 0.00, 1.00)
C        =  (1.00, 1.00, 1.00)
T        =  (-0.10, 0.10)
```

Output:

```
C        =  (1.00, 1.00, 1.00)
NINIT    =  3
S        =  (0.01, 0.01)
```

Example 3

This example is the same as Example 2 except that it specifies additional data points on the subsequent call to the subroutine. In this case, 0 < NINIT < N. On output here, the values in NINIT and C are updated. The interpolating polynomial is a degree of 4.

Call Statement and Input:

```
            X   Y   N   C  NINIT  T   S   M
            |   |   |   |    |    |   |   |
CALL SPINT( X , Y , 5 , C ,  3  , T , S , 2 )
```

```
X        =  (-0.50, 0.00, 1.00, -1.00, 0.50)
Y        =  (0.25, 0.00, 1.00, 1.10, 0.26)
C        =  (1.00, 1.00, 1.00, . , . )
T        =  (-0.10, 0.10)
```

Output:

```
C        =  (0.04, -0.06, 1.02, -0.56, 0.26)
NINIT    =  5
S        =  (0.0072, 0.0130)
```

STPINT and DTPINT (Local Polynomial Interpolation)

Purpose

These subroutines perform a polynomial interpolation at specified abscissas, using data points selected from a table of data.

Table 247: Data Types.

Data Types	
x, ***y***, ***t***, ***s***, *aux*	**Subroutine**
Short-precision real	STPINT
Long-precision real	DTPINT

Syntax

Language	**Syntax**
Fortran	CALL STPINT | DTPINT (*x*, *y*, *n*, *nint*, *t*, *s*, *m*, *aux*, *naux*)
C and C++	stpint | dtpint (*x*, *y*, *n*, *nint*, *t*, *s*, *m*, *aux*, *naux*);

On Entry

x

is the vector ***x*** of length *n*, containing the abscissas of the data points used in the interpolations. The elements of ***x*** must be distinct and sorted into ascending order.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 247 on page 1130.

y

is the vector ***y*** of length *n*, containing the ordinates of the data points used in the interpolations.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 247 on page 1130.

n

is the number of elements in vectors ***x*** and ***y***—that is, the number of data points. Specified as: an integer; $n \geq 0$.

nint

is the number of data points to be used in the interpolation at any given point.

Specified as: an integer; $0 \leq nint \leq n$.

t

is the vector ***t*** of length *m*, containing the abscissas at which interpolation is to be done. For optimal performance, ***t*** should be sorted into ascending order.

Specified as: a one-dimensional array of (at least) length *m*, containing numbers of the data type indicated in Table 247 on page 1130.

s

See On Return.

m
: is the number of elements in vectors ***t*** and ***s***—that is, the number of interpolations to be performed.

  Specified as: an integer; $m \geq 0$.

aux
: has the following meaning:

  If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

  Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

  Specified as: an area of storage, containing numbers of the data type indicated in Table 247 on page 1130. On output, the contents are overwritten.

naux
: is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

  Specified as: an integer, where:

  If *naux* = 0 and error 2015 is unrecoverable, STPINT and DTPINT dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

  Otherwise, it must have the following value:

  For STPINT

  For 32-bit integer arguments
  $naux \geq nint + m$

  For 64-bit integer arguments
  $naux \geq nint + 2m$

  For DTPINT
  $naux \geq nint + m$

On Return

s
: is the vector ***s*** of length *m*, containing the resulting interpolated values; that is, each s_i is the value of the interpolating polynomial evaluated at t_i. Returned as: a one-dimensional array of (at least) length *m*, containing numbers of the data type indicated in Table 247 on page 1130.

Notes

1. Vectors ***x, y,*** and ***t*** must have no common elements with vector ***s*** or work area *aux*; otherwise, results are unpredictable. See "Vector concepts" on page 71.
2. The elements of vector ***x*** must be distinct and must be sorted into ascending order; that is, $x_1 < x_2 < ... < x_n$. Otherwise, results are unpredictable. For details on how to do this, see "ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)" on page 1110.
3. The elements of vector ***t*** should be sorted into ascending order; that is, $t_1 \leq t_2 \leq t_3 \leq ... \leq t_m$. Otherwise, performance is affected.
4. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

Polynomial interpolation is performed at specified abscissas, t_i for $i = 1, m$, in vector ***t***, using *nint* points selected from the following data:

(x_j, y_j) for $j = 1, n$

where:

$x_1 < x_2 < x_3 < ... < x_n$
x_j are elements of vector ***x***.
y_j are elements of vector ***y***.

The points (x_j, y_j), used in the interpolation at a given abscissa t_i, are chosen as follows, where $k = nint/2$:

For $t_i \leq x_{k+1}$, the first *nint* points are used.
For $t_i > x_{n-nint+k}$, the last *nint* points are used.
Otherwise, points *h* through *h*+*nint*-1 are used, where:

$x_{h+k-1} < t_i \leq x_{h+k}$

The interpolated value at each t_i is returned in s_i for $i = 1, m$. See references [“23” on page 1254] and [“65” on page 1256]. If *n, nint*, or *m* is 0, no computation is performed. For a definition of the polynomial interpolation function performed through a set of data points, see “Function” on page 1127.

For STPINT, the Newton divided differences and interpolating values are accumulated in long precision.

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors
None

Input-Argument Errors

1. *n* < 0
2. *nint* < 0 or *nint* > *n*
3. *m* < 0
4. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value specified in the syntax for this argument. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows interpolation using two data points—that is, linear interpolation—at each t_i value.

Call Statement and Input:

```
             X   Y   N   NINT   T   S   M   AUX  NAUX
             |   |   |     |    |   |   |    |    |
CALL STPINT( X , Y , 10 ,  2  , T , S , 5 , AUX , 7  )

X        =  (0.0, 0.4, 1.0, 1.5, 2.1, 2.6, 3.0, 3.4, 3.9, 4.3)
Y        =  (1.0, 2.0, 3.0, 4.0, 5.0, 5.0, 4.0, 3.0, 2.0, 1.0)
T        =  (-1.0, 0.1, 1.1, 1.2, 3.9)
```

Output:

```
S        =  (-1.5000, 1.2500, 3.2000, 3.4000, 2.0000)
```

Example 2

This example shows interpolation using three data points—that is, quadratic interpolation—at each t_i value.

Call Statement and Input:

```
             X   Y   N   NINT   T   S   M   AUX  NAUX
             |   |   |     |    |   |   |    |    |
CALL STPINT( X , Y , 10 ,  3  , T , S , 5 , AUX , 8  )
```

```
X        =  (0.0, 0.4, 1.0, 1.5, 2.1, 2.6, 3.0, 3.4, 3.9, 4.3)
Y        =  (1.0, 2.0, 3.0, 4.0, 5.0, 5.0, 4.0, 3.0, 2.0, 1.0)
T        =  (-1.0, 0.1, 1.1, 1.2, 3.9)
```

Output:

```
S        =  (-2.6667, 1.2750, 3.2121, 3.4182, 2.0000)
```

SCSINT and DCSINT (Cubic Spline Interpolation)

Purpose

These subroutines compute the coefficients of the cubic spline through a set of data points and evaluate the spline at specified abscissas.

Table 248: Data Types.

Data Types	
x, ***y***, ***C***, ***t***, ***s***	**Subroutine**
Short-precision real	SCSINT
Long-precision real	DCSINT

Syntax

Language	Syntax
Fortran	CALL SCSINT | DCSINT (*x*, *y*, *c*, *n*, *init*, *t*, *s*, *m*)
C and C++	scsint | dcsint (*x*, *y*, *c*, *n*, *init*, *t*, *s*, *m*);

On Entry

x

is the vector ***x*** of length *n*, containing the abscissas of the data points that define the spline. The elements of ***x*** must be distinct and sorted into ascending order. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 248 on page 1133.

y

is the vector ***y*** of length *n*, containing the ordinates of the data points that define the spline.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 248 on page 1133.

c

is the matrix ***C*** with elements c_{jk} for $j = 1, n$ and $k = 1, 4$ that contain the following:

If *init* ≤ 0, all elements of ***c*** are undefined on entry.

If *init* = 1, c_{11} contains the spline derivative at x_1.

If *init* = 2, c_{21} contains the spline derivative at x_n.

If *init* = 3, c_{11} contains the spline derivative at x_1, and c_{21} contains the spline derivative at x_n.

If *init* > 3, ***c*** contains the coefficients of the spline computed for the data points (x_j,yj) for $j = 1, n$ on a previous call to this subroutine.

Specified as: an *n* by (at least) 4 array, containing numbers of the data type indicated in Table 248 on page 1133.

n

is the number of elements in vectors ***x*** and ***y*** and the number of rows in matrix ***C***—that is, the number of data points.

Specified as: an integer; $n \geq 0$.

init

indicates the following, where in those cases for uninitialized coefficients, this is the first call to this subroutine with the data in ***x*** and ***y***:

If *init* ≤ 0, the coefficients are uninitialized. The second derivatives of the spline at x_1 and x_n are set to zero. (These are free end conditions, also called natural boundary conditions.)

If *init* = 1, the coefficients are uninitialized. The value in c_{11} is used as the spline derivative at x_1.

If *init* = 2, the coefficients are uninitialized. The value in c_{21} is used as the spline derivative at x_n.

If *init* = 3, the coefficients are uninitialized. The value in c_{11} is used as the spline derivative at x_1 and the value in c_{21} is used as the spline derivative at x_n.

If *init* > 3, the coefficients in ***c*** were computed for data points (x_j, y_j) for $j = 1, n$ on a previous call to this subroutine.

Specified as: an integer. It can have any value.

t

is the vector ***t*** of length *m*, containing the abscissas at which the spline is evaluated.

Specified as: a one-dimensional array of (at least) length *m*, containing numbers of the data type indicated in Table 248 on page 1133.

s

See On Return.

m

is the number of elements in vectors ***t*** and ***s***—that is, the number of points at which the spline interpolation is evaluated.

Specified as: an integer; $m \geq 0$.

On Return

c

is the matrix ***C***, containing the coefficients of the spline through the data points (x_j,y_j) for $j = 1, n$. Returned as: an *n* by (at least) 4 array, containing numbers of the data type indicated in Table 248 on page 1133.

init

is an indicator that is set to indicate that the coefficients have been initialized. (If you call this subroutine again with the same data, this value should be specified for *init*.) Returned as: an integer; *init* = 4.

s

is the vector ***s*** of length *m*, containing the resulting values of the spline; that is, each s_i is the value of the spline evaluated at t_i. Returned as: a one-dimensional array of (at least) length *m*, containing numbers of the data type indicated in Table 248 on page 1133.

Notes

1. In your C program, argument *init* must be passed by reference.
2. Vectors ***x, y,*** and ***t*** must have no common elements with matrix ***C*** and vector ***s***, and matrix ***C*** must have no common elements with vector ***s***; otherwise, results are unpredictable.
3. The elements of vector ***x*** must be distinct and must be sorted into ascending order; that is, $x_1 < x_2 < ... < x_n$. Otherwise, results are unpredictable. For details on how to do this, see "ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)" on page 1110.

Function

Interpolation is performed at specified abscissas, t_i for $i = 1, m$, in vector ***t***, using the cubic spline passing through the data points:

(x_j, y_j) for $j = 1, n$

where:

$x_1 < x_2 < x_3 < ... < x_n$
x_j are elements of vector ***x***.
y_j are elements of vector ***y***.

The value of the cubic spline at each t_i is returned in s_i for $i = 1, m$. See references [“23” on page 1254] and [“65” on page 1256]. The coefficients of the spline, c_{jk} for $j = 1, n$ and $k = 1, 4$, are returned in matrix ***C***. These coefficients can then be reused on subsequent calls to this subroutine, using the same data points (x_j, y_j), but with new values of t_i. The cubic spline values returned in ***s*** are computed using the coefficients as follows:

$s_i = c_{j1} + c_{j2} (x_j-t_i) + c_{j3} (x_j-t_i)^2 + c_{j4} (x_j-t_i)^3$ for $i = 1, m$

where:

$j = 1$ for $t_i \le x_1$
$j = k$ for $x_1 < t_i \le x_n$, such that $x_{k-1} < t_i \le x_k$
$j = n$ for $x_n < t_i$

The values specified for *m* and *init* indicate which combination of functions are performed by this subroutine:

- If $m = 0$ and $init > 3$, no computation is performed.
- If $m = 0$ and $init \le 3$, only the coefficients are computed, and no interpolation is performed.
- If $m \ne 0$ and $init > 3$, the coefficients are not computed, and the interpolation is performed.
- If $m \ne 0$ and $init \le 3$, the coefficients are computed, and the interpolation is performed.

In addition, if $n = 0$, no computation is performed.

The values specified for *n* and *init* determine the type of spline function:

- If $n = 1$, the constructed spline is a constant function.
- If $n = 2$ and $init = 0$, the constructed spline is a line through the points.
- If $n = 2$ and $init = 1$, the constructed spline is a cubic function through the points whose derivative at x_1 is c_{11}.
- If $n = 2$ and $init = 2$, the constructed spline is a cubic function through the points whose derivative at x_n is c_{21}.
- If $n = 2$ and $init = 3$, the constructed spline is a cubic function through the points whose derivative at x_1 is c_{11} and at x_n is c_{21}.

Error conditions

Computational Errors
None

Input-Argument Errors

1. $n < 0$
2. $m < 0$

Examples

Example 1

This example computes the spline coefficients through a set of data points with no derivative value specified. It also evaluates the spline at the abscissas specified in T. On output, `INIT` and `C` are updated with new values.

Call Statement and Input:

```
             X   Y   C   N  INIT  T   S   M
             |   |   |   |   |    |   |   |
CALL SCSINT( X , Y , C , 6 , 0  , T , S , 4 )
```

X = (1.000, 2.000, 3.000, 4.000, 5.000, 6.000)
Y = (0.000, 1.000, 2.000, 1.100, 0.000, -1.000)
C =(not relevant)
T = (-1.000, 2.500, 4.000, 7.000)

Output:

```
        |  0.000  -0.868   0.000  -0.132 |
        |  1.000  -1.264   0.396  -0.132 |
C    =  |  2.000  -0.076  -1.585   0.660 |
        |  1.100   1.267   0.243  -0.609 |
        |  0.000   1.010   0.014   0.076 |
        | -1.000   0.995   0.000   0.005 |
```

```
INIT     =  4
S        =  (-2.792, 1.649, 1.100, -2.000)
```

Example 2

This example computes the spline coefficients through a set of data points with a derivative value specified at the right endpoint. It also evaluates the spline at the abscissas specified in T. On output, `INIT` and `C` are updated with new values.

Call Statement and Input:

```
             X   Y   C   N  INIT  T   S   M
             |   |   |   |   |    |   |   |
CALL SCSINT( X , Y , C , 6 , 2  , T , S , 4 )

X        =  (1.000, 2.000, 3.000, 4.000, 5.000, 6.000)
Y        =  (0.000, 1.000, 2.000, 1.100, 0.000, -1.000)
```

```
        |  .   .   .   . |
        | 0.1  .   .   . |
C    =  |  .   .   .   . |
        |  .   .   .   . |
        |  .   .   .   . |
        |  .   .   .   . |


T        =  (-1.000, 2.500, 4.000, 7.000)
```

Output:

```
        |  0.000  -0.865   0.000  -0.135 |
        |  1.000  -1.270   0.405  -0.135 |
C    =  |  2.000  -0.054  -1.621   0.675 |
        |  1.100   1.188   0.379  -0.667 |
        |  0.000   1.303  -0.494   0.291 |
        | -1.000   0.100   1.897  -0.797 |
INIT     =  4
S        =  (-2.810, 1.652, 1.100, 1.794)
```

Example 3

This example computes the spline coefficients through a set of data points with a derivative value specified at both endpoints. It does not evaluate the spline at any points. On output, INIT and C are updated with new values. Because arrays are not needed for arguments *t* and *s*, the value 0 is specified in their place.

Call Statement and Input:

```
             X   Y   C   N  INIT  T   S   M
             |   |   |   |   |    |   |   |
CALL SCSINT( X , Y , C , 6 , 3  , 0 , 0 , 0 )

X        =  (1.000, 2.000, 3.000, 4.000, 5.000, 6.000)
Y        =  (0.000, 1.000, 2.000, 1.100, 0.000, -1.000)
```

```
         | -1.0  .   .   . |
         |  0.1  .   .   . |
C    =   |   .   .   .   . |
         |   .   .   .   . |
         |   .   .   .   . |
         |   .   .   .   . |
```

Output:

```
         |  0.000   1.000   3.230   1.230 |
         |  1.000  -1.770  -0.460   1.230 |
C    =   |  2.000   0.079  -1.389   0.310 |
         |  1.100   1.152   0.316  -0.568 |
         |  0.000   1.312  -0.476   0.264 |
         | -1.000  -0.100   1.888  -0.788 |


INIT     =  4
```

Example 4

This example evaluates the spline at a set of points, using the coefficients obtained in Example 3.

Call Statement and Input:

```
             X   Y   C   N  INIT  T   S   M
             |   |   |   |   |    |   |   |
CALL SCSINT( X , Y , C , 6 , 4  , T , S , 4 )
```

X = (1.000, 2.000, 3.000, 4.000, 5.000, 6.000)
Y = (0.000, 1.000, 2.000, 1.100, 0.000, -1.000)
C =(same as output C in Example 3)
T = (-1.000, 2.500, 4.000, 7.000)

Output:

C =(same as output C in Example 3)
S = (24.762, 1.731, 1.100, 1.776)
INIT = 4

SCSIN2 and DCSIN2 (Two-Dimensional Cubic Spline Interpolation)

Purpose

These subroutines compute the interpolation values at a specified set of points, using data defined on a rectangular mesh in the x-y plane.

Table 249: Data Types.	
Data Types	
x, ***y***, ***Z***, ***t***, ***u***, *aux*, ***S***	**Subroutine**
Short-precision real	SCSIN2
Long-precision real	DCSIN2

Syntax

Language	Syntax
Fortran	CALL SCSIN2 | DCSIN2 (*x*, *y*, *z*, *n1*, *n2*, *ldz*, *t*, *u*, *m1*, *m2*, *s*, *lds*, *aux*, *naux*)
C and C++	scsin2 | dcsin2 (*x*, *y*, *z*, *n1*, *n2*, *ldz*, *t*, *u*, *m1*, *m2*, *s*, *lds*, *aux*, *naux*);

On Entry

x

is the vector ***x*** of length *n1*, containing the x-coordinates of the data points that define the spline. The elements of ***x*** must be distinct and sorted into ascending order.

Specified as: a one-dimensional array of (at least) length *n1*, containing numbers of the data type indicated in Table 249 on page 1138.

y

is the vector ***y*** of length *n2*, containing the y-coordinates of the data points that define the spline. The elements of ***y*** must be distinct and sorted into ascending order.

Specified as: a one-dimensional array of (at least) length *n2*, containing numbers of the data type indicated in Table 249 on page 1138.

z

is the matrix ***Z***, containing the data at (x_i, y_j) for $i = 1, n1$ and $j = 1, n2$ that defines the spline.

Specified as: an *ldz* by (at least) *n2* array, containing numbers of the data type indicated in Table 249 on page 1138.

n1

is the number of elements in vector ***x*** and the number of rows in matrix ***Z***—that is, the number of x-coordinates at which the spline is defined.

Specified as: an integer; $n1 \geq 0$.

n2

is the number of elements in vector ***y*** and the number of columns in matrix ***Z***—that is, the number of y-coordinates at which the spline is defined.

Specified as: an integer; $n2 \geq 0$.

ldz

is the leading dimension of the array specified for *z*.

Specified as: an integer; $ldz > 0$ and $ldz \geq n1$.

t

is the vector ***t*** of length *m1*, containing the x-coordinates at which the spline is evaluated.

Specified as: a one-dimensional array of (at least) length *m1*, containing numbers of the data type indicated in Table 249 on page 1138.

u

is the vector ***u*** of length *m2*, containing the y-coordinates at which the spline is evaluated.

Specified as: a one-dimensional array of (at least) length *m2*, containing numbers of the data type indicated in Table 249 on page 1138.

m1

is the number of elements in vector ***t***—that is, the number of x-coordinates at which the spline interpolation is evaluated. Specified as: an integer; $m1 \geq 0$.

m2

is the number of elements in vector ***u***—that is, the number of y-coordinates at which the spline interpolation is evaluated. Specified as: an integer; $m2 \geq 0$.

s

See On Return.

lds

is the leading dimension of the array specified for *s*.

Specified as: an integer; $lds > 0$ and $lds \geq m1$.

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 247 on page 1130. On output, the contents are overwritten.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SCSIN2 and DCSIN2 dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise:

For SCSIN2

For 32-bit integer arguments

$naux \geq (10)(\max(n1, n2)) + (n2 + 1)(m1) + 2(m2)$

For 64-bit integer arguments

$naux \geq (10)(\max(n1, n2)) + (n2 + 2)(m1) + 3(m2)$

For DCSIN2

$naux \geq (10)(\max(n1, n2)) + (n2 + 1)(m1) + 2(m2)$

On Return

s

is the matrix ***S*** with elements s_{kh} that contain the interpolation values at (t_k, u_h) for $k = 1, m1$ and $h = 1, m2$. Returned as: an *lds* by (at least) *m2* array, containing numbers of the data type indicated in Table 249 on page 1138.

Notes

1. The cyclic reduction method used to solve the equations in this subroutine can generate underflows on well-scaled problems. This does not affect accuracy, but it may decrease performance. For this reason, you may want to disable underflow before calling this subroutine.
2. Vectors ***x***, ***y***, ***t***, and ***u***, matrix ***Z***, and the *aux* work area must have no common elements with matrix ***S***; otherwise, results are unpredictable.
3. The elements within vectors ***x*** and ***y*** must be distinct. In addition, the elements in the vectors must be sorted into ascending order; that is, $x_1 < x_2 < ... < x_{n1}$ and $y_1 < y_2 < ... < y_{n2}$. Otherwise, results are

unpredictable. For details on how to do this, see "ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)" on page 1110.

4. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

Interpolation is performed at a specified set of points:

(t_k, u_h) for $k = 1, m1$ and $h = 1, m2$

by fitting bicubic spline functions with natural boundary conditions, using the following set of data, defined on a rectangular grid, (x_i, y_j) for $i = 1, n1$ and $j = 1, n2$:

z_{ij} for $i = 1, n1$ and $j = 1, n2$

where t_k, u_h, x_i, y_j, and z_{ij} are elements of vectors ***t***, ***u***, ***x***, and ***y*** and matrix ***Z***, respectively. In vectors ***x*** and ***y***, elements are assumed to be sorted into ascending order.

The interpolation involves two steps:

1. For each *j* from 1 to *n2*, the single variable cubic spline:

   $$s_{y_j}(x)$$

   with natural boundary conditions, is constructed using the data points:

   (x_i, z_{ij}) for $i = 1, n1$

   The following interpolation values are then computed:

   $$s_{y_j}(t_k) \qquad \text{for } k = 1, m1$$

2. For each *k* from 1 to *m1*, the single variable cubic spline:

   $$s_{t_k}(y),$$

   with natural boundary conditions, is constructed using the data points:

   $$\left(y_j, s_{y_j}(t_k)\right) \qquad \text{for } j = 1, n2$$

   The following interpolation values are then computed:

   $$s_{kh} = s_{t_k}(u_h) \qquad \text{for } l = 1, m2$$

See references ["65" on page 1256] and ["73" on page 1257]. Because natural boundary conditions (zero second derivatives at the end of the ranges) are used for the splines, unless the underlying function has these properties, interpolated values near the boundaries may be less satisfactory than elsewhere. If *n1*, *n2*, *m1*, or *m2* is 0, no computation is performed.

Error conditions

Resource Errors

Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors

None

Input-Argument Errors

1. *n1* < 0 or *n1* > *ldz*
2. *n2* < 0
3. *m1* < 0 or *m1* > *lds*
4. *m2* < 0
5. *ldz* < 0
6. *lds* < 0
7. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value specified in the syntax for this argument. Return code 1 is returned if error 2015 is recoverable.

Examples

Example

This example computes the interpolated values at a specified set of points, given by T and U, from a set of data points defined on a rectangular mesh in the x-y plane, using X, Y, and Z.

Call Statement and Input:

```
             X   Y   Z   N1  N2 LDZ  T   U   M1  M2  S  LDS  AUX  NAUX
             |   |   |   |   |   |   |   |   |   |   |   |    |    |
CALL SCSIN2( X , Y , Z , 6 , 5 , 6 , T , U , 4 , 3 , S , 4 , AUX , 90 )


X        =  (0.0, 0.2, 0.3, 0.4, 0.5, 0.7)
Y        =  (0.0, 0.2, 0.3, 0.4, 0.6)
```

```
         | 0.000  0.008  0.027  0.064  0.216 |
         | 0.008  0.016  0.035  0.072  0.224 |
Z    =   | 0.027  0.035  0.054  0.091  0.243 |
         | 0.064  0.072  0.091  0.128  0.280 |
         | 0.125  0.133  0.152  0.189  0.341 |
         | 0.343  0.351  0.370  0.407  0.559 |
```

```
T        =  (0.10, 0.15, 0.25, 0.35)
U        =  (0.05, 0.25, 0.45)
```

Output:

```
         | 0.001  0.017  0.095 |
S    =   | 0.003  0.019  0.097 |
         | 0.016  0.031  0.110 |
         | 0.043  0.059  0.137 |
```

Chapter 15. Numerical quadrature subroutines

The numerical quadrature subroutines are described.

Overview of the Numerical Quadrature Subroutines

The numerical quadrature subroutines provide Gaussian quadrature methods for integrating a tabulated function and a user-supplied function over a finite, semi-infinite, or infinite region of integration.

Table 250: List of Numerical Quadrature Subroutines.

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SPTNQ	DPTNQ	"SPTNQ and DPTNQ (Numerical Quadrature Performed on a Set of Points)" on page 1145
SGLNQ†	DGLNQ†	"SGLNQ and DGLNQ (Numerical Quadrature Performed on a Function Using Gauss-Legendre Quadrature)" on page 1147
SGLNQ2†	DGLNQ2†	"SGLNQ2 and DGLNQ2 (Numerical Quadrature Performed on a Function Over a Rectangle Using Two-Dimensional Gauss-Legendre Quadrature)" on page 1149
SGLGQ†	DGLGQ†	"SGLGQ and DGLGQ (Numerical Quadrature Performed on a Function Using Gauss-Laguerre Quadrature)" on page 1155
SGRAQ†	DGRAQ†	"SGRAQ and DGRAQ (Numerical Quadrature Performed on a Function Using Gauss-Rational Quadrature) " on page 1158
SGHMQ†	DGHMQ†	"SGHMQ and DGHMQ (Numerical Quadrature Performed on a Function Using Gauss-Hermite Quadrature)" on page 1161

† This subprogram is invoked as a function in a Fortran program.

Use Considerations

This contains some key points about using the numerical quadrature subroutines.

Choosing the Method

The theoretical aspects of choosing the method to use for integration can be found in the references ["34" on page 1254], ["74" on page 1257], and ["113" on page 1259].

Performance and Accuracy Considerations

1. There are *n* function evaluations for a method of order *n*. Because function evaluations are expensive in terms of computing time, you should weigh the considerations for computing time and accuracy in choosing a value for *n*.
2. To achieve optimal performance in the _GLNQ2 subroutines, specify the first variable integrated to be the variable having more points. This allows both the subroutine and the function evaluation to achieve optimal performance. Details on how to do this are given in "Notes" on page 1151.

3. There are some ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 61.

Programming Considerations for the SUBF Subroutine

This describes how to design and code the *subf* subroutine for use by the numerical quadrature subrutines.

Designing SUBF

For the Gaussian quadrature subroutines, you must supply a separate subroutine that is callable by ESSL. You specify the name of the subroutine in the *subf* argument. This subroutine name is selected by you. You should design the *subf* subroutine so it receives, as input, a tabulated set of points at which the integrand is evaluated, and it returns, as output, the values of the integrand evaluated at these points.

Depending on the numerical quadrature subroutine that you use, the *subf* subroutine is defined in one of the two following ways:

- For _GLNQ, _GLGQ, _GRAQ, and _GHMQ, you define the *subf* subroutine with three arguments: *t*, *y*, and *n*, where:

  t
  is an input array, referred to as T, of tabulated Gaussian quadrature abscissas, containing *n* real numbers, t_i, where t_i is automatically provided by the ESSL subroutine and is determined by *n* and the Gaussian quadrature method chosen.

  y
  is an output array, referred to as Y, containing *n* real numbers, where for the integrand, the following is true: $y_i = f(t_i)$ for $i = 1, n$.

  n
  is a positive integer indicating the number of elements in T and Y.

- For _GLNQ2, you define the *subf* subroutine with six arguments: *s*, *n1*, *t*, *n2*, *z*, and *ldz*, where:

  s
  is an input array, referred to as S, of tabulated Gaussian quadrature abscissas, containing *n1* real numbers, s_i, where s_i is automatically provided by the ESSL subroutine and is determined by *n1* and the Gaussian quadrature method.

  n1
  is a positive integer indicating the number of elements in S and the number of rows to be used in array ***Z***.

  t
  is an input array, referred to as T, of tabulated Gaussian quadrature abscissas, containing *n2* real numbers, t_i, where t_i is automatically provided by the ESSL subroutine and is determined by *n2* and the Gaussian quadrature method.

  n2
  is a positive integer indicating the number of elements in T and the number of columns to be used in array ***Z***.

  z
  is an *ldz* by (at least) *n2* output array, referred to as Z, of real numbers, where for the integrand, the following is true: $z_{ij} = f(s_i, t_j)$ for $i = 1, n1$ and $j = 1, n2$.

  ldz
  is a positive integer indicating the size of the leading dimension of the array ***Z***.

Coding and Setting Up SUBF in Your Program

Examples of coding a *subf* subroutine in Fortran are provided for each subroutine here. Examples of coding a *subf* subroutine in C, and C++ are provided in Example 1.

Depending on the programming language you use for your program that calls the numerical quadrature subroutines, you have a choice of one or more languages that you can use for writing *subf*. These rules and other language-related coding rules for setting up *subf* in your program are described in the following:

- "Setting Up a User-Supplied Subroutine for ESSL in Fortran" on page 125
- "Setting Up a User-Supplied Subroutine for ESSL in C" on page 144
- "Setting Up a User-Supplied Subroutine for ESSL in C++" on page 157

Numerical Quadrature Subroutines

This contains the numerical quadrature subroutine descriptions.

SPTNQ and DPTNQ (Numerical Quadrature Performed on a Set of Points)

Purpose

These subroutines approximate the integral of a real valued function specified in tabular form, (x_i, y_i) for i = 1, n. For more than four points, an error estimate is returned along with the resulting value.

Table 251: Data Types.

Data Types	
x, ***y***, *xyint*, *eest*	**Subroutine**
Short-precision real	SPTNQ
Long-precision real	DPTNQ

Syntax

Language	**Syntax**
Fortran	CALL SPTNQ | DPTNQ (*x*, *y*, *n*, *xyint*, *eest*)
C and C++	sptnq | dptnq (*x*, *y*, *n*, *xyint*, *eest*);

On Entry

x

is the vector ***x*** of length *n*, containing the abscissas of the data points to be integrated. The elements of ***x*** must be distinct and sorted into ascending or descending order.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 251 on page 1145.

y

is the vector ***y*** of length *n*, containing the ordinates of the data points to be integrated.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 251 on page 1145.

n

is the number of elements in vectors ***x*** and ***y***—that is, the number of data points. The value of *n* determines the algorithm used by this subroutine. For details, see "Function" on page 1146.

Specified as: an integer; $n \geq 2$.

xyint

See On Return.

eest
See On Return.

On Return

xyint
is the approximation *xyint* of the integral. Returned as: a number of the data type indicated in Table 251 on page 1145.

eest
has the following meaning, where:

If $n < 5$, it is undefined and is set to 0.

If $n \geq 5$, it is an estimate, *eest*, of the error in the integral, where *xyint*+*eest* tends to give a better approximation to the integral than *xyint*. For details, see references [“34” on page 1254] and [“74” on page 1257].

Returned as: a number of the data type indicated in Table 251 on page 1145.

Notes

1. In your C program, arguments *xyint* and *eest* must be passed by reference.
2. The elements of vector ***x*** must be distinct—that is, $x_i \neq x_j$ for $i \neq j$,—and they must be sorted into ascending or descending order; otherwise, results are unpredictable. For how to do this, see “ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)” on page 1110.

Function

The integral is approximated for a real valued function specified in tabular form, (x_i, y_i) for $i = 1, n$, where x_i are distinct and sorted into ascending or descending order, and $n \geq 2$. If $y_i = f(x_i)$ for $i = 1, n$, then on output, *xyint* is an approximation to the integral of the following form:

$$\int_{x_1}^{x_n} f(x)dx$$

The algorithm used by this subroutine is based on the number of data points used in the computation, where:

- If $n = 2$, the trapezoid rule is used to do the integration.
- If $n = 3$, the parabola through the three points is integrated.
- If $n \geq 4$, the method of Gill and Miller is used to do the integration.

For $n \geq 5$, an estimate of the error *eest* is returned. For the method of Gill and Miller, it is shown that adding the estimate of the error *eest* to the result *xyint* often gives a better approximation to the integral than the result *xyint* by itself. For $n < 5$, an estimate of the error is not returned. In this case, a value of 0 is returned for *eest*. See references [“74” on page 1257] and [“34” on page 1254].

Error conditions

Computational Errors
None

Input-Argument Errors

$n < 2$

Examples

Example 1

This example shows the result of an integration, where the abscissas in X are sorted into ascending order.

Call Statement and Input:

```
             X   Y   N    XYINT   EEST
             |   |   |      |      |
CALL SPTNQ( X , Y , 10 , XYINT , EEST )

X        =  (0.0, 0.4, 1.0, 1.5, 2.1, 2.6, 3.0, 3.4, 3.9, 4.3)
Y        =  (1.0, 2.0, 3.0, 4.0, 5.0, 4.5, 4.0, 3.0, 3.5, 3.3)
```

Output:

```
XYINT    =  15.137
EEST     =  -0.003
```

Example 2

This example shows the result of an integration, where the abscissas in X are sorted into descending order.

Call Statement and Input:

```
             X   Y   N    XYINT   EEST
             |   |   |      |      |
CALL SPTNQ( X , Y , 10 , XYINT , EEST )

X        =  (4.3, 3.9, 3.4, 3.0, 2.6, 2.1, 1.5, 1.0, 0.4, 0.0)
Y        =  (3.3, 3.5, 3.0, 4.0, 4.5, 5.0, 4.0, 3.0, 2.0, 1.0)
```

Output:

```
XYINT    =  -15.137
EEST     =  0.003
```

SGLNQ and DGLNQ (Numerical Quadrature Performed on a Function Using Gauss-Legendre Quadrature)

Purpose

These functions approximate the integral of a real valued function over a finite interval, using the Gauss-Legendre Quadrature method of specified order.

Table 252: Data Types.

Data Types	
a, *b*, **Result**	**Subroutine**
Short-precision real	SGLNQ
Long-precision real	DGLNQ

Syntax

Language	Syntax
Fortran	SGLNQ | DGLNQ (*subf*, *a*, *b*, *n*)
C and C++	sglnq | dglnq (*subf*, *a*, *b*, *n*);

On Entry

subf

is the user-supplied subroutine that evaluates the integrand function. The subroutine should be defined with three arguments: *t*, *y*, and *n*. For details, see "Programming Considerations for the SUBF Subroutine" on page 1144.

Specified as: *subf* must be declared as an external subroutine in you application program. It can be whatever name you choose.

a

is the lower limit of integration, *a*.

Specified as: a number of the data type indicated in Table 252 on page 1147.

b

is the upper limit of integration, *b*.

Specified as: a number of the data type indicated in Table 252 on page 1147.

n

is the order of the quadrature method to be used.

Specified as: an integer; *n* = 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, 20, 24, 32, 40, 48, 64, 96, 128, or 256.

On Return

Function value

is the approximation of the integral. Returned as: a number of the data type indicated in Table 252 on page 1147.

Notes

1. Declare the DGLNQ function in your program as returning a long-precision real number. Declare the SGLNQ, if necessary, as returning a short-precision real number.
2. The subroutine specified for *subf* must be declared as external in your program. Also, data types used by *subf* must agree with the data types specified by this ESSL subroutine. The variable *x*, described under "Function" on page 1148, and the argument *n* correspond to the *subf* arguments *t* and *n*, respectively. For details on how to set up the subroutine, see "Programming Considerations for the SUBF Subroutine" on page 1144.

Function

The integral is approximated for a real valued function over a finite interval, using the Gauss-Legendre Quadrature method of specified order. The region of integration is from *a* to *b*. The method of order *n* is theoretically exact for integrals of the following form, where *f* is a polynomial of degree less than 2*n*:

$$\int_a^b f(x)dx$$

The method of order *n* is a good approximation when your integrand is closely approximated by a function of the form *f*(*x*), where *f* is a polynomial of degree less than 2*n*. See references ["34" on page 1254] and ["113" on page 1259]. The result is returned as the function value.

Error conditions

Computational Errors

None

Input-Argument Errors

n is not an allowable value, as listed in the syntax for this argument.

Examples

Example

This example shows how to compute the integral of the function *f* given by:

$f(x) = x^2+e^x$

over the interval (0.0, 2.0), using the Gauss-Legendre method with 10 points:

$$\int_{0.0}^{2.0}\left(x^2+e^x\right)dx$$

The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in Fortran as follows:

```
   SUBROUTINE FUN1 (T,Y,N)
   INTEGER*4 N
   REAL*4 T(*),Y(*)
   DO 1 I=1,N
1     Y(I)=T(I)**2+EXP(T(I))
   RETURN
   END
```

Program Statements and Input:

```
EXTERNAL FUN1
        .
        .
        .
              SUBF    A     B    N
               |      |     |    |
XINT = SGLNQ( FUN1 , 0.0 , 2.0 , 10 )
        .
        .
        .
```

FUN1 = (see above)

Output:

```
XINT     =  9.056
```

SGLNQ2 and DGLNQ2 (Numerical Quadrature Performed on a Function Over a Rectangle Using Two-Dimensional Gauss-Legendre Quadrature)

Purpose

These functions approximate the integral of a real valued function of two variables over a rectangular region, using the Gauss-Legendre Quadrature method of specified order in each variable.

Table 253: Data Types.

Data Types	
a, *b*, *c*, *d*, ***Z***, **Result**	**Subroutine**
Short-precision real	SGLNQ2
Long-precision real	DGLNQ2

Syntax

Language	Syntax
Fortran	SGLNQ2 | DGLNQ2 (*subf*, *a*, *b*, *n1*, *c*, *d*, *n2*, *z*, *ldz*)
C and C++	sglnq2 | dglnq2 (*subf*, *a*, *b*, *n1*, *c*, *d*, *n2*, *z*, *ldz*);

On Entry

subf

is the user-supplied subroutine that evaluates the integrand function. The subroutine should be defined with six arguments: *s*, *n1*, *t*, *n2*, *z*, and *ldz*. For details, see "Programming Considerations for the SUBF Subroutine" on page 1144.

Specified as: *subf* must be declared as an external subroutine in your application program. It can be whatever name you choose.

a

is the lower limit of integration, *a*, for the first variable integrated.

Specified as: a number of the data type indicated in Table 253 on page 1149.

b

is the upper limit of integration, *b*, for the first variable integrated.

Specified as: a number of the data type indicated in Table 253 on page 1149.

n1

is the order of the quadrature method to be used for the first variable integrated.

Specified as: an integer; *n1* = 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, 20, 24, 32, 40, 48, 64, 96, 128, or 256.

c

is the lower limit of integration, *c*, for the second variable integrated.

Specified as: a number of the data type indicated in Table 253 on page 1149.

d

is the upper limit of integration, *d*, for the second variable integrated.

Specified as: a number of the data type indicated in Table 253 on page 1149.

n2

is the order of the quadrature method to be used for the second variable integrated.

Specified as: an integer; *n2* = 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, 20, 24, 32, 40, 48, 64, 96, 128, or 256.

z

is the matrix ***Z***, containing the *n1* rows and *n2* columns of data used to evaluate the integrand function. (The output values from the *subf* subroutine are placed in ***Z***.) Specified as: an *ldz* by (at least) *n2* array, containing numbers of the data type indicated in Table 253 on page 1149.

ldz

is the size of the leading dimension of the array specified for *z*.

Specified as: an integer; *ldz* > 0 and *ldz* ≥ *n1*.

On Return

Function value

is the approximation of the integral. Returned as: a number of the data type indicated in Table 253 on page 1149.

Notes

1. Declare the DGLNQ2 function in your program as returning a long-precision real number. Declare the SGLNQ2 function, if necessary, as returning a short-precision real number.
2. The subroutine specified for *subf* must be declared as external in your program. Also, data types used by *subf* must agree with the data types specified by this ESSL subroutine. For details on how to set up the subroutine, see "Programming Considerations for the SUBF Subroutine" on page 1144.

Function

The integral:

$$\int_c^d \int_a^b f(s,t)\, ds\, dt$$

is approximated for a real valued function of two variables *s* and *t*, over a rectangular region, using the Gauss-Legendre Quadrature method of specified order in each variable. The region of integration is:

(*a*, *b*) for *s*
(*c*, *d*) for *t*

The method gives a good approximation when your integrand is closely approximated by a function of the form *f*(*s*, *t*), where *f* is a polynomial of degree less than 2(*n1*) for *s* and 2(*n2*) for *t*. See the function description for "SGLNQ and DGLNQ (Numerical Quadrature Performed on a Function Using Gauss-Legendre Quadrature)" on page 1147 and references ["34" on page 1254] and ["113" on page 1259]. The result is returned as the function value.

Special Usage

To achieve optimal performance in this subroutine and in the functional evaluation, specify the first variable integrated in this subroutine as the variable having more points. The first variable integrated is the variable in the inner integral. For example, in the following integration, *x* is the first variable integrated:

$$\int_{u1}^{u2} \int_{r1}^{r2} f(x,y)\, dx\, dy$$

This is the suggested order of integration if the *x* variable has more points than the *y* variable. On the other hand, if the *y* variable has more points, you make *y* the first variable integrated.

Because the order of integration does not matter to the resulting approximation, you may be able to reverse the order that *x* and *y* are integrated and get better performance. This can be expressed as:

$$\int_{u1}^{u2} \int_{r1}^{r2} f(x,y)\, dx\, dy = \int_{r1}^{r2} \int_{u1}^{u2} f(x,y)\, dy\, dx$$

Results are mathematically equivalent. However, because the algorithm is computed in a different way, results may not be bitwise identical.

Table 254 on page 1152 shows how to assign your variables to the _GLNQ2 and *subf* arguments for the *x*-*y* integration shown on the left and for the *y*-*x* integration shown on the right. For examples of how to do each of these, see Example 1 and Example 2.

Table 254: How to Assign Your Variables for x-y Integration Versus y-x Integration.

_GLNQ2 and SUBF Arguments	**Variables for *x-y* Integration**	**Variables for *y-x* Integration**
For _GLNQ2:		
a	*r1*	*u1*
b	*r2*	*u2*
n1	(order for *x*)	(order for *y*)
c	*u1*	*r1*
d	*u2*	*r2*
n2	(order for *y*)	(order for *x*)
For *subf*:		
s	*x*	*y*
t	*y*	*x*
n1	(order for *x*)	(order for *y*)
n2	(order for *y*)	(order for *x*)

Error conditions

Computational Errors

None

Input-Argument Errors

1. *ldz* ≤ 0
2. *n1* > *ldz*
3. *n1* or *n2* is not an allowable value, as listed in the syntax for this argument.

Examples

Example 1

This example shows how to compute the integral of the function *f* given by:

$f(x, y) = e^x \sin y$

over the intervals (0.0, 2.0) for the first variable *x* and (-2.0, -1.0) for the second variable *y*, using the Gauss-Legendre method with 10 points in the *x* variable and 5 points in the *y* variable:

$$\int_{-2.0}^{-1.0}\int_{0.0}^{2.0}\left(e^{x} \sin y\right)dx\, dy$$

Because the variable *x* has more points, it is the first variable integrated. This allows the SGLNQ2 subroutine and the FUN1 evaluation to achieve optimal performance. Therefore, the *x* and *y* variables correspond to S and T in the FUN1 subroutine. Also, the *x* and *y* variables correspond to the A, B, N1 and C, D, N2 sets of arguments, respectively, for SGLNQ2.

Using Fortran for SUBF:

The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in Fortran as follows:

```
      SUBROUTINE FUN1 (S,N1,T,N2,Z,LDZ)
      INTEGER*4 N1,N2,LDZ
      REAL*4 S(*),T(*),Z(LDZ,*)
      DO 1 J=1,N2
      DO 2 I=1,N1
 2       Z(I,J)=EXP(S(I))*SIN(T(J))
 1       CONTINUE
```

```
    RETURN
    END
```

Note: The computation for this user-supplied subroutine FUN1 can also be performed by using the following statements in place of the above DO loops, using T1 and T2 as temporary storage areas:

```
    .
    .
    .
    DO 1 I=1,N1
1      T1(I)=EXP(S(I))
    DO 2 J=1,N2
2      T2(J)=SIN(T(J))
    DO 3 J=1,N2
    DO 4 I=1,N1
4      Z(I,J)=T1(I)*T2(J)
3      CONTINUE
    .
    .
    .
```

When coding your application, this is the preferred technique. It reduces the number of evaluations performed and, therefore, provides better performance.

Using C for SUBF:

The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in C as follows:

```
    void fun1(s, n1, t, n2, z, ldz)
    float *s, *t, *z;
    int *n1, *n2, *ldz;
    {
         int i, j;
         for(j = 0; j < *n2; ++j, z += *ldz)
             {
              for(i = 0; i < *n1; ++i)
              z[i] = exp(s[i]) * sin(t[j]);
             }
    }
```

Using C++ for SUBF:

The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in C++ as follows:

```
    void fun1(float *s, int *n1, float *t, int *n2, float *z, int *ldz)
    {
         int i, j;
         for(j = 0; j < *n2; ++j, z += *ldz)
             {
             for(i = 0; i < *n1; ++i)
              z[i] = exp(s[i]) * sin(t[j]);
             }
    }
```

Program Statements and Input:

```
EXTERNAL FUN1
        .
        .
        .
                SUBF    A     B    N1     C      D    N2  Z   LDZ
                 |      |     |    |      |      |    |   |    |
XYINT = SGLNQ2( FUN1 , 0.0 , 2.0 , 10 , -2.0 , -1.0 , 5 , Z , 10 )
        .
        .
        .
```

FUN1 = (see above)
Z = (not relevant)

Output:

```
XYINT    =  -6.1108
```

Example 2

This example shows how to reverse the order of integration of the variables *x* and *y*. It computes the integral of the function *f* given by:

$f(x, y) = \cos x \sin y$

over the intervals (0.0, 1.0) for the variable *x* and (0.0, 20.0) for the variable *y*, using the Gauss-Legendre method with 5 points in the *x* variable and 48 points in the *y* variable. Because the order of integration does not matter to the approximation:

$$\int_{0.0}^{20.0}\int_{0.0}^{1.0}(\cos\ x\ \sin\ y)\,dx\,dy = \int_{0.0}^{1.0}\int_{0.0}^{20.0}(\cos\ x\ \sin\ y)\,dy\,dx$$

the variable *y*, having more points, is the first variable integrated (performing the integration shown on the right.) This allows the SGLNQ2 subroutine and the FUN1 evaluation to achieve optimal performance. Therefore, the *x* and *y* variables correspond to T and S in the FUN2 subroutine. Also, the *x* and *y* variables correspond to the C, D, N2 and A, B, N1 sets of arguments, respectively, for SGLNQ2.

The user-supplied subroutine FUN2, which evaluates the integrand function, is coded in Fortran as follows:

```
   SUBROUTINE FUN2 (S,N1,T,N2,Z,LDZ)
   INTEGER*4 N1,N2,LDZ
   REAL*4 S(*),T(*),Z(LDZ,*)
   DO 1 J=1,N2
   DO 2 I=1,N1
2     Z(I,J)=COS(T(J))*SIN(S(I))
1     CONTINUE
   RETURN
   END
```

Note: The same coding principles for achieving good performance that are noted in Example 1 also apply to this user-supplied subroutine FUN2.

Program Statements and Input:

```
EXTERNAL FUN2.
        .
        .
        .
                SUBF    A      B    N1    C     D    N2   Z  LDZ
                 |      |      |    |     |     |    |    |   |
YXINT = SGLNQ2( FUN2 , 0.0 , 20.0 , 48 , 0.0 , 1.0 , 5  , Z , 48 )
        .
        .
        .
```

FUN2 = (see above)
Z = (not relevant)

Output:

```
YXINT    =  0.4981
```

SGLGQ and DGLGQ (Numerical Quadrature Performed on a Function Using Gauss-Laguerre Quadrature)

Purpose

These functions approximate the integral of a real valued function over a semi-infinite interval, using the Gauss-Laguerre Quadrature method of specified order.

Table 255: Data Types.

Data Types	
a, *b*, **Result**	**Subroutine**
Short-precision real	SGLGQ
Long-precision real	DGLGQ

Syntax

Language	Syntax
Fortran	SGLGQ | DGLGQ (*subf*, *a*, *b*, *n*)
C and C++	sglgq | dglgq (*subf*, *a*, *b*, *n*);

On Entry

subf

is the user-supplied subroutine that evaluates the integrand function. The subroutine should be defined with three arguments: *t*, *y*, and *n*. For details, see "Programming Considerations for the SUBF Subroutine" on page 1144.

Specified as: *subf* must be declared as an external subroutine in your application program. It can be whatever name you choose.

a

has the following meaning, where:

If $b > 0$, it is the lower limit of integration.

If $b < 0$, it is the upper limit of integration.

Specified as: a number of the data type indicated in Table 255 on page 1155.

b

is the scaling constant *b* for the exponential.

Specified as: a number of the data type indicated in Table 255 on page 1155; $b > 0$ or $b < 0$.

n

is the order of the quadrature method to be used.

Specified as: an integer; n = 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, 20, 24, 32, 40, 48, or 64.

On Return

Function value

is the approximation of the integral. Returned as: a number of the data type indicated in Table 255 on page 1155.

Notes

1. Declare the DGLGQ function in your program as returning a long-precision real number. Declare the SGLGQ function, if necessary, as returning a short-precision real number.

2. The subroutine specified for *subf* must be declared as external in your program. Also, data types used by *subf* must agree with the data types specified by this ESSL subroutine. The variable *x*, described under "Function" on page 1156, and the argument *n* correspond to the *subf* arguments *t* and *n*, respectively. For details on how to set up the subroutine, see "Programming Considerations for the SUBF Subroutine" on page 1144.

Function

The integral is approximated for a real valued function over a semi-infinite interval, using the Gauss-Laguerre Quadrature method of specified order. The region of integration is:

(a, ∞) if $b > 0$
$(-\infty, a)$ if $b < 0$

The method of order *n* is theoretically exact for integrals of the following form, where *f* is a polynomial of degree less than 2*n*:

$$\int_a^\infty f(x)e^{-bx}dx \qquad \text{if } b > 0$$

$$\int_{-\infty}^a f(x)e^{-bx}dx \qquad \text{if } b < 0$$

The method of order *n* is a good approximation when your integrand is closely approximated by a function of the form $f(x)e^{-bx}$, where *f* is a polynomial of degree less than 2*n*. See references ["34" on page 1254] and ["113" on page 1259]. The result is returned as the function value.

Error conditions

Computational Errors
None

Input-Argument Errors

1. $b = 0$
2. *n* is not an allowable value, as listed in the syntax for this argument.

Examples

Example 1

This example shows how to compute the integral of the function *f* given by:

$f(x) = \sin(3.0x)e^{-1.5x}$

over the interval (-2.0, ∞), using the Gauss-Laguerre method with 20 points:

$$\int_{-2.0}^\infty \left(\sin(3.0x)e^{-1.5x}\right)dx$$

The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in Fortran as follows:

```
      SUBROUTINE FUN1 (T,Y,N)
      INTEGER*4 N
      REAL*4 T(*),Y(*)
      DO 1 I=1,N
```

```
1     Y(I)=SIN(3.0*T(I))*EXP(-1.5*T(I))
   RETURN
   END
```

Program Statements and Input:

```
EXTERNAL FUN1
        .
        .
        .
              SUBF     A     B    N
               |       |     |    |
XINT = SGLGQ( FUN1 , -2.0 , 1.5 , 20 )
        .
        .
        .
```

FUN1 = (see above)

Output:

```
XINT     =  5.891
```

Example 2

This example shows how to compute the integral of the function *f* given by:

$f(x) = \sin(3.0x)e^{1.5x}$

over the interval (-∞, -2.0), using the Gauss-Laguerre method with 20 points:

$$\int_{-\infty}^{-2.0} \sin(3.0x)e^{1.5x}\,dx$$

The user-supplied subroutine FUN2, which evaluates the integrand function, is coded in Fortran as follows:

```
   SUBROUTINE FUN2 (T,Y,N)
   INTEGER*4 N
   REAL*4 T(*),Y(*),TEMP
   DO 1 I=1,N
1     Y(I)=SIN(3.0*T(I))*EXP(1.5*T(I))
   RETURN
   END
```

Program Statements and Input:

```
EXTERNAL FUN2
        .
        .
        .
              SUBF     A      B    N
               |       |      |    |
XINT = SGLGQ( FUN2 , -2.0 , -1.5 , 20 )
        .
        .
        .
FUN2     =  (see above)
```

Output:

```
XINT     =  -0.011
```

SGRAQ and DGRAQ (Numerical Quadrature Performed on a Function Using Gauss-Rational Quadrature)

Purpose

These functions approximate the integral of a real valued function over a semi-infinite interval, using the Gaussian-Rational quadrature method of specified order.

Table 256: Data Types.

Data Types	
a, *b*, **Result**	**Subroutine**
Short-precision real	SGRAQ
Long-precision real	DGRAQ

Syntax

Language	Syntax
Fortran	SGRAQ | DGRAQ (*subf*, *a*, *b*, *n*)
C and C++	sgraq | dgraq (*subf*, *a*, *b*, *n*);

On Entry

subf

is the user-supplied subroutine that evaluates the integrand function. The subroutine should be defined with three arguments: *t*, *y*, and *n*. For details, see "Programming Considerations for the SUBF Subroutine" on page 1144.

Specified as: *subf* must be declared as an external subroutine in your application program. It can be whatever name you choose.

a

has the following meaning, where:

If $a+b > 0$, it is the lower limit of integration.

If $a+b < 0$, it is the upper limit of integration.

Specified as: a number of the data type indicated in Table 256 on page 1158.

b

is the centering constant *b* for the integrand.

Specified as: a number of the data type indicated in Table 256 on page 1158.

n

is the order of the quadrature method to be used.

Specified as: an integer; n = 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, 20, 24, 32, 40, 48, 64, 96, 128, or 256.

On Return

Function value

is the approximation of the integral. Returned as: a number of the data type indicated in Table 256 on page 1158.

Notes

1. Declare the DGRAQ function in your program as returning a long-precision real number. Declare the SGRAQ function, if necessary, as returning a short-precision real number.

2. The subroutine specified for *subf* must be declared as external in your program. Also, data types used by *subf* must agree with the data types specified by this ESSL subroutine. The variable *x*, described under "Function" on page 1159, and the argument *n* correspond to the *subf* arguments *t* and *n*, respectively. For details on how to set up the subroutine, see "Programming Considerations for the SUBF Subroutine" on page 1144.

Function

The integral is approximated for a real valued function over a semi-infinite interval, using the Gauss-Rational quadrature method of specified order. The region of integration is:

(a, ∞) if $a+b > 0$
$(-\infty, a)$ if $a+b < 0$

The method of order *n* is theoretically exact for integrals of the following form, where *f* is a polynomial of degree less than 2*n*:

$$\int_a^\infty f\left(\frac{1}{x+b}\right)\frac{1}{(x+b)^2}\,dx \qquad \text{if } a+b>0$$

$$\int_{-\infty}^a f\left(\frac{1}{x+b}\right)\frac{1}{(x+b)^2}\,dx \qquad \text{if } a+b<0$$

The method of order *n* is a good approximation when your integrand is closely approximated by a function of the following form, where *f* is a polynomial of degree less than 2*n*:

$$f\left(\frac{1}{x+b}\right)\frac{1}{(x+b)^2}$$

See references ["34" on page 1254] and ["113" on page 1259]. The result is returned as the function value to a Fortran, C, or C++ program.

Error conditions

Computational Errors
None

Input-Argument Errors

1. $a+b = 0$
2. *n* is not an allowable value, as listed in the syntax for this argument.

Examples

Example 1

This example shows how to compute the integral of the function *f* given by:

$f(x) = (e^{1.0}/x) / x^2$

over the interval $(-\infty, -2.0)$, using the Gauss-Rational method with 10 points:

$$\int_{-\infty}^{-2.0} \left(\frac{e^{1.0/x}}{x^2} \right) dx$$

The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in Fortran as follows:

```
   SUBROUTINE FUN1 (T,Y,N)
   INTEGER*4 N
   REAL*4 T(*),Y(*),TEMP
   DO 1 I=1,N
        TEMP=1.0/T(I)
1       Y(I)=EXP(TEMP)*TEMP**2
   RETURN
   END
```

Program Statements and Input:

```
EXTERNAL FUN1
        .
        .
        .
              SUBF     A     B    N
               |       |     |    |
XINT = SGRAQ( FUN1 , -2.0 , 0.0 , 10 )
        .
        .
        .
```

FUN1 = (see above)

Output:

```
XINT     =  0.393
```

Example 2

This example shows how to compute the integral of the function *f* given by:

$f(x) = (x\text{-}3.0)^{-2} + 10(x\text{-}3.0)^{-11}$

over the interval (4.0, ∞), using the Gauss-Rational method with 6 points:

$$\int_{4.0}^{\infty} \left((x-3.0)^{-2} + 10(x-3.0)^{-11} \right) dx$$

The user-supplied subroutine FUN2, which evaluates the integrand function, is coded in Fortran as follows:

```
   SUBROUTINE FUN2 (T,Y,N)
   INTEGER*4 N
   REAL*4 T(*),Y(*),TEMP
   DO 1 I=1,N
        TEMP=1.0/(T(I)-3.0)
1       Y(I)=TEMP**2+10.0*TEMP**11
   RETURN
   END
```

Program Statements and Input:

```
EXTERNAL FUN2
        .
        .
        .
              SUBF    A      B    N
               |      |      |    |
XINT = SGRAQ( FUN2 , 4.0 , -3.0 , 6 )
        .
        .
        .
FUN2     =  (see above)
```

Output:

```
XINT     =  2.00
```

SGHMQ and DGHMQ (Numerical Quadrature Performed on a Function Using Gauss-Hermite Quadrature)

Purpose

These functions approximate the integral of a real valued function over the entire real line, using the Gauss-Hermite Quadrature method of specified order.

Table 257: Data Types.

Data Types	
a, *b*, **Result**	**Subroutine**
Short-precision real	SGHMQ
Long-precision real	DGHMQ

Syntax

Language	**Syntax**
Fortran	SGHMQ | DGHMQ (*subf*, *a*, *b*, *n*)
C and C++	sghmq | dghmq (*subf*, *a*, *b*, *n*);

On Entry

subf

is the user-supplied subroutine that evaluates the integrand function. The subroutine should be defined with three arguments: *t*, *y*, and *n*. For details, see "Programming Considerations for the SUBF Subroutine" on page 1144.

Specified as: *subf* must be declared as an external subroutine in your application program. It can be whatever name you choose.

a

is the centering constant *a* for the exponential.

Specified as: a number of the data type indicated in Table 257 on page 1161.

b

is the scaling constant *b* for the exponential.

Specified as: a number of the data type indicated in Table 257 on page 1161; *b* > 0.

n

is the order of the quadrature method to be used.

Specified as: an integer; *n* = 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, 20, 24, 32, 40, 48, 64, or 96.

On Return

Function value

is the approximation of the integral. Returned as: a number of the data type indicated in Table 257 on page 1161.

Notes

1. Declare the DGHMQ function in your program as returning a long-precision real number. Declare the SGHMQ function, if necessary, as returning a short-precision real number.
2. The subroutine specified for *subf* must be declared as external in your program. Also, data types used by *subf* must agree with the data types specified by this ESSL subroutine. The variable *x*, described under "Function" on page 1162, and the argument *n* correspond to the *subf* arguments *t* and *n*, respectively. For details on how to set up the subroutine, see "Programming Considerations for the SUBF Subroutine" on page 1144.

Function

The integral is approximated for a real valued function over the entire real line, using the Gauss-Hermite Quadrature method of specified order. The region of integration is from -∞ to ∞. The method of order *n* is theoretically exact for integrals of the following form, where *f* is a polynomial of degree less than 2*n*:

$$\int_{-\infty}^{\infty} f(x) e^{-b(x-a)^2} dx$$

The method of order *n* is a good approximation when your integrand is closely approximated by a function of the following form, where *f* is a polynomial of degree less than 2*n*:

$$f(x) e^{-b(x-a)^2}$$

See references ["34" on page 1254] and ["113" on page 1259]. The result is returned as the function value to a Fortran, C, or C++ program.

Error conditions

Computational Errors
None

Input-Argument Errors

1. $b \le 0$
2. *n* is not an allowable value, as listed in the syntax for this argument.

Examples

Example

This example shows how to compute the integral of the function *f* given by:

$$f(x) = x^2 e^{-2(x+5.0)^2}$$

over the interval (-∞, ∞), using the Gauss-Hermite method with 4 points:

$$\int_{-\infty}^{\infty} \left(x^2 e^{-2(x+5.0)^2} \right) dx$$

The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in Fortran as follows:

```
   SUBROUTINE FUN1 (T,Y,N)
   INTEGER*4 N
   REAL*4 T(*),Y(*)
   DO 1 I=1,N
1     Y(I)=T(I)**2*EXP(-2.0*(T(I)+5.0)**2)
   RETURN
   END
```

Program Statements and Input:

```
EXTERNAL FUN1
        .
        .
        .
              SUBF     A     B    N
               |       |     |    |
XINT = SGHMQ( FUN1 , -5.0 , 2.0 , 4 )
        .
        .
        .
```

FUN1 = (see above)

Output:

```
XINT     =  31.646
```

Chapter 16. Random number generation subroutines

The random number generation subroutines are described here.

Overview of the Random Number Generation Subroutines

Random number generation subroutines generate uniformly distributed random numbers or normally distributed random numbers using one of the following algorithms:

- SIMD-oriented Mersenne Twister algorithm
- Multiplicative congruential methods
- Polar methods
- Tausworthe exclusive-or algorithm

Table 258: List of Random Number Generation Initialization Subroutines.

Subroutine	**Descriptive Name and Location**
INITRNG	"INITRNG (Initialize Random Number Generators)" on page 1166

Table 259: List of Random Number Generation Subroutines.

Short-Precision Subroutine	**Long-Precision Subroutine**	**Descriptive Name and Location**
SURNG	DURNG	"SURNG and DURNG (Generate a Vector of Uniformly Distributed Pseudo-Random Numbers)" on page 1170
SNRNG	DNRNG	"SNRNG and DNRNG (Generate a Vector of Normally Distributed Pseudo-Random numbers)" on page 1173
SURAND	DURAND	"SURAND and DURAND (Generate a Vector of Uniformly Distributed Random Numbers)" on page 1176
SNRAND	DNRAND	"SNRAND and DNRAND (Generate a Vector of Normally Distributed Random Numbers)" on page 1178
SURXOR§	DURXOR§	"SURXOR and DURXOR (Generate a Vector of Long Period Uniformly Distributed Random Numbers)" on page 1181
§ This subroutine is provided for migration from earlier releases of ESSL and is not intended for use in new programs.		

Use Considerations

If you need a very long period random number generator, you should select the following subroutines:

- SURNG rather than SURAND or SURXOR
- DURNG rather than DURAND or DURXOR
- SNRNG rather than SNRAND
- DNRNG rather than DNRAND.

Random Number Generation Subroutines

This contains the random number generation subroutine descriptions.

INITRNG (Initialize Random Number Generators)

Purpose

This subroutine initializes the selected pseudo-random number generator for use in subsequent calls to SURNG, DURNG, SNRNG or DNRNG. To generate a repeatable or non-repeatable vector of pseudo-random numbers, follow the call to INITRNG with one or more calls to SURNG, DURNG, SNRNG or DNRNG.

Syntax

Language	**Syntax**
Fortran	CALL INITRNG (*iopt*, *irepeat*, *iseed*, *liseed*, *istate*, *listate*)
C and C++	initrng (*iopt*, *irepeat*, *iseed*, *liseed*, *istate*, *listate*);

On Entry

iopt

indicates the random number generator desired for use, where:

If *iopt* = 1, a single-precision, SIMD-oriented Mersenne Twister pseudo-random number generator with a period of 2^{19937}-1 (SFMT19937) is used.

If *iopt* = 2, a long-precision, SIMD-oriented Mersenne Twister pseudo-random number generator with a period of 2^{19937}-1 (DSFMT19937) is used.

Specified as: an integer; *iopt* = 1 or 2.

irepeat

indicates whether repeatable or non-repeatable pseudo-random number sequences will be generated, where:

If *irepeat* = 0, the pseudo-random number generator uses values from *iseed* to generate repeatable pseudo-random number sequences.

If *irepeat* = 1, the pseudo-random number generator uses hardware-generated values to generate non-repeatable pseudo-random number sequences.

Specified as: an integer; *irepeat* = 0 or 1.

iseed

If *irepeat* = 0, *iseed* is an array containing the initial seed values to use in initializing the pseudo-random number generator to generate repeatable pseudo-random number sequences.

If *irepeat* = 1, *iseed* is ignored.

Specified as: a one-dimensional integer array of (at least) length max(1,*liseed*).

liseed

is the number of elements in array ISEED, where:

If *irepeat* = 0, *liseed* is determined as follows:

32-bit integer environment

If *iopt* = 1 or 2, *liseed* ≥ 624.

64-bit integer environment
If *iopt* = 1 or 2, *liseed* ≥ 312.

Note: If *irepeat* = 0 and insufficient seeds are provided, the seed values supplied in *iseed* are used and this subroutine initializes the remaining seed values on the basis of the supplied seed values.

If *irepeat* = 1, *liseed* is ignored.

Specified as: If *irepeat* = 0, an integer > 0.

istate
See "On Return".

listate
If *listate* ≠ -1, *listate* is the number of elements in the array *istate*, where *listate* depends on both the environment the subroutine is running in and the value of *iopt*, as follows:

32-bit integer environment

- If *iopt* = 1, *listate* ≥ 696.
- If *iopt* = 2, *listate* ≥ 839.

64-bit integer environment

- If *iopt* = 1, *listate* ≥ 348.
- If *iopt* = 2, *listate* ≥ 420.

If *listate* = -1, an *istate* size query is assumed. The subroutine returns the minimum required size of *istate* in the output argument *listate*.

Specified as: an integer; -1 or > 0.

On Return

istate
If *listate* > 0 on entry, *istate* contains information about the pseudo-random number generator and the initial seeds for use in subsequent calls to SURNG, DURNG, SNRNG or DNRNG.

If *listate* = -1 on entry, then *istate* is unchanged.

Returned as: a one-dimensional integer array of (at least) length max(1,*listate*)

listate
If *listate* = -1 on entry, then on return it contains the minimum required size of *istate*.

Otherwise, it remains unchanged.

Returned as: an integer.

Notes

1. In your C program, argument *listate* must be passed by reference.
2. For a 64-bit integer environment where *iopt* = 1 or *iopt* = 2, if *liseed* is larger than 2^{31}, only the first 2^{31}-1 *iseed* values are used to initialize the *istate* output value.
3. *iseed* and *istate* must have no common elements; otherwise, results are unpredictable.

Function

This subroutine initializes the selected pseudo-random number generator for use in subsequent calls to SURNG, DURNG, SNRNG or DNRNG. To generate a repeatable or non-repeatable vector of pseudo-random numbers, follow the call to INITRNG with one or more calls to SURNG, DURNG, SNRNG or DNRNG.

The following pseudo-random number generators are supported:

1. SIMD-oriented fast Mersenne Twister pseudo-random number generator SFMT19937 (see [“98” on page 1258]) with a period length equal to 2^{19937}-1 of the produced sequence.

2. Double precision floating point SFMT19937 pseudo-random number generator DSFMT19937 ((see [“99” on page 1258]) with a period length equal to 2^{19937}-1 of the produced sequence.

See references [“97” on page 1258], [“98” on page 1258] and [“99” on page 1258].

Error conditions

Computational Errors
None

Input-Argument Errors

1. *iopt* ≠ 1 or 2
2. *irepeat* = 0 and *liseed* < 1
3. In a 32-bit integer environment:
   - *iopt* = 1 and *listate* ≠ -1 and *listate* < 696.
   - *iopt* = 2 and *listate* ≠ -1 and *listate* < 839.
4. In a 64-bit integer environment:
   - *iopt* = 1 and *listate* ≠ -1 and *listate* < 348.
   - *iopt* = 2 and *listate* ≠ -1 and *listate* < 420.

Examples

Example 1

This example shows a call to INITRNG to find the optimal size of the *istate* array needed by the SFMT19937 pseudo-random number generator.

Call Statement and Input:

```
             IOPT    IREPEAT   ISEED   LISEED   ISTATE LISTATE
               |        |        |        |        |      |
CALL INITRNG(  1 ,   IREPEAT , ISEED , LISEED , ISTATE , -1 )
```

Output:

```
LISTATE  =  696 (in a 32-bit integer environment)

LISTATE  =  348 (in a 64-bit integer environment)
```

Example 2

This example shows a call to INITRNG to find the optimal size of the *istate* array needed by the DSFMT19937 pseudo-random number generator.

Call Statement and Input:

```
             IOPT    IREPEAT   ISEED   LISEED   ISTATE LISTATE
               |        |        |        |        |      |
CALL INITRNG(  2 ,   IREPEAT , ISEED , LISEED , ISTATE , -1 )
```

Output:

```
LISTATE  =  839 (in a 32-bit integer environment)

LISTATE  =  420 (in a 64-bit integer environment)
```

Example 3

This example shows how to initialize the *istate* array with the seed values for the SFMT19937 pseudo-random number generator to generate repeatable random sequences in a subsequent call to SURNG or SNRNG.

Call Statement and Input:

```
             IOPT    IREPEAT   ISEED   LISEED   ISTATE  LISTATE
               |        |        |        |        |       |
CALL INITRNG(  1 ,      0 ,    ISEED , LISEED , ISTATE , 1000 )
```

In a 32-bit integer environment:

```
ISEED(1) = 0
ISEED(2) = 1234
ISEED(3) = 0
ISEED(4) = 5678
LISEED   = 4
```

In a 64-bit integer environment:

```
ISEED(1) = 1234
ISEED(2) = 5678
LISEED   = 2
```

Output:

istate contains an array of seeds for the SFMT19937 pseudo-random number generator to generate repeatable random sequences, which can be used in a subsequent call to SURNG or SNRNG.

Example 4

This example shows how to initialize the *istate* array with the seed values for the DSFMT19937 pseudo-random number generator to generate repeatable random sequences in a subsequent call to DURNG or DNRNG.

Call Statement and Input:

```
             IOPT    IREPEAT   ISEED   LISEED   ISTATE  LISTATE
               |        |        |        |        |       |
CALL INITRNG(  2 ,      0 ,    ISEED , LISEED , ISTATE , 1000 )
```

In a 32-bit integer environment:

```
ISEED(1) = 0
ISEED(2) = 1234
ISEED(3) = 0
ISEED(4) = 5678
LISEED   = 4
```

In a 64-bit integer environment:

```
ISEED(1) = 1234
ISEED(2) = 5678
LISEED   = 2
```

Output:

istate contains an array of seeds for the DSFMT19937 pseudo-random number generator, which can be used in a subsequent call to DURNG or DNRNG.

Example 5

This example shows how to initialize the *istate* array with the seed values for the DSFMT19937 pseudo-random number generator to generate non-repeatable random sequences in a subsequent call to DURNG or DNRNG.

Call Statement and Input:

```
               IOPT    IREPEAT   ISEED   LISEED   ISTATE  LISTATE
                |         |        |        |        |        |
CALL INITRNG(  2 ,       1 ,     ISEED , LISEED , ISTATE , 1000 )
```

Output:

istate contains an array of seeds for the DSFMT19937 pseudo-random number generator, which can be used in a subsequent call to DURNG or DNRNG.

SURNG and DURNG (Generate a Vector of Uniformly Distributed Pseudo-Random Numbers)

Purpose

These subroutines generate a repeatable or non-repeatable vector ***x*** of uniform pseudo-random numbers uniformly distributed over the interval [*a*, *b*].

For the initial call to these subroutines, you must initialize the pseudo-random number generator with a preceding call to INITRNG.

Table 260: Data Types.

Data Types	
x, *a*, *b*	**Subroutine**
Short-precision real	SURNG
Long-precision real	DURNG

Syntax

Language	**Syntax**
Fortran	CALL SURNG | DURNG (*n*, *a*, *b*, *x*, *istate*, *listate*)
C and C++	surng | durng (*n*, *a*, *b*, *x*, *istate*, *listate*);

On Entry

n

is the number of pseudo-random numbers to be generated.

Specified as: an integer; *n* ≥ 0.

a

is the left boundary of the interval [*a*, *b*].

Specified as: a number of the data type indicated in Table 260 on page 1170.

b

is the right boundary of the interval [*a*, *b*].

Specified as: a number of the data type indicated in Table 260 on page 1170.

x

See "On Return".

istate

is an array containing information about the current state of the pseudo-random number generator.

Note: If you are invoking this subroutine for the first time, *istate* must be the output of a preceding call to subroutine INITRNG, as follows:

- For SURNG, INITRNG must have been invoked with *iopt* = 1
- For DURNG, INITRNG must have been invoked with *iopt* = 2

Specified as: a one-dimensional integer array of (at least) length *listate*.

listate

is the number of elements in the array *istate* and depends on both the environment the subroutine is running in and the value of *iopt* specified on the previous call to INITRNG, as follows:

32-bit integer environment

- If INITRNG was called with *iopt* = 1, *listate* ≥ 696.
- If INITRNG was called with *iopt* = 2, *listate* ≥ 839.

64-bit integer environment

- If INITRNG was called with *iopt* = 1, *listate* ≥ 348.
- If INITRNG was called with *iopt* = 2, *listate* ≥ 420.

Specified as: an integer; *listate* > 0.

On Return

x

is a vector of length *n*, containing the uniformly distributed pseudo-random numbers.

Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 260 on page 1170.

istate

is an array of length *listate* containing updated information about the state of the pseudo-random number generator for use in subsequent calls to this subroutine.

Returned as: a one-dimensional integer array of (at least) length *listate*.

Notes

x and *istate* must have no common elements; otherwise, results are unpredictable.

Function

These subroutines generate a repeatable or non-repeatable vector ***x*** of uniform pseudo-random numbers uniformly distributed over the interval [*a*, *b*].

For the initial call to these subroutines, you must initialize the pseudo-random number generator with a preceding call to INITRNG.

The computation involves the following steps:

1. Retrieve the information for the initialized pseudo-random number generator.
2. Generate the uniformly distributed sequence with the selected pseudo-random number generator.
3. Scale the sequence of pseudo-random numbers.

See references [“97” on page 1258], [“98” on page 1258] and [“99” on page 1258].

If *n* is 0, no computation is performed.

Error conditions

Computational Errors

None

Input-Argument Errors

1. $n < 0$
2. $a \geq b$
3. *istate* is not initialized (by a preceding call to INITRNG)
4. *istate* is initialized (by a preceding call to INITRNG):
   - With *iopt* = 2 for a call to SURNG
   - With *iopt* = 1 for a call to DURNG
5. *listate* is less than the minimum required value.

Examples

Example 1

This example shows a call to SURNG to generate 10 uniformly distributed short-precision pseudo-random numbers between 0.0 and 1.0.

Call Statement and Input:

```
             N     A     B    X   ISTATE  LISTATE
             |     |     |    |      |       |
CALL SURNG( 10 ,  0.0 , 1.0 , X , ISTATE , 1000 )
```

ISTATE = (same as ouput ISTATE in Example 3)

Note: For the initial call to SURNG, you must initialize the pseudo-random number generator with a preceding call to INITRNG (see Example 3).

Output:

```
         | 0.439785 |
         | 0.064906 |
         | 0.385660 |
         | 0.695451 |
X    =   | 0.496463 |
         | 0.154272 |
         | 0.002247 |
         | 0.725402 |
         | 0.037238 |
         | 0.892588 |
```

ISTATE = contains the updated state of the pseudo-random number generator.

Example 2

This example shows a call to DURNG to generate 10 uniformly distributed long-precision pseudo-random numbers between 0.0 and 1.0.

Call Statement and Input:

```
             N     A     B    X   ISTATE  LISTATE
             |     |     |    |      |       |
CALL DURNG( 10 ,  0.0 , 1.0 , X , ISTATE , 1000 )
```

ISTATE = (same as ouput ISTATE in Example 4)

Note: For the initial call to DURNG, you must initialize the pseudo-random number generator with a preceding call to INITRNG (see Example 4).

Output:

```
        | 0.948207 |
        | 0.388311 |
        | 0.758121 |
        | 0.430842 |
X    =  | 0.261129 |
        | 0.693552 |
        | 0.113275 |
        | 0.607048 |
        | 0.192948 |
        | 0.669879 |
```

ISTATE = contains the updated state of the pseudo-random number generator.

SNRNG and DNRNG (Generate a Vector of Normally Distributed Pseudo-Random numbers)

Purpose

These subroutines generate a repeatable or non-repeatable vector ***x*** of normally distributed pseudo-random numbers normally distributed with a mean of *rmean* and a standard deviation of *sigma*, using the BoxMuller2 method.

For the initial call to these subroutines, you must initialize the pseudo-random number generator with a preceding call to INITRNG.

Table 261: Data Types.

Data Types	
x, *rmean*, *sigma*	**Subroutine**
Short-precision real	SNRNG
Long-precision real	DNRNG

Syntax

Language	**Syntax**
Fortran	CALL SNRNG | DNRNG (*n*, *rmean*, *sigma*, *x*, *istate*, *listate*)
C and C++	SNRNG | DNRNG (*n*, *rmean*, *sigma*, *x*, *istate*, *listate*);

On Entry

n

is the number of pseudo-random numbers to be generated.

Specified as: an integer; *n* must be an even number and *n* ≥ 0.

rmean

is the mean value of the distribution.

Specified as: a number of the data type indicated in Table 261 on page 1173.

sigma
is the standard deviation value of the distribution.

Specified as: a number of the data type indicated in Table 261 on page 1173.

x
See "On Return".

istate
is an array containing information about the current state of the pseudo-random number generator.

Note: If you are invoking this subroutine for the first time, *istate* must be the non-zero output of a preceding call to subroutine INITRNG, as follows:

- For SNRNG, INITRNG must have been invoked with *iopt* = 1
- For DNRNG, INITRNG must have been invoked with *iopt* = 2

Specified as: a one-dimensional integer array of (at least) length *listate*.

listate
is the number of elements in the array *istate* and depends on both the environment the subroutine is running in and the value of *iopt* specified on the previous call to INITRNG, as follows:

32-bit pointer environment

- If INITRNG was called with *iopt* = 1, *listate* ≥ 696.
- If INITRNG was called with *iopt* = 2, *listate* ≥ 839.

64-bit pointer environment

- If INITRNG was called with *iopt* = 1, *listate* ≥ 348.
- If INITRNG was called with *iopt* = 2, *listate* ≥ 420.

Specified as: an integer; *listate* > 0.

On Return

x
is a vector of length *n*, containing the normally distributed pseudo-random numbers.

Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 261 on page 1173.

istate
is an array of length *listate* containing updated information about the state of the pseudo-random number generator for use in subsequent calls to this subroutine.

Returned as: a one-dimensional integer array of (at least) length *listate*.

Notes

x and *istate* must have no common elements; otherwise, results are unpredictable.

Function

These subroutines generate a repeatable or non-repeatable vector ***x*** of normally distributed pseudo-random numbers normally distributed with a mean of *rmean* and a standard deviation of *sigma*, using the BoxMuller2 method.

For the initial call to these subroutines, you must initialize the pseudo-random number generator with a preceding call to INITRNG.

The computation involves the following steps:

1. Retrieve the information for the initialized pseudo-random number generator.
2. Generate the uniformly distributed sequence with the selected pseudo-random number generator.
3. Generate the normally distributed sequence using the BoxMuller2 method.

See references [“14” on page 1254], [“97” on page 1258], [“98” on page 1258] and [“99” on page 1258].

If *n* is 0, no computation is performed.

Error conditions

Computational Errors

None

Input-Argument Errors

1. *n* < 0 or *n* is an odd number
2. *sigma* ≤ 0
3. *istate* is not initialized (by a preceding call to INITRNG)
4. *istate* is initialized:
   - With *iopt* = 2 for a call to SNRNG
   - With *iopt* = 1 for a call to DNRNG
5. *listate* is less than the minimum required value.

Examples

Example 1

This example shows a call to SNRNG to generate 10 normally distributed short-precision pseudo-random numbers with a mean value of 0.0 and a standard deviation of 1.0.

Call Statement and Input:

```
              N   RMEAN SIGMA  X   ISTATE  LISTATE
              |     |     |    |      |       |
CALL SNRNG( 10 ,  0.0 , 1.0 , X , ISTATE , 1000 )
```

ISTATE = (same as ouput ISTATE in Example 3)

Note: For the initial call to SNRNG, you must initialize the pseudo-random number generator with a preceding call to INITRNG (see Example 3).

Output:

```
         ┌           ┐
         | -0.426951 |
         |  0.988221 |
         |  0.929709 |
         | -0.331744 |
X    =   | -0.965826 |
         |  0.662854 |
         |  0.066279 |
         | -0.010326 |
         |  0.172133 |
         |  0.215101 |
                    ┘
```

ISTATE = contains the updated state of the pseudo-random number generator.

Example 2

This example shows a call to DNRNG to generate 10 normally distributed long-precision pseudo-random numbers with a mean value of 0.0 and a standard deviation of 1.0.

Call Statement and Input:

```
              N   RMEAN SIGMA  X   ISTATE  LISTATE
              |     |     |    |      |       |
CALL DNRNG( 10 ,  0.0 , 1.0 , X , ISTATE , 1000 )
```

ISTATE = (same as ouput ISTATE in Example 4)

Note: For the initial call to DNRNG, you must initialize the pseudo-random number generator with a preceding call to INITRNG (see Example 4).

Output:

```
         | -0.426951 |
         | -1.570857 |
         | -1.858332 |
         | -0.709286 |
         | -1.528250 |
X    =   |  0.729566 |
         | -0.270181 |
         |  0.305498 |
         | -0.383550 |
         |  0.573548 |
         | -0.315877 |
         └
```

ISTATE = contains the updated state of the pseudo-random number generator.

SURAND and DURAND (Generate a Vector of Uniformly Distributed Random Numbers)

Purpose

These subroutines generate vector ***x*** of uniform (0,1) pseudo-random numbers, using the multiplicative congruential method with a user-specified seed.

Table 262: Data Types.

Data Types		
x	*seed*	**Subroutine**
Short-precision real	Long-precision real	SURAND
Long-precision real	Long-precision real	DURAND

Note: If you need a very long period random number generator, use SURXOR and DURXOR instead of these subroutines.

Syntax

Language	**Syntax**
Fortran	CALL SURAND | DURAND (*seed*, *n*, *x*)
C and C++	surand | durand (*seed*, *n*, *x*);

On Entry

seed

is the initial value used to generate the random numbers.

Specified as: a number of the data type indicated in Table 262 on page 1176. It should be a whole number; that is, the fraction part should be 0. (If you specify a mixed number, it is truncated.) Its value must be 1.0 ≤ *seed* < (2147483647.0 = 2^{31}-1).

Note: *seed* is always a long-precision real number, even in SURAND.

n

is the number of random numbers to be generated.

Specified as: an integer; $n \geq 0$.

x

See On Return.

On Return

seed

is the new seed that is to be used to generate additional random numbers in subsequent invocations of SURAND or DURAND. Returned as: a number of the data type indicated in Table 262 on page 1176. It is a whole number whose value is $1.0 \leq seed < (2147483647.0 = 2^{31}-1)$.

x

is a vector of length *n*, containing the uniform pseudo-random numbers with values between 0 and 1. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 262 on page 1176.

Notes

In your C program, argument *seed* must be passed by reference.

Function

The uniform (0,1) pseudo-random numbers are generated as follows, using the multiplicative congruential method:

$s_i = (a(s_{i-1})) \bmod(m) = (a^i s_0) \bmod(m)$
$x_i = s_i/m \quad$ for $i = 1, 2, ..., n$

where:

s_i is a random sequence.
x_i is a random number.
s_0 is the initial seed provided by the caller.
$a = 7^5 = 16807.0$
$m = 2^{31}-1 = 2147483647.0$
n is the number of random numbers to be generated.

See references [“92” on page 1258] and [“96” on page 1258]. If *n* is 0, no computation is performed, and the initial seed is unchanged.

Error conditions

Computational Errors

None

Input-Argument Errors

1. $n < 0$
2. $seed < 1.0$ or $seed \geq 2147483647.0$

Examples

Example 1

This example shows a call to SURAND to generate 10 random numbers.

Call Statement and Input:

```
                SEED     N   X
                  |      |   |
CALL SURAND( SEED , 10 , X )

SEED     =  80629.0
```

Note: It is important to note that SEED is a long-precision number, even though X contains short-precision numbers.

Output:

```
SEED     =  759150100.0

X        =  (0.6310323,
             0.7603202,
             0.7015232,
             0.5014868,
             0.4895853,
             0.4602344,
             0.1603608,
             0.1832564,
             0.9899062,
             0.3535068)
```

Example 2

This example shows a call to DURAND to generate 10 random numbers.

Call Statement and Input:

```
             SEED    N   X
               |     |   |
CALL DURAND( SEED , 10 , X )

SEED     =  80629.0
```

Output:

```
SEED     =  759150100.0

X        =  (0.6310323270182275,
             0.7603201953509451,
             0.7015232633340746,
             0.5014868557925740,
             0.4895853057920864,
             0.4602344475967038,
             0.1603607578018497,
             0.1832563756887132,
             0.9899062002030695,
             0.3535068129904134)
```

SNRAND and DNRAND (Generate a Vector of Normally Distributed Random Numbers)

Purpose

These subroutines generate vector ***x*** of normally distributed pseudo-random numbers, with a mean of 0 and a standard deviation of 1, using Polar methods with a user-specified seed.

Table 263: Data Types.

Data Types		
x, *aux*	*seed*	**Subroutine**
Short-precision real	Long-precision real	SNRAND
Long-precision real	Long-precision real	DNRAND

Syntax

Language	Syntax
Fortran	CALL SNRAND | DNRAND (*seed*, *n*, *x*, *aux*, *naux*)
C and C++	snrand | dnrand (*seed*, *n*, *x*, *aux*, *naux*);

On Entry

seed

is the initial value used to generate the random numbers.

Specified as: a number of the data type indicated in Table 263 on page 1178. It must be a whole number; that is, the fraction part must be 0. Its value must be $1.0 \leq$ *seed* $<$ ($2147483647.0 = 2^{31}-1$).

Note: *seed* is always a long-precision real number, even in SNRAND.

n

is the number of random numbers to be generated.

Specified as: an integer; *n* must be an even number and $n \geq 0$.

x

See On Return.

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size must be greater than or equal to $n/2$.

Specified as: an area of storage, containing numbers of the data type indicated in Table 263 on page 1178. They can have any value.

naux

is the size of the work area specified by *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SNRAND and DNRAND dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, $naux \geq n/2$.

On Return

seed

is the new seed that is to be used to generate additional random numbers in subsequent invocations of SNRAND or DNRAND. Returned as: a number of the data type indicated in Table 263 on page 1178. It is a whole number whose value is $1.0 \leq$ *seed* $<$ ($2147483647.0 = 2^{31}-1$).

x

is a vector of length *n*, containing the normally distributed pseudo-random numbers. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 263 on page 1178.

Notes

1. In your C program, argument *seed* must be passed by reference.
2. Vector ***x*** must have no common elements with the storage area specified for *aux*; otherwise, results are unpredictable.

3. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

The normally distributed pseudo-random numbers, with a mean of 0 and a standard deviation of 1, are generated as follows, using Polar methods with a user-specified seed. The Polar method, which this technique is based on, was developed by G. E. P. Box, M. E. Muller, and G. Marsaglia and is described in reference ["92" on page 1258].

1. Using *seed*, a vector of uniform (0,1) pseudo-random numbers, u_i for $i = 1, n$, is generated by calling SURAND or DURAND, respectively. These u_i values are then used in the subsequent steps.
2. All (y_j, z_j) for $j = 1, n/2$ are set as follows, where each (y, z) is a point in the square -1 to 1:

   $y_j = 2u_{2j-1}-1$
   $z_j = 2u_{2j}-1$
3. All p_j for $j = 1, n/2$ are set as follows, where each p measures the square of the radius of (y, z):

   $$p_j = y_j^2 + z_j^2$$

   If $p_j \geq 1$, then p_j is discarded, and steps 1 through 3 are repeated until $p_j < 1$.
4. All x_i for $i = 1, n$ are set as follows to produce the normally distributed random numbers:

   $x_{2j-1} = y_j\ ((-2 \ln p_j) / p_j)^{0.5}$
   $x_{2j} = z_j\ ((-2 \ln p_j) / p_j)^{0.5}$
   for $j = 1, n/2$

If *n* is 0, no computation is performed, and the initial seed is unchanged.

Error conditions

Resource Errors

Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors

None

Input-Argument Errors

1. $n < 0$ or *n* is an odd number
2. *seed* < 1.0 or *seed* ≥ 2147483647.0
3. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows a call to SNRAND to generate 10 random numbers.

Call Statement and Input:

```
             SEED   N    X   AUX  NAUX
               |    |    |    |    |
CALL SNRAND( SEED , 10 , X , AUX , 5  )

SEED     =  80629.0
```

Note: It is important to note that SEED is a long-precision number, even though X contains short-precision numbers.

Output:

```
SEED     =  48669425.0

X        =  (0.660649538,
             1.312503695,
             1.906438112,
             0.014065863,
            -0.800935328,
            -3.058144093,
            -0.397426069,
            -0.370634943,
            -0.064151444,
            -0.275887042)
```

Example 2

This example shows a call to DNRAND to generate 10 random numbers.

Call Statement and Input:

```
             SEED   N    X   AUX  NAUX
               |    |    |    |    |
CALL DNRAND( SEED , 10 , X , AUX , 5  )

SEED     =  80629.0
```

Output:

```
SEED     =  48669425.0

X        =  (0.6606495655963802,
             1.3125037758861060,
             1.9064381379483730,
             0.0140658628770495,
            -0.8009353314494653,
            -3.0581441239248530,
            -0.3974260845722100,
            -0.3706349643478605,
            -0.0641514443372939,
            -0.2758870630332470)
```

SURXOR and DURXOR (Generate a Vector of Long Period Uniformly Distributed Random Numbers)

Purpose

These subroutines generate a vector ***x*** of uniform [0,1) pseudo-random numbers, using the Tausworthe exclusive-or algorithm.

Table 264: Data Types.

Data Types		
x, ***vseed***	***iseed***	**Subroutine**
Short-precision real	Integer	SURXOR
Long-precision real	Integer	DURXOR

Syntax

Language	**Syntax**
Fortran	CALL SURXOR | DURXOR (*iseed*, *n*, *x*, *vseed*)
C and C++	surxor | durxor (*iseed*, *n*, *x*, *vseed*);

On Entry

iseed

has the following meaning, where:

If *iseed* ≠ 0, *iseed* is the initial value used to generate the random numbers. You specify *iseed* ≠ 0 when you call this subroutine for the first time or when you changed *vseed* between calls to this subroutine.

If *iseed* = 0, *vseed* is used to generate the random numbers, where *vseed* was initialized by an earlier call to this subroutine. ESSL assumes you have not changed *vseed* between calls to this subroutine, when you specify *iseed* = 0.

Specified as: an integer, as indicated in Table 264 on page 1181.

n

is the number of random numbers to be generated.

Specified as: an integer; $n \geq 0$.

x

See On Return.

vseed

is the work area used by this subroutine and has the following meaning, where:

If *iseed* ≠ 0, *vseed* is not used for input. The work area can contain anything.

If *iseed* = 0, *vseed* contains the seed vector generated by a preceding call to this subroutine. *vseed* is used in this computation to generate the new random numbers. It should not be changed between calls to this subroutine.

Specified as: a one-dimensional array of (at least) length 10000, containing numbers of the data type indicated in Table 264 on page 1181.

On Return

iseed

is set to 0 for subsequent calls to SURXOR or DURXOR. Returned as: an integer, as indicated in Table 264 on page 1181.

x

is a vector of length *n*, containing the uniform pseudo-random numbers with the following values: $0 \leq x < 1$. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 264 on page 1181.

vseed

is the work area used by these subroutines, containing the new seed that is to be used in subsequent calls to this subroutine. Returned as: a one-dimensional array of (at least) length 10000, containing numbers of the data type indicated in Table 264 on page 1181.

Notes

1. You can generate the same vector *x* of random numbers by starting over and specifying your original nonzero *iseed* value.
2. Multiple calls to these subroutines with mixed sizes generate the same sequence of numbers as a single call the total length, assuming you specify the same initial *iseed* in both cases. For example, you can generate the same vector *x* of random numbers by calling this subroutine twice and specifying *n* = 10 or by calling this subroutine once and specifying *n* = 20. You need to specify the same *iseed* in the initial call in both cases, and *iseed* = 0 in the second call with *n* = 10.
3. Vector ***x*** must have no common elements with the storage area specified for *vseed*; otherwise, results are unpredictable.
4. In your C program, argument *iseed* must be passed by reference.

Function

The pseudo-random numbers uniformly distributed in the interval [0,1) are generated using the Tausworthe exclusive-or algorithm. This is based on a linear-feedback shift-register sequence. The very long period of the generator, $2^{1279}-1$, makes it useful in modern statistical simulations where the shorter period of other generators could be exhausted during a single run. If you need a large number of random numbers, you can use these subroutines, because with this generator you do not request more than a small percentage of the entire period of the generator.

This generator is based on two feedback positions to generate a new binary digit:

$$z_k = z_{(k-p)} \oplus z_{(k-q)}$$

where:

$p > q$
$k = 1, 2, \ldots$
z is a bit vector.
and where:

$\oplus$ is the bitwise exclusive-or operation.

For details, see references [“64” on page 1256], [“90” on page 1258], and [“115” on page 1259]. The values of *p* and *q* are selected according to the criteria stated in reference [“122” on page 1259].

The algorithm initializes a seed vector of length *p*, starting with *iseed*. The seed vector is stored in *vseed* for use in subsequent calls to this subroutine with *iseed* = 0.

If *n* is 0, no computation is performed, and the initial seed is unchanged.

Special Usage

For some specialized applications, if you need multiple sources of random numbers, you can specify different *vseed* areas, which are initialized with different seeds on multiple calls to this subroutine. You then get multiple sequences of the random number sequence provided by the generator that are sufficiently far apart for most purposes.

Error conditions

Computational Errors
None

Input-Argument Errors

1. *n* < 0
2. *iseed* = 0 and *vseed* does not contain valid data.

Examples

Example 1

This example shows a call to SURXOR to generate 10 random numbers.

Call Statement and Input:

```
             ISEED    N   X   VSEED
               |      |   |     |
CALL SURXOR( ISEED , 10 , X , VSEED )

ISEED    =  137
```

Output:

```
ISEED    =  0

X        =  (0.6440868,
             0.5105118,
             0.4878680,
             0.3209075,
             0.6624528,
             0.2499877,
             0.0056630,
             0.7329214,
             0.7486335,
             0.8050517)
```

Example 2

This example shows a call to SURXOR to generate 10 random numbers. This example specifies *iseed* = 0 and uses the *vseed* output generated from Example 1.

Call Statement and Input:

```
             ISEED    N   X   VSEED
               |      |   |     |
CALL SURXOR( ISEED , 10 , X , VSEED )

ISEED    =  0
```

Output:

```
ISEED    =  0

X        =  (0.9930249,
             0.0441873,
             0.6891295,
             0.3101060,
             0.6324178,
             0.3299408,
             0.3553145,
             0.0100013,
             0.0214620,
             0.8059390)
```

Example 3

This example shows a call to DURXOR to generate 20 random numbers. This sequence of numbers generated are like those generated in Examples 1 and 2.

Call Statement and Input:

```
             ISEED    N   X   VSEED
               |      |   |     |
CALL DURXOR( ISEED , 20 , X , VSEED )

ISEED    =  137
```

Output:

```
ISEED    =  0

X        =   (0.64408693438956721,
              0.51051182536460882,
              0.48786801310787142,
              0.32090755617007050,
              0.66245283144861666,
              0.24998782843358081,
              0.00566308101257373,
              0.73292147005172925,
              0.74863359794102236,
              0.80505169697755319,
              0.99302499462139138,
              0.04418740640269125,
              0.68912952155409579,
              0.31010611495627916,
              0.63241786342211936,
                 0.32994081459690583,
                 0.35531452631408911,
                 0.01000134413132581,
                 0.02146199494672940,
                 0.80593898487597615)
```

Chapter 17. Utilities subroutines

The utility subroutines are described here.

Overview of the Utility Subroutines

The utility subroutines perform general service functions that support ESSL, rather than mathematical computations.

Table 265: List of Utility Subroutines.

Subroutine	**Descriptive Name and Location**
EINFO	"EINFO (ESSL Error Information-Handler Subroutine)" on page 1188
ERRSAV	"ERRSAV (ESSL ERRSAV Subroutine)" on page 1190
ERRSET	"ERRSET (ESSL ERRSET Subroutine)" on page 1191
ERRSTR	"ERRSTR (ESSL ERRSTR Subroutine)" on page 1192
IVSSET§	Set the Vector Section Size (VSS) for the ESSL/370 Scalar Library
IEVOPS§	Set the Extended Vector Operations Indicator for the ESSL/370 Scalar Library
IESSL	"IESSL (Determine the Level of ESSL Installed)" on page 1193
SETGPUS	"SETGPUS (Set the Number of GPUs and Identify Which GPUs ESSL Should Use)" on page 1194
STRIDE	"STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)" on page 1195
>SLAMCH and DLAMCH	"SLAMCH and DLAMCH (Determine machine parameters for floating-point arithmetic)" on page 1210<
DSRSM	"DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)" on page 1212
DGKTRN	"DGKTRN (For a General Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode)" on page 1215
DSKTRN	"DSKTRN (For a Symmetric Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode)" on page 1219
§ This subroutine is provided for migration from earlier releases of ESSL and is not intended for use in new programs. Documentation for this subroutine is no longer provided.	

Use Considerations

This describes what you use the utility subroutines for.

Determining the Level of ESSL Installed

IESSL gets the level of ESSL and returns it to your program. The level consists of the following: version number, release number, modification number, and number of the most recently installed ESSL PTF. You can use this function to verify that you are running on or using the capabilities of the desired level.

Finding the Optimal Stride(s) for Your Fourier Transforms

STRIDE is used to determine optimal stride values for your Fourier transforms when using any of the Fourier transform subroutines, except _RCFT and _CRFT. You must invoke STRIDE for each optimal stride you want computed. Sometimes you need a separate stride for your input and output data. For the three-dimensional Fourier transforms, you need an optimal stride for both the second and third dimensions of the array. The examples provided for STRIDE explain how it is used for each of the subroutines listed above.

After obtaining the optimal strides from STRIDE, you should arrange your data using these stride values. After the data is set up, call the Fourier transform subroutine. For additional information on how to set up your data, see "Setting Up Your Data" on page 959.

Converting Sparse Matrix Storage

DSRSM is used to migrate your existing program from sparse matrices stored by rows to sparse matrices stored in compressed-matrix storage mode. This converts the matrices into a storage format that is compatible with the input requirements for some ESSL sparse matrix subroutines, such as DSMMX.

DGKTRN and DSKTRN are used to convert your sparse matrix from one skyline storage mode to another, if necessary, before calling the subroutines DGKFS/DGKFSP or DSKFS/DSKFSP, respectively.

Utility Subroutines

This contains the utility subroutine descriptions.

EINFO (ESSL Error Information-Handler Subroutine)

Purpose

This subroutine returns information to your program about the data involved in a computational error that occurred in an ESSL subroutine. This is the same information that is provided in the ESSL messages; however, it allows you to check the information in your program at run time and continue processing. You pass the computational error code of interest to this subroutine in *icode*, and it passes back one or more pieces of information in the output arguments *inf1* and, optionally, *inf2*, as defined in Table 266 on page 1188. **You should use this subroutine only for those computational errors listed in the table. It does not apply to computational errors that do not return information.**

For multithreaded application programs, if you want the error handling capabilities that this subroutine provides to be implemented on each thread created by your program, this subroutine must be called from each thread. If your application creates multiple threads, the action performed by a call to this subroutine applies to the thread that this subroutine was invoked from. For an example, see "Example of Handling Errors in a Multithreaded Application Program " on page 141.

Table 266: Computational Error Information Returned by EINFO.

Error Code	Receiver	Type of Information
2100	*inf1*	Lower range of a vector
	inf2	Upper range of a vector
2101	*inf1*	Index of the eigenvalue that failed to converge
	inf2	Number of iterations after which it failed to converge

Table 266: Computational Error Information Returned by EINFO. (continued)

Error Code	Receiver	Type of Information
2102	*inf1* *inf2*	Index of the last eigenvector that failed to converge Number of iterations after which it failed to converge
2103	*inf1*	Index of the pivot with zero value
2104	*inf1*	Index of the last pivot with nonpositive value
2105	*inf1*	Index of the pivot element near zero causing factorization to fail
2107	*inf1* *inf2*	Index of the singular value that failed to converge Number of iterations after which it failed to converge
2109	*inf1*	Iteration count when it was determined that the matrix was not definite
2114	*inf1* *inf2*	Index of the last eigenvalue that failed to converge Number of iterations after which it failed to converge
2115	*inf1*	Order of the leading minor that was discovered to have a nonpositive determinant
2117	*inf1*	Column number for which pivot value was near zero
2118	*inf1*	Row number for which pivot value was near zero
2120	*inf1*	Row number of empty row where factorization failed
2121	*inf1*	Column number of empty column where factorization failed
2126	*inf1*	Row number for which pivot value was unacceptable
2145	*inf1*	First diagonal element with zero value
2150	*inf1*	First diagonal element with zero value

Syntax

Language	Syntax
Fortran	CALL EINFO (*icode*[, *inf1*[, *inf2*]])
C and C++	einfo (*icode*, *inf1*, *inf2*);

On Entry

icode

has the following meaning, where:

If *icode* = 0, this indicates that the ESSL error option table is to be initialized. (You specify this value once in the beginning of your program before calls to ERRSET.)

If *icode* has any of the allowable error code values listed in Table 266 on page 1188, this is the computational error code of interest. (You specify one of these values whenever you want information returned about a computational error.)

Specified as: an integer; *icode* = 0 or an error code value indicated in Table 266 on page 1188.

inf1

See On Return.

inf2
See On Return.

On Return

inf1
has the following meaning, where:

If *icode* = 0, this argument is not used in the computation. In this case, *inf1* is an optional argument, except in C and C++ programs.

If *icode* ≠ 0, then *inf1* is the first information receiver, containing numerical information related to the computational error.

Returned as: an integer.

inf2
has the following meaning, where:

If *icode* = 0, this argument is not used in the computation.

If *icode* ≠ 0, then *inf2* is the second information receiver, containing numerical information related to the computational error. It should be specified when the error code provides a second piece of information, and you want the information.

In both of these cases, *inf2* is an optional argument, except in C and C++ programs. For more details, see "Notes" on page 1190.

Returned as: an integer.

Notes

1. If *icode* is not 0 and is not one of the error codes specified in Table 266 on page 1188, this subroutine returns to the caller, and no information is provided in *inf1* and *inf2*.
2. If there are two pieces of information for the error and you specify one output argument, the second piece of information is not returned to the caller.
3. If there is one piece of information for the error and you specify two output arguments, the second output argument is not set by this subroutine.
4. In C and C++ programs you must code the *inf1* and *inf2* arguments, because they are not optional arguments.
5. In Fortran programs, *inf1* and *inf2* are optional arguments. This is an exception to the rule, because other ESSL subroutines do not allow optional arguments.
6. Examples of how to use EINFO are provided in Chapter 4, "Coding programs," on page 125.

ERRSAV (ESSL ERRSAV Subroutine)

Purpose

The ERRSAV subroutine copies an ESSL error option table entry into an 8-byte storage area that is accessible to your program.

For multithreaded application programs, if you want the error handling capabilities that this subroutine provides to be implemented on each thread created by your program, this subroutine must be called from each thread. If your application creates multiple threads, the action performed by a call to this subroutine applies to the thread that this subroutine was invoked from. For an example, see "Example of Handling Errors in a Multithreaded Application Program " on page 141.

Syntax

Language	Syntax
Fortran	CALL ERRSAV (*ierno*, *tabent*)
C and C++	errsav (*ierno*, *tabent*);

On Entry

ierno

is the error number in the option table. The entry for *ierno* in the ESSL error option table is stored in the 8-byte storage area *tabent*.

Specified as: an integer; *ierno* must be one of the error numbers in the option table. For a list of these numbers, see Table 40 on page 67.

On Return

tabent

is the storage area where the option table entry is stored.

Specified as: an area of storage of length 8-bytes.

Notes

Examples of how to use ERRSAV are provided in Chapter 4, “Coding programs,” on page 125.

ERRSET (ESSL ERRSET Subroutine)

Purpose

The ERRSET subroutine allows you to control execution when error conditions occur. It modifies the information in the ESSL error option table for the error number indicated. For a range of error messages, you can specify the following:

- How many times a particular error is allowed to occur before the program is terminated
- How many times a particular error message is printed before printing is suppressed
- Whether the ESSL error exit routine is to be invoked

For multithreaded application programs, if you want the error handling capabilities that this subroutine provides to be implemented on each thread created by your program, this subroutine must be called from each thread. If your application creates multiple threads, the action performed by a call to this subroutine applies to the thread that this subroutine was invoked from. For an example, see “Example of Handling Errors in a Multithreaded Application Program ” on page 141.

Syntax

Language	Syntax
Fortran	CALL ERRSET (*ierno*, *inoal*, *inomes*, *itrace*, *iusadr*, *irange*)
C and C++	errset (*ierno*, *inoal*, *inomes*, *itrace*, *iusadr*, *irange*);

On Entry

ierno

is the error number in the option table. The entry for *ierno* in the ESSL error option table is updated as indicated by the other arguments. Specified as: an integer; *ierno* must be one of the error numbers in the option table. For a list of these numbers, see Table 40 on page 67.

inoal

indicates the number of errors allowed before each execution is terminated, where:

If *inoal* ≤ 0, the specification is ignored, and the number-of-errors option is not changed.

If *inoal* = 1, execution is terminated after one error.

If 2 ≤ *inoal* ≤ 255, then *inoal* specifies the number of errors allowed before each execution is terminated.

If *inoal* > 255, an unlimited number of errors is allowed.

Specified as: an integer, where:

If *iusadr* = ENOTRM, then 2 ≤ *inoal* ≤ 255.

inomes
indicates the number of messages to be printed, where:

If *inomes* < 0, all messages are suppressed.

If *inomes* = 0, the number-of-messages option is not changed.

If 0 < *inomes* ≤ 255, then *inomes* specifies the number of messages to be printed.

If *inomes* > 255, an unlimited number of error messages is allowed.

Specified as: an integer.

itrace
this argument is ignored, but must be specified.

Specified as: an integer where, *itrace* = 0, 1, or 2 (for migration purposes).

iusadr
indicates whether or not the ESSL error exit routine is to be invoked, where:

If *iusadr* is zero, the option table is not altered.

If *iusadr* is one, the option table is set to show no exit routine. Therefore, standard corrective action is to be used when continuing execution.

If *iusadr* = ENOTRM, the option table entry is set to the ESSL error exit routine ENOTRM. Therefore, the ENOTRM subroutine is to be invoked after the occurrence of the indicated errors. (ENOTRM must appear in an EXTERNAL statement in your program.)

Specified: as a 32-bit integer in a 32-bit integer, 32-bit pointer environment, or as the name of a subroutine; *iusadr* = 0, 1, or ENOTRM.

Specified: as a 64-bit integer in either a 32-bit integer, 64-bit pointer environment or a 64-bit integer, 64-bit pointer environment, or as the name of a subroutine; *iusadr* = 0_8, 1_8, or ENOTRM.

irange
indicates the range of errors to be updated in the ESSL error option table, where:

If *irange* < *ierno*, the parameter is ignored.

If *irange* ≥ *ierno*, the options specified for the other parameters are to be applied to the entire range of error conditions encompassed by *ierno* and *irange*.

Specified as: an integer.

Notes

1. Examples of how to use ERRSET are provided in Chapter 4, “Coding programs,” on page 125.
2. If you specify ENOTRM for *iusadr*, then *inoal* must be in the following range: 2 ≤ *inoal* ≤ 255.

ERRSTR (ESSL ERRSTR Subroutine)

Purpose

The ERRSTR subroutine stores an entry in the ESSL error option table.

For multithreaded application programs, if you want the error handling capabilities that this subroutine provides to be implemented on each thread created by your program, this subroutine must be called from each thread. If your application creates multiple threads, the action performed by a call to this subroutine applies to the thread that this subroutine was invoked from. For an example, see "Example of Handling Errors in a Multithreaded Application Program " on page 141.

Syntax

Language	Syntax
Fortran	CALL ERRSTR (*ierno*, *tabent*)
C and C++	errstr (*ierno*, *tabent*);

On Entry

ierno

is the error number in the option table. The information in the 8-byte storage area *tabent* is stored into the entry for *ierno* in the ESSL error option table.

Specified as: an integer; *ierno* must be one of the error numbers in the option table. For a list of these numbers, see Table 40 on page 67.

tabent

is the storage area containing the table entry data.

Specified as: an area of storage of length 8-bytes.

Notes

Examples of how to use ERRSTR are provided in Chapter 4, "Coding programs," on page 125.

IESSL (Determine the Level of ESSL Installed)

Purpose

This function returns the level of ESSL installed on your system, where the level consists of a version number, release number, and modification number, plus the fix number of the most recent PTF installed.

Syntax

Language	Syntax
Fortran	IESSL ()
C and C++	iessl ();

On Return

Function value

is the level of ESSL installed on your system. It is provided as an integer in the form *vvrrmmff*, where each two digits represents a part of the level:

- *vv* is the version number.
- *rr* is the release number.
- *mm* is the modification number.
- *ff* is the fix number of the most recent PTF installed.

Returned as: an integer; *vvrrmmff* > 0.

Notes

1. To use IESSL effectively, you must install your ESSL PTFs in their proper sequential order. As part of the result, IESSL returns the value *ff* of the **most recent** PTF installed, rather than the **highest number** PTF installed. Therefore, if you do not install your PTFs sequentially, the *ff* value returned by IESSL does not reflect the actual level of ESSL.
2. Declare the IESSL function in your program as returning an integer value.

Function

The IESSL function enables you to determine the current level of ESSL installed on your system. It is useful to you in those instances where your program is using a subroutine or feature that exists only in certain levels of ESSL. It is also useful when your program is dependent upon certain PTFs being applied to ESSL.

Examples

Example 1

This example shows several ways to use the IESSL function. Most typically, you use IESSL for checking the version and release level of ESSL. Suppose you are dependent on a new capability in ESSL, such as a new subroutine or feature, provided for the first time in ESSL Version 3. You can add the following check in your program before using the new capability:

```
IF IESSL()  ≥  3010000
```

By specifying 0000 for *mmff*, the modification and fix level, you are independent of the order in which your modifications and PTFs are installed.

Less typically, you use IESSL for checking the PTF level of ESSL. Suppose you are dependent on PTF 2 being installed on your ESSL Version 3 system. You want to know whether to call a different user-callable subroutine to set up your array data. You can add the following check in your program before making the call:

```
IF IESSL()  ≥  3010002
```

If your system support group installed the ESSL PTFs in their proper sequential order, this test works properly; otherwise, it is unpredictable.

SETGPUS (Set the Number of GPUs and Identify Which GPUs ESSL Should Use)

Purpose

SETGPUS allows you to set the number and specify which GPUs ESSL should use.

Syntax

Language	Syntax
Fortran	SETGPUS (*ngpus*,*ids*)
C and C++	setgpus (*ngpus*,*ids*);

On Entry

ngpus

is the number of GPUs ESSL should use.

Specified as: an integer; $0 < ngpus \leq$ number of CUDA devices.

ids
is the array of length *ngpus* containing the IDs for the GPUs ESSL should use.

Specified as: an integer; 0 ≤ ids_i < (number of CUDA devices) for *i* = 1, *ngpus*.

Function

This subroutine allows you to set the number and specify which GPUs ESSL should use.

Error conditions

Resource Errors

1. The number of OpenMP Threads is less than *ngpus*. ESSL issues attention message 2538-2615 and uses the same number of GPUs as there are OpenMP threads.
2. Not all the CUDA devices specified by the *ids* array are in the same NVIDIA compute mode.

Input-Argument Errors

1. SETGPUS has been called either:
   - More than once
   - After the first call to any ESSL subroutine that is GPU enabled.
2. *ngpus* ≤ 0 or *ngpus* > (number of CUDA devices).
3. ids_i < 0 or ids_i > (number of CUDA devices) - 1 for *i* = 1, *ngpus*.
4. NVIDIA compute mode is PROHIBITED for GPUs identified in the *ids* array.
5. Environment variable ESSL_CUDA_HYBRID is not 'yes', 'no', or unset.
6. Environment variable ESSL_CUDA_PIN is not 'yes', 'no', 'pinned', or unset.

Examples

Example

This example shows setting 2 GPUs that ESSL should use. This call results in ESSL using GPUs 1 and 0 for CUDA applications.

Call Statement and Input:

```
             NGPUS IDS
               |    |
CALL SETGPUS( 2 , IDS)

IDS      = (1,0)
```

STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)

Purpose

This subroutine determines an optimal stride value for you to use for your input or output data when you are computing large row Fourier transforms in any of the Fourier transform subroutines, except _RCFT and _CRFT. The strides determined by this subroutine allow your arrays to fit comfortably in various levels of storage hierarchy on your particular processor, thus allowing you to improve your run-time performance.

Note: This subroutine returns a single stride value. Where you need multiple strides, you must invoke this subroutine multiple times; for example, in the multidimensional Fourier transforms and, also, when input and output data types differ. For more details, see "Function" on page 1197.

Syntax

Language	Syntax
Fortran	CALL STRIDE (*n*, *incd*, *incr*, *dt*, *iopt*)
C and C++	stride (*n*, *incd*, *incr*, *dt*, *iopt*);

On Entry

n

is the length *n* of the Fourier transform for which the optimal stride is being determined. The transform corresponding to *n* is usually a row transform; that is, the data elements are stored using a stride value.

Specified as: an integer; *n* > 0.

incd

is the minimum allowable stride for the Fourier transform for which the optimal stride is being determined. For each situation in each subroutine, there is a specific way to compute this minimum value. This is explained in Example 1—SCFT.

Specified as: an integer; *incd* > 0 or *incd* < 0.

incr

See On Return.

dt

is the data type of the numbers for the Fourier transform for which the optimal stride is being determined, where:

If *dt* = 'S', the numbers are short-precision real.

If *dt* = 'D', the numbers are long-precision real.

If *dt* = 'C', the numbers are short-precision complex.

If *dt* = 'Z', the numbers are long-precision complex.

Specified as: a single character; *dt* = 'S', 'D', 'C', or 'Z'.

iopt

is provided only for migration purposes from ESSL Version 1 and is no longer used; however, you must still specify it as a dummy argument.

Specified as: an integer; *iopt* = 0, 1, or 2.

On Return

incr

is the stride that allows you to improve your run-time performance in your Fourier transform computation on your particular processor. In general, this value differs for each processor you are running on.

Returned as: an integer; *incr* > 0 or *incr* < 0 and |*incr*| ≥ |*incd*|, where *incr* has the same sign (+ or -) as *incd*.

Notes

1. In your C program, argument *incr* must be passed by reference.
2. All subroutines accept lowercase letters for the *dt* argument.
3. For each situation in each of the Fourier transform subroutines, there is a specific way to compute the value you should specify for the *incd* argument. Details on how to compute each of these values is given in Example 1—SCFT. See the example corresponding to the Fourier transform subroutine you are using.

4. Where different data types are specified for the input and output data in your Fourier transform subroutine, you should be careful to indicate the correct data type in the *dt* argument in this subroutine.
5. For additional information on how to set up your data, see "Setting Up Your Data" on page 959.

Function

This subroutine determines an optimal stride, *incr*, for you to use for your input or output data when computing large row Fourier transforms. The stride value returned by this subroutine is based on the size and structure of your transform data, using:

- The size of each data item (*dt*)
- The minimum allowable stride for this transform (*incd*)
- The length of the transform (*n*)

This information is used in determining the optimal stride for the processor you are currently running on. The stride determined by this subroutine allows your arrays to fit comfortably in various levels of storage hierarchy for that processor, thus giving you the ability to improve your run-time performance.

You get only one stride value returned by this subroutine on each invocation. Therefore, in many instances, you may need to invoke this subroutine multiple times to obtain several stride values to use in your Fourier transform computation:

- For multidimensional Fourier transforms using several strides, this subroutine must be called once for each optimal stride you want to obtain. Successive invocations should go from the lower (earlier) dimensions to the higher (later) dimensions, because the results from the lower dimensions are used to calculate the *incd* values for the higher dimensions.
- Where input and output data have different data types and you want to obtain optimal strides for each, this subroutine must be called once for each data type.

Where multiple invocations are necessary, they are explained in Example 1—SCFT. The examples also explain how to calculate the *incd* values for each invocation. There are nine examples to cover the Fourier transform subroutines that can use the STRIDE subroutine.

After calling this subroutine and obtaining the optimal stride value, you then set up your input or output array accordingly. This may involve movement of data for input arrays or increasing the sizes of input or output arrays. To accomplish this, you may want to set up a separate subroutine with the stride values passed into it as arguments. You can then dimension your arrays in that subroutine, depending on the values calculated by STRIDE. For additional information on how to set up your data, see "Setting Up Your Data" on page 959.

Error conditions

Computational Errors
None

Input-Argument Errors

1. $n \leq 0$
2. *incd* = 0
3. *iopt* ≠ 0, 1, or 2
4. *dt* ≠ S, D, C, or Z

Examples

Example 1—SCFT

This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the SCFT subroutine.

If *inc2x* = 1, the input sequences are stored in the transposed form as rows of a two-dimensional array X(INC1X,N). In this case, the STRIDE subroutine helps in determining a good value of *inc1x* for this array. The required minimum value of *inc1x* is *m*, the number of Fourier transforms being computed. To find a good value of *inc1x*, use STRIDE as follows:

```
             N  INCD  INCR    DT   IOPT
             |   |     |       |    |
CALL STRIDE( N , M  , INC1X , 'C' , 0  )
```

Here, the arguments refer to the SCFT subroutine. In the following table, values of *inc1x* are given (as obtained from the STRIDE subroutine) for some combinations of *n* and *m* and for POWER3 with 64KB level 1 cache:

```
     N      M    INC1X

   128     64       64
   240     32       32
   240     64       65
   256    256      264
   512     60       60
  1024     64       65
```

The above example also applies when the output sequences are stored in the transposed form (*inc2y* = 1). In that case, in the above example, *inc1x* is replaced by *inc1y*.

In computing column transforms (*inc1x* = *inc1y* = 1), the values of *inc2x* and *inc2y* are not very important. For these, any value over the required minimum of *n* can be used.

Example 2--DCOSF

This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the DCOSF subroutine.

If *inc2x* = 1, the input sequences are stored in the transposed form as rows of a two-dimensional array X(INC1X,N/2+1). In this case, the STRIDE subroutine helps in determining a good value of *inc1x* for this array. The required minimum value of *inc1x* is *m*, the number of Fourier transforms being computed. To find a good value of *inc1x*, use STRIDE as follows:

```
               N     INCD  INCR    DT  IOPT
               |      |      |      |    |
CALL STRIDE( N/2+1 , M  , INC1X , 'D' , 0  )
```

Here, the arguments refer to the DCOSF subroutine. In the following table, values of *inc1x* are given (as obtained from the STRIDE subroutine) for some combinations of *n* and *m* and for POWER3 with 64KB level 1 cache:

```
     N      M    INC1X

   128     64       64
   240     32       32
   240     64       64
   256    256      264
   512     60       60
  1024     64       65
```

The above example also applies when the output sequences are stored in the transposed form (*inc2y* = 1). In that case, in the above example, *inc1x* is replaced by *inc1y*.

In computing column transforms (*inc1x* = *inc1y* = 1), the values of *inc2x* and *inc2y* are not very important. For these, any value over the required minimum of *n*/2+1 can be used.

Example 3--DSINF

This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the DSINF subroutine.

If *inc2x* = 1, the input sequences are stored in the transposed form as rows of a two-dimensional array X(INC1X,N/2). In this case, the STRIDE subroutine helps in determining a good value of *inc1x* for this array. The required minimum value of *inc1x* is *m*, the number of Fourier transforms being computed. To find a good value of *inc1x*, use STRIDE as follows:

```
              N   INCD  INCR    DT   IOPT
              |    |     |       |    |
CALL STRIDE( N/2 , M  , INC1X , 'D' , 0  )
```

Here, the arguments refer to the DSINF subroutine. In the following table, values of *inc1x* are given (as obtained from the STRIDE subroutine) for some combinations of *n* and *m* and for POWER3 with 64KB level 1 cache:

```
     N      M    INC1X

   128    64       64
   240    32       32
   240    64       64
   256   256      264
   512    60       60
  1024    64       65
```

The above example also applies when the output sequences are stored in the transposed form (*inc2y* = 1). In that case, in the above example, *inc1x* is replaced by *inc1y*.

In computing column transforms (*inc1x* = *inc1y* = 1), the values of *inc2x* and *inc2y* are not very important. For these, any value over the required minimum of *n*/2 can be used.

Example 4--SCFT2

This example shows the use of the STRIDE subroutine in computing two-dimensional transforms using the SCFT2 subroutine.

If *inc1y* = 1, the two-dimensional output array is stored in the normal form. In this case, the output array can be declared as `Y(INC2Y,N2)`, where the required minimum value of *inc2y* is *n1*. The STRIDE subroutine helps in picking a good value of *inc2y*. To find a good value of *inc2y*, use STRIDE as follows:

```
             N   INCD  INCR    DT   IOPT
             |    |     |       |    |
CALL STRIDE( N2 , N1 , INC2Y , 'C' , 0  )
```

Here, the arguments refer to the SCFT2 subroutine. In the following table, values of *inc2y* are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with *n1* = *n2* and for POWER3 with 64KB level 1 cache:

```
     N1      N2   INC2Y

     64      64      64
    128     128     136
    240     240     240
    512     512     520
    840     840     848
```

If the input array is stored in the normal form (*inc1x* = 1), the value of *inc2x* is not important. However, if you want to use the same array for input and output, you should use *inc2x* = *inc2y*.

If *inc2y* = 1, the two-dimensional output array is stored in the transposed form. In this case, the output array can be declared as `Y(INC1Y,N1)`, where the required minimum value of *inc1y* is *n2*. The STRIDE subroutine helps in picking a good value of *inc1y*. To find a good value of *inc1y*, use STRIDE as follows:

```
             N   INCD  INCR    DT   IOPT
             |    |     |       |    |
CALL STRIDE( N1 , N2 , INC1Y , 'C' , 0  )
```

Here, the arguments refer to the SCFT2 subroutine. In the following table, values of *inc1y* are given (as obtained from the STRIDE subroutine) for some combinations of *n1* and *n2* and for POWER3 with 64K level 1 cache:

```
     N1     N2   INC1Y

     60     64      64
    120    128     136
    256    240     240
    512    512     520
    840    840     848
```

If the input array is stored in the transposed form (*inc2x* = 1), the value of *inc1x* is also important. The above example can be used to find a good value of *inc1x*, by replacing *inc1y* with *inc1x*. If both arrays are stored in the transposed form, a good value for *inc1y* is also a good value for *inc1x*. In that situation, the two arrays can also be made equivalent.

Example 5--SRCFT2

This example shows the use of the STRIDE subroutine in computing two-dimensional transforms using the SRCFT2 subroutine.

For this subroutine, the output array is declared as `Y(INC2Y,N2)`, where the required minimum value of *inc2y* is *n1*/2+1. The STRIDE subroutine helps in picking a good value of *inc2y*. To find a good value of *inc2y*, use STRIDE as follows:

```
             N      INCD      INCR    DT   IOPT
             |        |         |      |    |
CALL STRIDE( N2 , N1/2 + 1 , INC2Y , 'C' , 0  )
```

Here, the arguments refer to the SRCFT2 subroutine. In the following table, values of *inc2y* are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with *n1* = *n2* and for POWER3 with 64KB level 1 cache:

```
     N1      N2    INC2Y

    240     240      121
    420     420      211
    512     512      257
    840     840      421
   1024    1024      513
   2048    2048     1032
```

For this subroutine, the leading dimension of the input array (*inc2x*) is not important. If you want to use the same array for input and output, you should use *inc2x* ≥ 2(*inc2y*).

Example 6--SCRFT2

This example shows the use of the STRIDE subroutine in computing two-dimensional transforms using the SCRFT2 subroutine.

For this subroutine, the output array is declared as `Y(INC2Y,N2)`, where the required minimum value of *inc2y* is *n1*+2. The STRIDE subroutine helps in picking a good value of *inc2y*. To find a good value of *inc2y*, use STRIDE as follows:

```
             N     INCD    INCR    DT   IOPT
             |       |       |      |    |
CALL STRIDE( N2 , N1 + 2 , INC2Y , 'S' , 0  )
```

Here, the arguments refer to the SCRFT2 subroutine. In the following table, values of *inc2y* are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with *n1* = *n2* and for POWER3 with 64KB level 1 cache:

```
     N1      N2    INC2Y

    240     240      242
    420     420      422
    512     512      514
    840     840      842
    1024    1024     1026
    2048    2048     2064
```

For this subroutine, the leading dimension of the input array (*inc2x*) is also important. In general, *inc2x* = *inc2y*/2 is a good choice. This is also the requirement if you want to use the same array for input and output.

Example 7--SCFT3

This example shows the use of the STRIDE subroutine in computing three-dimensional transforms using the SCFT3 subroutine.

For this subroutine, the strides for the input array are not important. They are important for the output array. The STRIDE subroutine helps in picking good values of *inc2y* and *inc3y*. This requires two calls to the STRIDE subroutine as shown below. First, you should find a good value for *inc2y*. The minimum acceptable value for *inc2y* is *n1*.

```
             N   INCD  INCR    DT   IOPT
             |    |      |      |    |
CALL STRIDE( N2 , N1 , INC2Y , 'C' , 0  )
```

Here, the arguments refer to the SCFT3 subroutine. Next, you should find a good value for *inc3y*. The minimum acceptable value for *inc3y* is (*n2*)(*inc2y*).

```
             N      INCD    INCR    DT   IOPT
             |       |       |       |    |
CALL STRIDE( N3 , N2*INC2Y, INC3Y , 'C' , 0  )
```

If *inc3y* turns out to be a multiple of *inc2y*, then Y can be declared a three-dimensional array as `Y(INC2Y,INC3Y/INC2Y,N3)`. For large problems, this may not happen. In that case, you can declare the Y array as a two-dimensional array `Y(0:INC3Y-1,0:N3-1)` or a one-dimensional array `Y(0:INC3Y*N3-1)`. Using zero-based indexing, the element *y(k1,k2,k3)* is stored in the following location in these arrays:

- For the two-dimensional array, location *(k1+k2*inc2y,k3)*
- For the one-dimensional array, location *(k1+k2*inc2y+k3*inc3y)*

In the following table, values of *inc2y* and *inc3y* are given (as obtained from the STRIDE subroutine) for some three-dimensional arrays with *n1* = *n2* = *n3* and for POWER3 with 64KB level 1 cache:

```
  N1,N2,N3    INC2Y    INC3Y

        30       30      900
        32       32     1032
        64       64     4112
       120      120    14408
       128      136    17416
       240      240    57608
       256      264    67592
       420      420   176400
```

As mentioned before, the strides of the input array are not important. The array can be declared as a three-dimensional array. If you want to use the same array for input and output, the requirements are *inc2x* ≥ *inc2y* and *inc3x* ≥ *inc3y*. A simple thing to do is to use *inc2x* = *inc2y* and make *inc3x* a multiple of *inc2x* not smaller than *inc3y*. Then X can be declared as a three-dimensional array `X(INC2X,INC3X/INC2X,N3)`.

Example 8--SRCFT3

This example shows the use of the STRIDE subroutine in computing three-dimensional transforms using the SRCFT3 subroutine.

For this subroutine, the strides for the input array are not important. They are important for the output array. The STRIDE subroutine helps in picking good values of *inc2y* and *inc3y*. This requires two calls to the STRIDE subroutine as shown below. First, you should find a good value for *inc2y*. The minimum acceptable value for *inc2y* is *n1*/2+1.

```
              N       INCD    INCR    DT   IOPT
              |        |       |      |     |
CALL STRIDE( N2 , N1/2 + 1 , INC2Y , 'C' , 0  )
```

Here, the arguments refer to the SRCFT3 subroutine. Next, you should find a good value for *inc3y*. The minimum acceptable value for *inc3y* is (*n2*)(*inc2y*).

```
              N      INCD     INCR    DT   IOPT
              |       |        |      |     |
CALL STRIDE( N3 , N2*INC2Y , INC3Y , 'C' , 0  )
```

If *inc3y* turns out to be a multiple of *inc2y*, then Y can be declared a three-dimensional array as `Y(INC2Y,INC3Y/INC2Y,N3)`. For large problems, this may not happen. In that case, you can declare the Y array as a two-dimensional array `Y(0:INC3Y-1,0:N3-1)` or a one-dimensional array `Y(0:INC3Y*N3-1)`. Using zero-based indexing, the element *y(k1,k2,k3)* is stored in the following location in these arrays:

- For the two-dimensional array, location *(k1+k2*inc2y,k3)*
- For the one-dimensional array, location *(k1+k2*inc2y+k3*inc3y)*

In the following table, values of *inc2y* and *inc3y* are given (as obtained from the STRIDE subroutine) for some three-dimensional arrays with *n1* = *n2* = *n3* and for POWER3 with 64KB level 1 cache:

```
  N1,N2,N3   INC2Y      INC3Y

        30      16        488
        32      17        552
        64      33       2128
       120      61       7320
       128      65       8328
       240     121      29064
       256     129      33032
       420         211      88620
```

As mentioned before, the strides of the input array are not important. The array can be declared as a three-dimensional array. If you want to use the same array for input and output, the requirements are *inc2x* ≥ 2(*inc2y*) and *inc3x* ≥ 2(*inc3y*). A simple thing to do is to use *inc2x* = 2(*inc2y*) and make *inc3x* a multiple of *inc2x* not smaller than 2(*inc3y*). Then X can be declared as a three-dimensional array `X(INC2X,INC3X/INC2X,N3)`.

Example 9--SCRFT3

This example shows the use of the STRIDE subroutine in computing three-dimensional transforms using the SCRFT3 subroutine.

The STRIDE subroutine helps in picking good values of *inc2y* and *inc3y*. This requires two calls to the STRIDE subroutine as shown below. First, you should find a good value for *inc2y*. The minimum acceptable value for *inc2y* is *n1*+2.

```
              N     INCD    INCR    DT   IOPT
              |      |       |      |     |
CALL STRIDE( N2 , N1 + 2 , INC2Y , 'S' , 0  )
```

Here, the arguments refer to the SCRFT3 subroutine. Next, you should find a good value for *inc3y*. The minimum acceptable value for *inc3y* is (*n2*)(*inc2y*).

```
              N      INCD     INCR    DT   IOPT
              |        |       |      |     |
CALL STRIDE( N3 , N2*INC2Y , INC3Y , 'S' , 0  )
```

If *inc3y* turns out to be a multiple of *inc2y*, then Y can be declared a three-dimensional array as `Y(INC2Y,INC3Y/INC2Y,N3)`. For large problems, this may not happen. In that case, you can declare the Y array as a two-dimensional array `Y(0:INC3Y-1,0:N3-1)` or a one-dimensional array

`Y(0:INC3Y*N3-1)`. Using zero-based indexing, the element *y(k1,k2,k3)* is stored in the following location in these arrays:

- For the two-dimensional array, location *(k1+k2*inc2y,k3)*
- For the one-dimensional array, location *(k1+k2*inc2y+k3*inc3y)*

In the following table, values of *inc2y* and *inc3y* are given (as obtained from the STRIDE subroutine) for some three-dimensional arrays with *n1* = *n2* = *n3* and for POWER3 with 64KB level 1 cache:

```
  N1,N2,N3    INC2Y     INC3Y

        30       32       976
        32       34      1104
        64       66      4256
       120      122     14640
       128      130     16656
       240      242     58128
       256      258     66064
       420      422    177240
```

For this subroutine, the strides (*inc2x* and *inc3x*) of the input array are also important. In general, *inc2x* = *inc2y*/2 and *inc3x* = *inc3y*/2 are good choices. These are also the requirement if you want to use the same array for input and output.

Example 10--SCFTD, D = 1

This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the SCFTD subroutine.

If *incmx* = 1, the input sequences are stored in the transposed form as rows of a two-dimensional array `X(INCX(1),N(1))`. In this case, the STRIDE subroutine helps in determining a good value of $incx_1$ for this array. The required minimum value of $incx_1$ is *m*, the number of Fourier transforms being computed. To find a good value of $incx_1$, use STRIDE as follows:

```
             N    INCD   INCR      DT  IOPT
             |      |      |        |    |
CALL STRIDE( N(1) , M  , INCX(1) , 'C' , 0 )
```

Here, the arguments refer to the SCFTD subroutine. In the following table, values of $incx_1$ are given (as obtained from the STRIDE subroutine) for some combinations of n_1 and *m* and for POWER6® with 64KB level 1 cache:

```
     N      M    INC1X

   128     64       66
   240     32       34
   240     64       66
   256    256      264
   512     60       60
  1024     64       66
```

The above example also applies when the output sequences are stored in the transposed form (*incmy* = 1). In that case, in the above example, $incx_1$ is replaced by $incy_1$.

In computing column transforms ($incx_1$ = $incy_1$ = 1), the values of *incmx* and *incmy* are not very important. For these, any value over the required minimum of n_1 can be used.

Example 11--SCFTD, D = 2

This example shows the use of the STRIDE subroutine in computing two-dimensional transforms using the SCFTD subroutine with *m* = 1.

If $incy_1$ = 1, the two-dimensional output array is stored in the normal form. In this case, the output array can be declared as `Y(INCY(2),N(2))`, where the required minimum value of $incy_2$ is n_1. The

STRIDE subroutine helps in picking a good value of *incy*$_2$. To find a good value of *incy*$_2$, use STRIDE as follows:

```
              N     INCD    INCR      DT   IOPT
              |      |       |         |    |
CALL STRIDE( N(2) , N(1) , INCY(2) , 'C' , 0 )
```

Here, the arguments refer to the SCFTD subroutine. In the following table, values of *incy*$_2$ are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with $n_1 = n_2$ and for POWER6 with 64KB level 1 cache:

```
    N(1)    N(2)   INCY(2)

     64      64      64
    128     128     136
    240     240     240
    512     512     520
    840     840     840
```

If the input array is stored in the normal form (*incx*$_1$ = 1), the value of *incx*$_2$ is not important. However, if you want to use the same array for input and output, you should use *incx*$_2$ = *incy*$_2$.

If *incy*$_2$ = 1, the two-dimensional output array is stored in the transposed form. In this case, the output array can be declared as `Y(INCY(1),N(1))`, where the required minimum value of *incy*$_1$ is n_2. The STRIDE subroutine helps in picking a good value of *incy*$_1$. To find a good value of *incy*$_1$, use STRIDE as follows:

```
              N     INCD    INCR      DT   IOPT
              |      |       |         |    |
CALL STRIDE( N(1) , N(2) , INCY(1) , 'C' , 0 )
```

Here, the arguments refer to the SCFTD subroutine. In the following table, values of *incy*$_1$ are given (as obtained from the STRIDE subroutine) for some combinations of n_1 and n_2 and for POWER6 with 64KB level 1 cache:

```
    N(1)   N(2)   INCY(1)

     60     64      64
    120    128     136
    256    240     240
    512    512     520
    840    840     840
```

If the input array is stored in the transposed form (*incx*$_2$ = 1), the value of *incx*$_1$ is also important. The above example can be used to find a good value of *incx*$_1$, by replacing *incy*$_1$ with *incx*$_1$. If both arrays are stored in the transposed form, a good value for *incy*$_1$ is also a good value for *incx*$_1$. In that situation, the two arrays can also be made equivalent.

Example 12--SCFTD, D = 3

This example shows the use of the STRIDE subroutine in computing three-dimensional transforms using the SCFTD subroutine with *m* = 1.

For this subroutine, the strides for the input array are not important. They are important for the output array. The STRIDE subroutine helps in picking good values of *incy*$_2$ and *incy*$_3$. This requires two calls to the STRIDE subroutine as shown below. First, you should find a good value for *incy*$_2$. The minimum acceptable value for *incy*$_2$ is n_1.

```
              N      INCD     INCR    DT   IOPT
              |       |        |       |    |
CALL STRIDE( N(2) , N(1) , INCY(2) , 'C' , 0 )
```

Here, the arguments refer to the SCFTD subroutine. Next, you should find a good value for $incy_3$. The minimum acceptable value for $incy_3$ is $(n_2)(incy_2)$ assuming $incy_1 = 1$.

```
                N        INCD       INCR      DT  IOPT
                |          |          |        |    |
CALL STRIDE( N(3) , N(2)*INCY(2), INCY(3) , 'C' , 0 )
```

If $incy_3$ turns out to be a multiple of $incy_2$, then Y can be declared a three-dimensional array as `Y(INCY(2),INCY(3)/INCY(2),N(3))`. For large problems, this may not happen. In that case, you can declare the Y array as a two-dimensional array `Y(0:INCY(3)-1,0:N(3)-1)` or a one-dimensional array `Y(0:INCY(3)*N(3)-1)`. Using zero-based indexing, the element $y_{k1,k2,k3}$ is stored in the following location in these arrays:

- For the two-dimensional array, location $(k1+k2*incy_2,k3)$
- For the one-dimensional array, location $(k1+k2*incy_2+k3*incy_3)$

In the following table, values of $incy_2$ and $incy_3$ are given (as obtained from the STRIDE subroutine) for some three-dimensional arrays with $n_1 = n_2 = n_3$ and for POWER6 with 64KB level 1 cache:

```
  N1,N2,N3    INCY(2)   INCY(3)

        30       30       900
        32       32      1032
        64       64      4104
       120      120     14400
       128      136     17416
       240      240     57608
       256      264     67592
       420      420    176400
```

As mentioned before, the strides of the input array are not important. The array can be declared as a three-dimensional array. If you want to use the same array for input and output, the requirements are $incx_2 \geq incy_2$ and $incx_3 \geq incy_3$. A simple thing to do is to use $incx_2 = incy_2$ and make $incx_3$ a multiple of $incx_2$ not smaller than $incy_3$. Then X can be declared as a three-dimensional array `X(INCX(2),INCX(3)/INCX(2),N(3))`.

Example 13--SRCFTD, D = 1

This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the SRCFTD subroutine.

If *incmx* equal to 1, the input sequences are stored in the transposed form as rows of a two-dimensional array `X(INCX(1),N(1))`. In this case, the STRIDE subroutine helps in determining a good value of $incx_1$ for this array. The required minimum value of $incx_1$ is *m*, the number of Fourier transforms being computed. To find a good value of $incx_1$, use STRIDE as follows:

```
               N    INCD   INCR      DT  IOPT
               |      |      |        |    |
CALL STRIDE( N(1) , M  , INCX(1) , 'S' , 0 )
```

Here, the arguments refer to the SRCFTD subroutine. In the following table, values of $incx_1$ are given (as obtained from the STRIDE subroutine) for some combinations of n_1 and *m* and for POWER6 with 64KB level 1 cache:

```
   N(1)     M    INCX(1)

    128    64       64
    240    32       32
    240    64       68
    256   256      272
    512    60       60
   1024    64       64
```

If *incmy* equal to 1, the output sequences are stored in the transposed form as rows of a two-dimensional array `Y(INCY(1),N(1)/2+1)`. In this case, the STRIDE subroutine helps in determining a good value of $incy_1$ for this array. The required minimum value of $incy_1$ is *m*, the number of Fourier transforms being computed. To find a good value of $incy_1$, use STRIDE as follows:

```
              N         INCD   INCR     DT  IOPT
              |           |      |       |    |
CALL STRIDE( N(1)/2+1 , M  , INCY(1) , 'C' , 0 )
```

Here, the arguments refer to the SRCFTD subroutine. In the following table, values of $incy_1$ are given (as obtained from the STRIDE subroutine) for some combinations of n_1 and *m* and for POWER6 with 64KB level 1 cache:

```
    N(1)     M    INCY(1)

     128    64       66
     240    32       32
     240    64       66
     256   256      264
     512    60       60
    1024    64       66
```

In computing column transforms ($incx_1$ equal to $incy_1$ equal to 1), the values of *incmx* and *incmy* are not very important. For these, any value over the required minimum can be used.

Example 14--SRCFTD, D = 2

This example shows the use of the STRIDE subroutine in computing two-dimensional transforms using the SRCFTD subroutine with *m* equal to 1.

If $incy_1$ equal to 1, the two-dimensional output array is stored in the normal form. In this case, the output array can be declared as `Y(INCY(2),N(2))`, where the required minimum value of $incy_2$ is $n_1/2+1$. The STRIDE subroutine helps in picking a good value of $incy_2$. To find a good value of $incy_2$, use STRIDE as follows:

```
              N         INCD    INCR      DT  IOPT
              |           |       |        |    |
CALL STRIDE( N(2) , N(1)/2+1 , INCY(2) , 'C' , 0 )
```

Here, the arguments refer to the SRCFTD subroutine. In the following table, values of $incy_2$ are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with n_1 equal to n_2 and for POWER6 with 64KB level 1 cache:

```
     N(1)    N(2)   INCY(2)

     240     240     122
     420     420     212
     512     512     258
     840     840     422
    1024    1024     514
    2048    2048    1026
```

If the input array is stored in the normal form ($incx_1$ equal to 1), the value of $incx_2$ is not important. However, if you want to use the same array for input and output, you should use $incx_2$ equal to $2(incy_2)$.

If $incy_2$ equal to 1, the two-dimensional output array is stored in the transposed form. In this case, the output array can be declared as `Y(INCY(1),N(1)/2+1)`, where the required minimum value of $incy_1$ is n_2. The STRIDE subroutine helps in picking a good value of $incy_1$. To find a good value of $incy_1$, use STRIDE as follows:

```
              N         INCD    INCR      DT  IOPT
              |           |       |        |    |
CALL STRIDE( N(1)/2+1 , N(2) , INCY(1) , 'C' , 0 )
```

Here, the arguments refer to the SRCFTD subroutine. In the following table, values of $incy_1$ are given (as obtained from the STRIDE subroutine) for some combinations of n_1 and n_2 and for POWER6 with 64KB level 1 cache:

```
    N(1)   N(2)   INCY(1)

    240     240     240
    420     420     420
    512     512     520
    840     840     840
   1024    1024    1032
   2048    2048    2056
```

Example 15--SRCFTD, D = 3

This example shows the use of the STRIDE subroutine in computing three-dimensional transforms using the SRCFTD subroutine with *m* equal to 1.

For this subroutine, the strides for the input array are not important. They are important for the output array. The STRIDE subroutine helps in picking good values of $incy_2$ and $incy_3$. This requires two calls to the STRIDE subroutine as shown below. First, you should find a good value for $incy_2$. The minimum acceptable value for $incy_2$ is $n_1/2+1$.

```
              N        INCD      INCR     DT  IOPT
              |         |          |       |    |
CALL STRIDE( N(2) , N(1)/2+1 , INCY(2) , 'C' , 0 )
```

Here, the arguments refer to the SRCFTD subroutine. Next, you should find a good value for $incy_3$. The minimum acceptable value for $incy_3$ is $(n_2)(incy_2)$ assuming $incy_1$ equal to 1.

```
               N        INCD       INCR      DT  IOPT
               |         |          |         |    |
CALL STRIDE( N(3) , N(2)*INCY(2), INCY(3) , 'C' , 0 )
```

If $incy_3$ turns out to be a multiple of n_2, then Y can be declared a three-dimensional array as `Y(INCY(2),INCY(3)/INCY(2),N(3))`. For large problems, this may not happen. In that case, you can declare the Y array as a two-dimensional array `Y(0:INCY(3)-1,0:N(3)-1)` or a one-dimensional array `Y(0:INCY(3)*N(3)-1)`. Using zero-based indexing, the element $y_{k1,k2,k3}$ is stored in the following location in these arrays:

- For the two-dimensional array, location $(k1+k2*incy_2,k3)$
- For the one-dimensional array, location $(k1+k2*incy_2+k3*incy_3)$

In the following table, values of $incy_2$ and $incy_3$ are given (as obtained from the STRIDE subroutine) for some three-dimensional arrays with n_1 equal to n_2 equal to n_3 and for POWER6 with 64KB level 1 cache:

```
N(1),N(2),N(3) INCY(2) INCY(3)

        30       16      480
        32       18      576
        64       34     2176
       120       62     7440
       128       66     8456
       240      122    29280
       256      130    33288
       420      212    89040
```

As mentioned before, the strides of the input array are not important. The array can be declared as a three-dimensional array. If you want to use the same array for input and output, the requirements are $incx_2$ equal to $2(incy_2)$ and $incx_3$ equal to $2(incy_3)$.

Example 16--SCRFTD, D = 1

This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the SCRFTD subroutine.

If *incmx* equal to 1, the input sequences are stored in the transposed form as rows of a two-dimensional array X(INCX(1),N(1)). In this case, the STRIDE subroutine helps in determining a good value of $incx_1$ for this array. The required minimum value of $incx_1$ is *m*, the number of Fourier transforms being computed. To find a good value of $incx_1$, use STRIDE as follows:

```
             N         INCD   INCR     DT  IOPT
             |          |      |        |    |
CALL STRIDE( N(1)/2+1 , M  , INCX(1) , 'C' , 0 )
```

Here, the arguments refer to the SCRFTD subroutine. In the following table, values of $incx_1$ are given (as obtained from the STRIDE subroutine) for some combinations of n_1 and *m* and for POWER6 with 64KB level 1 cache:

```
   N(1)     M    INCX(1)

    128    64       66
    240    32       32
    240    64       66
    256   256      264
    512    60       60
   1024    64       66
```

If *incmy* equal to 1, the output sequences are stored in the transposed form as rows of a two-dimensional array Y(INCY(1),N(1)). In this case, the STRIDE subroutine helps in determining a good value of $incy_1$ for this array. The required minimum value of $incy_1$ is m, the number of Fourier transforms being computed. To find a good value of $incy_1$, use STRIDE as follows:

```
             N     INCD   INCR     DT  IOPT
             |      |      |        |    |
CALL STRIDE( N(1) , M  , INCY(1) , 'S' , 0 )
```

Here, the arguments refer to the SCRFTD subroutine. In the following table, values of $incy_1$ are given (as obtained from the STRIDE subroutine) for some combinations of n_1 and *m* and for POWER6 with 64KB level 1 cache:

```
   N(1)     M    INCY(1)

    128    64       64
    240    32       32
    240    64       68
    256   256      272
    512    60       60
   1024    64       64
```

In computing column transforms ($incx_1$ equal to $incy_1$ equal to 1), the values of *incmx* and *incmy* are not very important. For these, any value over the required minimum can be used.

Example 17--SCRFTD, D = 2

This example shows the use of the STRIDE subroutine in computing two-dimensional transforms using the SCRFTD subroutine with *m* equal to 1.

If $incy_1$ equal to 1, the two-dimensional output array is stored in the normal form. In this case, the output array can be declared as Y(INCY(2),N(2)), where the required minimum value of $incy_2$ is n_1+2. The STRIDE subroutine helps in picking a good value of $incy_2$. To find a good value of $incy_2$, use STRIDE as follows:

```
             N      INCD      INCR     DT  IOPT
             |       |         |        |    |
CALL STRIDE( N(2) , N(1)+2 , INCY(2) , 'S' , 0 )
```

Here, the arguments refer to the SCRFTD subroutine. In the following table, values of $incy_2$ are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with n_1 equal to n_2 and for POWER6 with 64KB level 1 cache:

```
    N(1)    N(2)  INCY(2)

    240     240     244
```

```
     420     420     424
     512     512     516
     840     840     844
    1024    1024    1028
    2048    2048    2052
```

If the input array is stored in the normal form ($incx_1$ equal to 1), the value of $incx_2$ is not important. However, if you want to use the same array for input and output, you should use $incy_2$ equal to 2($incx_2$).

If $incy_2$ equal to 1, the two-dimensional output array is stored in the transposed form. In this case, the output array can be declared as `Y(INCY(1),N(1)+2)`, where the required minimum value of $incy_1$ is n_2. The STRIDE subroutine helps in picking a good value of $incy_1$. To find a good value of $incy_1$, use STRIDE as follows:

```
             N       INCD    INCR      DT  IOPT
             |        |       |         |    |
CALL STRIDE( N(1)+2 , N(2) , INCY(1) , 'S' , 0 )
```

Here, the arguments refer to the SCRFTD subroutine. In the following table, values of $incy_1$ are given (as obtained from the STRIDE subroutine) for some combinations of n_1 and n_2 and for POWER6 with 64KB level 1 cache:

```
    N(1)   N(2)   INCY(1)

     240     240     240
     420     420     420
     512     512     528
     840     840     840
    1024    1024    1040
    2048    2048    2064
```

Example 18--SCRFTD, D = 3

This example shows the use of the STRIDE subroutine in computing three-dimensional transforms using the SCRFTD subroutine with m equal to 1.

For this subroutine, the strides for the input array are not important. They are important for the output array. The STRIDE subroutine helps in picking good values of $incy_2$ and $incy_3$. This requires two calls to the STRIDE subroutine as shown below. First, you should find a good value for $incy_2$. The minimum acceptable value for $incy_2$ is n_1+2.

```
             N        INCD    INCR     DT  IOPT
             |         |       |        |    |
CALL STRIDE( N(2) , N(1)+2 , INCY(2) , 'S' , 0 )
```

Here, the arguments refer to the SCRFTD subroutine. Next, you should find a good value for $incy_3$. The minimum acceptable value for $incy_3$ is (n_2)($incy_2$) assuming $incy_1$ equal to 1.

```
              N        INCD       INCR      DT  IOPT
              |         |          |         |    |
CALL STRIDE( N(3) , N(2)*INCY(2), INCY(3) , 'S' , 0 )
```

If $incy_3$ turns out to be a multiple of $incy_2$, then Y can be declared a three-dimensional array as `Y(INCY(2),INCY(3)/INCY(2),N(3))`. For large problems, this may not happen. In that case, you can declare the Y array as a two-dimensional array `Y(0:INCY(3)-1,0:N(3)-1)` or a one-dimensional array `Y(0:INCY(3)*N(3)-1)`. Using zero-based indexing, the element $y_{k1,k2,k3}$ is stored in the following location in these arrays:

- For the two-dimensional array, location (k1+k2*$incy_2$,k3)
- For the one-dimensional array, location (k1+k2*$incy_2$+k3*$incy_3$)

In the following table, values of $incy_2$ and $incy_3$ are given (as obtained from the STRIDE subroutine) for some three-dimensional arrays with n_1 equal to n_2 equal to n_3 and for POWER6 with 64KB level 1 cache:

```
N(1),N(2),N(3) INCY(2) INCY(3)

        30       32      960
        32       36     1152
        64       68     4352
       120      124    14880
       128      132    16912
       240      244    58560
       256      260    66576
       420      424   178080
```

As mentioned before, the strides of the input array are not important. The array can be declared as a three-dimensional array. If you want to use the same array for input and output, the requirements are $incy_2$ equal to 2($incx_2$) and $incy_3$ equal to 2($incx_3$).

>|SLAMCH and DLAMCH (Determine machine parameters for floating-point arithmetic)

Purpose

SLAMCH and DLAMCH determine the machine parameters for floating-point arithmetic.

Table 267: Data Types.

Data Types	
Result	**Subprogram**
Short-precision real	SLAMCH
Long-precision real	DLAMCH

Syntax

Table 268: Syntax for supported computing languages.

Language	**Syntax**
Fortran	SLAMCH | DLAMCH (*cmach*)
C and C++	slamch | dlamch (*cmach*);
LAPACKE	LAPACKE_slamch | LAPACKE_dlamch (*cmach*);

On Entry

cmach

If *cmach* = 'E', the relative machine precision (eps) is returned.

If *cmach* = 'S', the safe minimum (sfmin) is returned, such that 1.0/sfmin does not overflow.

If *cmach* = 'B', the base of the machine (base) is returned.

If *cmach* = 'P', *eps*base* (prec) is returned.

If *cmach* = 'N', the number of (base) digits in the mantissa (t) is returned.

If *cmach* = 'R', one is returned when rounding occurs in addition; otherwise, zero (rnd) is returned.

If *cmach* = 'M', the minimum exponent before gradual underflow (emin) is returned.

If *cmach* = 'U', *underflow_threshold - base**(emin-1.0)* (rmin) is returned.

If *cmach* = 'L', the largest exponent before overflow (emax) is returned.

If *cmach* = 'O', *overflow_threshold - (base**emax)*(1.0-eps)* (rmax) is returned.

Specified as: a single character. It can be any value.

On Return

Function value

is the machine parameter. Returned as: a number of the data type indicated in Table 267 on page 1210.

Note: Declare this function in your program as returning a value of the type indicated in Table 267 on page 1210.

Notes

1. All subroutines accept lowercase letters for the *cmach* argument.
2. If you specify a value that is not listed for the *cmach* argument, a zero is returned.

Function

SLAMCH and DLAMCH determine the machine parameters for floating-point arithmetic.

Error conditions

Computational Errors

None

Input-Argument Errors

None

Examples

Example 1

The following example determines the relative machine precision.

Call Statement and Input:

```
                CMACH
                |
EPS = SLAMCH ('E')
```

Output:

```
EPS = 0.5960464478E-07
```

Example 2

The following example determines the relative machine precision.

Call Statement and Input:

```
                CMACH
                |
EPS = DLAMCH ('E')
```

Output:

```
EPS = 0.111022302462515654D-15
```

DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)

Purpose

This subroutine converts either *m* by *n* general sparse matrix ***A*** or symmetric sparse matrix ***A*** of order *n* from storage-by-rows to compressed-matrix storage mode, where matrix ***A*** contains long-precision real numbers.

Syntax

Language	**Syntax**
Fortran	CALL DSRSM (*iopt*, *ar*, *ja*, *ia*, *m*, *nz*, *ac*, *ka*, *lda*)
C and C++	dsrsm (*iopt*, *ar*, *ja*, *ia*, *m*, *nz*, *ac*, *ka*, *lda*);

On Entry

iopt

indicates the storage variation used for sparse matrix ***A*** storage-by-rows:

If *iopt* = 0, matrix ***A*** is a general sparse matrix, where all the nonzero elements in matrix ***A*** are used to set up the storage arrays.

If *iopt* = 1, matrix ***A*** is a symmetric sparse matrix, where only the upper triangle and diagonal elements are used to set up the storage arrays.

Specified as: an integer; *iopt* = 0 or 1.

ar

is the sparse matrix ***A***, stored by rows in an array, referred to as AR. The *iopt* argument indicates the storage variation used for storing matrix ***A***. Specified as: a one-dimensional array, containing long-precision real numbers. The number of elements, *ne*, in this array can be determined by subtracting 1 from the value in IA(m+1).

ja

is the array, referred to as JA, containing the column numbers of each nonzero element in sparse matrix ***A***.

Specified as: a one-dimensional array, containing integers; 1 ≤ (JA elements) ≤ *n*. The number of elements, *ne*, in this array can be determined by subtracting 1 from the value in IA(m+1).

ia

is the row pointer array, referred to as IA, containing the starting positions of each row of matrix ***A*** in array AR and one position past the end of array AR. Specified as: a one-dimensional array of (at least) length m+1, containing integers; IA(i+1) ≥ IA(i) for i = 1, m+1.

m

is the number of rows in sparse matrix ***A***. Specified as: an integer; $m \geq 0$.

nz

is the number of columns in output arrays AC and KA that are available for use.

Specified as: an integer; $nz > 0$.

ac

See On Return.

ka

See On Return.

lda

is the size of the leading dimension of the arrays specified for *ac* and *ka*.

Specified as: an integer; $0 < lda \leq m$.

On Return

nz
: is the maximum number of nonzero elements, *nz*, in each row of matrix ***A***, which is stored in compressed-matrix storage mode. Returned as: an integer; (input argument) *nz* ≤ (output argument) *nz*.

ac
: is the *m* by *n* general sparse matrix ***A*** or symmetric matrix ***A*** of order *n* stored in compressed-matrix storage mode in an array, referred to as AC. Returned as: an *lda* by at least (input argument) *nz* array, containing long-precision real numbers, where only the first (output argument) *nz* columns are used to store the matrix.

ka
: is the array, referred to as KA, containing the column numbers of the matrix ***A*** elements that are stored in the corresponding positions in array AC. Returned as: an *lda* by at least (input argument) *nz* array, containing integers, where only the first (output argument) *nz* columns are used to store the column numbers.

Notes

1. In your C program, argument *nz* must be passed by reference.
2. The value specified for input argument *nz* should be greater than or equal to the number of nonzero elements you estimate to be in each row of sparse matrix ***A***. The value returned in output argument *nz* corresponds to the *nz* value defined for compressed-matrix storage mode. This value is less than or equal to the value specified for input argument *nz*.
3. For a description of the storage modes for sparse matrices, see "Compressed-Matrix Storage Mode" on page 110 and "Storage-by-Rows" on page 115.

Function

A sparse matrix ***A*** is converted from storage-by-rows (using arrays AR, JA, and IA) to compressed-matrix storage mode (using arrays AC and KA). The argument *iopt* indicates whether the input matrix ***A*** is stored by rows using the storage variation for general sparse matrices or for symmetric sparse matrices. See reference ["89" on page 1258].

This subroutine is meant for existing programs that need to convert their sparse matrices to a storage mode compatible with some of the ESSL sparse matrix subroutines, such as DSMMX.

Error conditions

Computational Errors

None

Input-Argument Errors

1. *iopt* ≠ 0 or 1
2. *m* < 0
3. *lda* < 1
4. *lda* < *m*
5. *nz* ≤ 0
6. IA(*m*+1) < 1
7. IA(*i*+1)-IA(*i*) < 0, for any *i* = 1, *m*
8. *nz* is too small to store matrix ***A*** in array AC, where:
   - If *iopt* = 0 , AC and KA are not modified.
   - If *iopt* = 1 , AC and KA are modified.

Examples

Example 1

This example shows a general sparse matrix ***A***, which is stored by rows and converted to compressed-matrix storage mode, where sparse matrix ***A*** is:

```
                     | 11.0   0.0   0.0  14.0 |
                     |  0.0  22.0   0.0  24.0 |
                     |  0.0   0.0  33.0  34.0 |
                     |  0.0   0.0   0.0  44.0 |
```

Because there is a maximum of only two nonzero elements in each row of ***A***, and argument *nz* is specified as 5, columns 3 through 5 of arrays AC and KA are not used.

Call Statement and Input:

```
           IOPT  AR   JA   IA   M   NZ  AC   KA  LDA
            |    |    |    |    |   |   |    |    |
CALL DSRSM( 0  , AR , JA , IA , 4 , 5 , AC , KA , 4 )

AR       =  (11.0, 14.0, 22.0, 24.0, 33.0, 34.0, 44.0)
JA       =  (1, 4, 2, 4, 3, 4, 4)
IA       =  (1, 3, 5, 7, 8)
```

Output:

```
NZ       =  2


        | 11.0  14.0  .  .  . |
AC   =  | 22.0  24.0  .  .  . |
        | 33.0  34.0  .  .  . |
        | 44.0   0.0  .  .  . |
```

```
        | 1  4  .  .  . |
KA   =  | 2  4  .  .  . |
        | 3  4  .  .  . |
        | 4  4  .  .  . |
```

Example 2

This example shows a symmetric sparse matrix ***A***, which is stored by rows and converted to compressed-matrix storage mode, where sparse matrix ***A*** is:

```
                     | 11.0   0.0   0.0  14.0 |
                     |  0.0  22.0   0.0  24.0 |
                     |  0.0   0.0  33.0  34.0 |
                     | 14.0  24.0  34.0  44.0 |
```

Because there is a maximum of only four nonzero elements in each row of ***A***, and argument *nz* is specified as 6, columns 5 and 6 of arrays AC and KA are not used.

Call Statement and Input:

```
           IOPT  AR   JA   IA   M   NZ  AC   KA  LDA
            |    |    |    |    |   |   |    |    |
CALL DSRSM( 1  , AR , JA , IA , 4 , 6 , AC , KA , 4 )

AR       =  (11.0, 14.0, 22.0, 24.0, 33.0, 34.0, 44.0)
JA       =  (1, 4, 2, 4, 3, 4, 4)
IA       =  (1, 3, 5, 7, 8)
```

Output:

```
NZ       =  4


        | 11.0  14.0   0.0   0.0  .  . |
AC   =  | 22.0  24.0   0.0   0.0  .  . |
        | 33.0  34.0   0.0   0.0  .  . |
        | 44.0  24.0  34.0  14.0  .  . |
```

```
        | 1  4  4  4  .  . |
KA   =  | 2  4  4  4  .  . |
        | 3  4  4  4  .  . |
        | 4  2  3  1  .  . |
```

DGKTRN (For a General Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode)

Purpose

This subroutine converts general sparse matrix ***A*** of order *n* from one skyline storage mode to another—that is, between the following:

- Diagonal-out skyline storage mode
- Profile-in skyline storage mode

Syntax

Language	Syntax
Fortran	CALL DGKTRN (*n*, *au*, *nu*, *idu*, *al*, *nl*, *idl*, *itran*, *aux*, *naux*)
C and C++	dgktrn (*n*, *au*, *nu*, *idu*, *al*, *nl*, *idl*, *itran*, *aux*, *naux*);

On Entry

n
is the order of general sparse matrix ***A***. Specified as: an integer; *n* ≥ 0.

au
is the array, referred to as AU, containing the upper triangular part of general sparse matrix ***A***, stored as follows, where:

If `ITRAN(1)` = 0, ***A*** is stored in diagonal-out skyline storage mode.

If `ITRAN(1)` = 1, ***A*** is stored in profile-in skyline storage mode.

Specified as: a one-dimensional array of (at least) length *nu*, containing long-precision real numbers.

nu
is the length of array AU.

Specified as: an integer; *nu* ≥ 0 and *nu* ≥ (`IDU`(*n*+1)-1).

idu
is the array, referred to as `IDU`, containing the relative positions of the diagonal elements of matrix ***A*** in input array AU.

Specified as: a one-dimensional array of (at least) length *n*+1, containing integers.

al
is the array, referred to as AL, containing the lower triangular part of general sparse matrix ***A***, stored as follows, where:

If `ITRAN(1)` = 0, ***A*** is stored in diagonal-out skyline storage mode.

If `ITRAN(1)` = 1, ***A*** is stored in profile-in skyline storage mode.

Note: Entries in AL for diagonal elements of ***A*** are assumed not to have meaningful values.

Specified as: a one-dimensional array of (at least) length *nl*, containing long-precision real numbers.

nl

is the length of array AL.

Specified as: an integer; $nl \ge 0$ and $nl \ge$ (IDL(n+1)-1).

idl

is the array, referred to as `IDL`, containing the relative positions of the diagonal elements of matrix ***A*** in input array AL.

Specified as: a one-dimensional array of (at least) length n+1, containing integers.

itran

is an array of parameters, `ITRAN`(*i*), where:

- `ITRAN(1)` indicates the input storage mode used for matrix ***A***. This determines the arrangement of data in arrays AU, `IDU`, AL, and `IDL` on input, where:

  If `ITRAN(1)` = 0, diagonal-out skyline storage mode is used.

  If `ITRAN(1)` = 1, profile-in skyline storage mode is used.
- `ITRAN(2)` indicates the output storage mode used for matrix ***A***. This determines the arrangement of data in arrays AU, `IDU`, AL, and `IDL` on output, where:

  If `ITRAN(2)` = 0, diagonal-out skyline storage mode is used.

  If `ITRAN(2)` = 1, profile-in skyline storage mode is used.
- `ITRAN(3)` indicates the direction of sweep that ESSL uses through the matrix ***A***, allowing you to optimize performance (see "Notes" on page 1217), where:

  If `ITRAN(3)` = 1, matrix ***A*** is transformed in the positive direction, starting in row or column 1 and ending in row or column *n*.

  If `ITRAN(3)` = -1, matrix ***A*** is transformed in the negative direction, starting in row or column *n* and ending in row or column 1.

Specified as: a one-dimensional array of (at least) length 3, containing integers, where:

`ITRAN(1)` = 0 or 1
`ITRAN(2)` = 0 or 1
`ITRAN(3)` = -1 or 1

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing *naux* long-precision real numbers.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, DGKTRN dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, it must have one of the following values:

For 32-bit integer arguments

$naux \ge 2n$

For 64-bit integer arguments
naux ≥ 4*n*

On Return

au

is the array, referred to as AU, containing the upper triangular part of general sparse matrix ***A***, stored as follows, where:

If `ITRAN(2)` = 0, ***A*** is stored in diagonal-out skyline storage mode.

If `ITRAN(2)` = 1, ***A*** is stored in profile-in skyline storage mode.

Returned as: a one-dimensional array of (at least) length *nu*, containing long-precision real numbers.

idu

is the array, referred to as `IDU`, containing the relative positions of the diagonal elements of matrix ***A*** in output array AU. Returned as: a one-dimensional array of (at least) length *n*+1, containing integers.

al

is the array, referred to as AL, containing the lower triangular part of general sparse matrix ***A***, stored as follows, where:

If `ITRAN(2)` = 0, ***A*** is stored in diagonal-out skyline storage mode.

If `ITRAN(2)` = 1, ***A*** is stored in profile-in skyline storage mode.

Note: You should assume that entries in AL for diagonal elements of ***A*** do not have meaningful values.

Returned as: a one-dimensional array of (at least) length *nl*, containing long-precision real numbers.

idl

is the array, referred to as `IDL`, containing the relative positions of the diagonal elements of matrix ***A*** in output array AL. Returned as: a one-dimensional array of (at least) length *n*+1, containing integers.

Notes

1. Your various arrays must have no common elements; otherwise, results are unpredictable.
2. The `ITRAN(3)` argument allows you to specify the direction of travel through matrix ***A*** that ESSL takes during the transformation. By properly specifying `ITRAN(3)`, you can optimize the performance of the transformation, which is especially beneficial when transforming large matrices.

   The direction specified by `ITRAN(3)` should be opposite the most recent direction of access through the matrix performed by the DGKFS or DGKFSP subroutine, as indicated in the following table:

Most Recent Computation Performed by DGKFS/ DGKFSP	Direction Used by DGKFS/ DGKFSP	Direction to Specify in ITRAN(3)
Factor and Solve	Negative	Positive (`ITRAN(3)` = 1)
Factor Only	Positive	Negative (`ITRAN(3)` = -1)
Solve Only	Negative	Positive (`ITRAN(3)` = 1)

3. For a description of how sparse matrices are stored in skyline storage mode, see "Profile-In Skyline Storage Mode" on page 119 and "Diagonal-Out Skyline Storage Mode" on page 117.
4. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

A general sparse matrix ***A***, stored in diagonal-out or profile-in skyline storage mode is converted to either of these same two storage modes. (Generally, you convert from one to the other, but the capability exists to specify the same storage mode for input and output.) The argument `ITRAN(3)` indicates the direction in which you want the transformation performed on matrix ***A***, allowing you to optimize your performance in this subroutine. This is especially beneficial for large matrices.

This subroutine is meant to be used in conjunction with DGKFS and DGKFSP, which process matrices stored in these skyline storage modes.

Error conditions

Resource Errors

Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors

None

Input-Argument Errors

1. *n* < 0
2. *nu* < 0
3. `IDU`(*n*+1) > *nu*+1
4. `IDU`(*i*+1) ≤ `IDU`(*i*) for *i* = 1, *n*
5. `IDU`(*i*+1) > `IDU`(*i*)+*i* and `ITRAN(1)` = 0 for *i* = 1, *n*
6. `IDU`(*i*) > `IDU`(*i*-1)+*i* and `ITRAN(1)` = 1 for *i* = 2, *n*
7. *nl* < 0
8. `IDL`(*n*+1) > *nl*+1
9. `IDL`(*i*+1) ≤ `IDL`(*i*) for *i* = 1, *n*
10. `IDL`(*i*+1) > `IDL`(*i*)+*i* and `ITRAN(1)` = 0 for *i* = 1, *n*
11. `IDL`(*i*) > `IDL`(*i*-1)+*i* and `ITRAN(1)` = 1 for *i* = 2, *n*
12. `ITRAN(1)` ≠ 0 or 1
13. `ITRAN(2)` ≠ 0 or 1
14. `ITRAN(3)` ≠ -1 or 1
15. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows how to convert a 9 by 9 general sparse matrix ***A*** from diagonal-out skyline storage mode to profile-in skyline storage mode. Matrix ***A*** is:

```
      | 11.0  12.0  13.0   0.0   0.0   0.0   0.0   0.0   0.0 |
      | 21.0  22.0  23.0  24.0  25.0   0.0   0.0   0.0  29.0 |
      | 31.0  32.0  33.0  34.0  35.0   0.0  37.0   0.0  39.0 |
      | 41.0  42.0  43.0  44.0  45.0  46.0  47.0   0.0  49.0 |
      |  0.0   0.0   0.0  54.0  55.0  56.0  57.0  58.0  59.0 |
      |  0.0  62.0  63.0  64.0  65.0  66.0  67.0  68.0  69.0 |
      |  0.0   0.0   0.0  74.0  75.0  76.0  77.0  78.0  79.0 |
      |  0.0   0.0   0.0  84.0  85.0  86.0  87.0  88.0  89.0 |
      | 91.0  92.0  93.0  94.0  95.0  96.0  97.0  98.0  99.0 |
```

Assuming that DGKFS last performed a solve on matrix ***A***, the direction of the transformation is positive; that is, `ITRAN(3)` is 1. This provides the best performance here.

Note: On input and output, the diagonal elements in AL do not have meaningful values.

Call Statement and Input:

```
             N   AU   NU   IDU   AL   NL   IDL   ITRAN   AUX   NAUX
             |   |    |     |    |    |     |      |      |     |
CALL DGKTRN( 9 , AU , 33 , IDU , AL , 35 , IDL , ITRAN , AUX ,  18
```

```
AU       =  (11.0, 22.0, 12.0, 33.0, 23.0, 13.0, 44.0, 34.0, 24.0,
             55.0, 45.0, 35.0, 25.0, 66.0, 56.0, 46.0, 77.0, 67.0,
             57.0, 47.0, 37.0, 88.0, 78.0, 68.0, 58.0, 99.0, 89.0,
             79.0, 69.0, 59.0, 49.0, 39.0, 29.0)
IDU      =  (1, 2, 4, 7, 10, 14, 17, 22, 26, 34)
AL       =  ( . , . , 21.0, . , 32.0, 31.0, . , 43.0, 42.0, 41.0, . ,
             54.0, . , 65.0, 64.0, 63.0, 62.0, . , 76.0, 75.0, 74.0,
             . , 87.0, 86.0, 85.0, 84.0, . , 98.0, 97.0, 96.0, 95.0,
             94.0, 93.0, 92.0, 91.0)
IDL      =  (1, 2, 4, 7, 11, 13, 18, 22, 27, 36)
ITRAN    =  (0, 1, 1)
```

Output:

```
AU       =  (11.0, 12.0, 22.0, 13.0, 23.0, 33.0, 24.0, 34.0, 44.0,
             25.0, 35.0, 45.0, 55.0, 46.0, 56.0, 66.0, 37.0, 47.0,
             57.0, 67.0, 77.0, 58.0, 68.0, 78.0, 88.0, 29.0, 39.0,
             49.0, 59.0, 69.0, 79.0, 89.0, 99.0)
IDU      =  (1, 3, 6, 9, 13, 16, 21, 25, 33, 34)
AL       =  ( . , 21.0, . , 31.0, 32.0, . , 41.0, 42.0, 43.0, . , 54.0,
             . , 62.0, 63.0, 64.0, 65.0, . , 74.0, 75.0, 76.0, . ,
             84.0, 85.0, 86.0, 87.0, . , 91.0, 92.0, 93.0, 94.0, 95.0,
             96.0, 97.0, 98.0, . )
IDL      =  (1, 3, 6, 10, 12, 17, 21, 26, 35, 36)
```

Example 2

This example shows how to convert the same 9 by 9 general sparse matrix ***A*** in Example 1 from profile-in skyline storage mode to diagonal-out skyline storage mode.

Assuming that DGKFS last performed a factorization on matrix ***A***, the direction of the transformation is negative; that is, `ITRAN(3)` is -1. This provides the best performance here.

Note: On input and output, the diagonal elements in AL do not have meaningful values.

Call Statement and Input:

```
             N   AU   NU   IDU   AL   NL   IDL   ITRAN   AUX   NAUX
             |   |    |     |    |    |     |      |      |     |
CALL DGKTRN( 9 , AU , 33 , IDU , AL , 35 , IDL , ITRAN , AUX ,  18
```

`AU` =(same as output AU in Example 1)
`IDU` =(same as output `IDU` in Example 1)
`AL` =(same as output AL in Example 1)
`IDL` =(same as output `IDL` in Example 1)
`ITRAN = (1, 0, -1)`

Output:

`AU` =(same as input AU in Example 1)
`IDU` =(same as input `IDU` in Example 1)
`AL` =(same as input AL in Example 1)
`IDL` =(same as input `IDL` in Example 1)

DSKTRN (For a Symmetric Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode)

Purpose

This subroutine converts symmetric sparse matrix ***A*** of order *n* from one skyline storage mode to another—that is, between the following:

- Diagonal-out skyline storage mode
- Profile-in skyline storage mode

Syntax

Language	Syntax
Fortran	CALL DSKTRN (*n*, *a*, *na*, *idiag*, *itran*, *aux*, *naux*)
C and C++	dsktrn (*n*, *a*, *na*, *idiag*, *itran*, *aux*, *naux*);

On Entry

n

is the order of symmetric sparse matrix ***A***. Specified as: an integer; *n* ≥ 0.

a

is the array, referred to as A, containing the upper triangular part of symmetric sparse matrix ***A***, stored as follows, where:

If `ITRAN(1)` = 0, ***A*** is stored in diagonal-out skyline storage mode.

If `ITRAN(1)` = 1, ***A*** is stored in profile-in skyline storage mode.

Specified as: a one-dimensional array of (at least) length *na*, containing long-precision real numbers.

na

is the length of array A.

Specified as: an integer; *na* ≥ 0 and *na* ≥ (`IDIAG`(*n*+1)-1).

idiag

is the array, referred to as `IDIAG`, containing the relative positions of the diagonal elements of matrix ***A*** in input array A.

Specified as: a one-dimensional array of (at least) length *n*+1, containing integers.

itran

is an array of parameters, `ITRAN`(*i*), where:

- `ITRAN(1)` indicates the input storage mode used for matrix ***A***. This determines the arrangement of data in arrays A and `IDIAG` on input, where:

  If `ITRAN(1)` = 0, diagonal-out skyline storage mode is used.

  If `ITRAN(1)` = 1, profile-in skyline storage mode is used.

- `ITRAN(2)` indicates the output storage mode used for matrix ***A***. This determines the arrangement of data in arrays A and `IDAIG` on output, where:

  If `ITRAN(2)` = 0, diagonal-out skyline storage mode is used.

  If `ITRAN(2)` = 1, profile-in skyline storage mode is used.

- `ITRAN(3)` indicates the direction of sweep that ESSL uses through the matrix ***A***, allowing you to optimize performance (see "Notes" on page 1221), where:

  If `ITRAN(3)` = 1, matrix ***A*** is transformed in the positive direction, starting in row or column 1 and ending in row or column *n*.

  If `ITRAN(3)` = -1, matrix ***A*** is transformed in the negative direction, starting in row or column *n* and ending in row or column 1.

Specified as: a one-dimensional array of (at least) length 3, containing integers, where:

`ITRAN(1)` = 0 or 1
`ITRAN(2)` = 0 or 1
`ITRAN(3)` = -1 or 1

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing *naux* long-precision real numbers.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: an integer, where:

If *naux* = 0 and error 2015 is unrecoverable, DSKTRN dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise

For 32-bit integer arguments
naux ≥ *n*

For 64-bit integer arguments
naux ≥ 2*n*

On Return

a

is the array, referred to as A, containing the upper triangular part of symmetric sparse matrix ***A***, stored as follows, where:

If `ITRAN(2)` = 0, ***A*** is stored in diagonal-out skyline storage mode.

If `ITRAN(2)` = 1, ***A*** is stored in profile-in skyline storage mode.

Returned as: a one-dimensional array of (at least) length *na*, containing long-precision real numbers.

idiag

is the array, referred to as `IDIAG`, containing the relative positions of the diagonal elements of matrix ***A*** in output array A. Returned as: a one-dimensional array of (at least) length *n*+1, containing integers.

Notes

1. Your various arrays must have no common elements; otherwise, results are unpredictable.
2. The `ITRAN(3)` argument allows you to specify the direction of travel through matrix ***A*** that ESSL takes during the transformation. By properly specifying `ITRAN(3)`, you can optimize the performance of the transformation, which is especially beneficial when transforming large matrices.

   The direction specified by `ITRAN(3)` should be opposite the most recent direction of access through the matrix performed by the DSKFS or DSKFSP subroutine, as indicated in the following table:

Most Recent Computation Performed by DSKFS/ DSKFSP	Direction Used by DSKFS/ DSKFSP	Direction to Specify in ITRAN(3)
Factor and Solve	Negative	Positive (`ITRAN(3)` = 1)
Factor Only	Positive	Negative (`ITRAN(3)` = -1)
Solve Only	Negative	Positive (`ITRAN(3)` = 1)

3. For a description of how sparse matrices are stored in skyline storage mode, see "Profile-In Skyline Storage Mode" on page 119 and "Diagonal-Out Skyline Storage Mode" on page 117.
4. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 49.

Function

A symmetric sparse matrix ***A***, stored in diagonal-out or profile-in skyline storage mode is converted to either of these same two storage modes. (Generally, you convert from one to the other, but the capability exists to specify the same storage mode for input and output.) The argument `ITRAN(3)` indicates the direction in which you want the transformation performed on matrix ***A***, allowing you to optimize your performance in this subroutine. This is especially beneficial for large matrices.

This subroutine is meant to be used in conjunction with DSKFS and DSKFSP, which process matrices stored in these skyline storage modes.

Error conditions

Resource Errors
Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors
None

Input-Argument Errors

1. *n* < 0
2. *na* < 0
3. `IDIAG`(*n*+1) > *na*+1
4. `IDIAG`(*i*+1) ≤ `IDIAG`(*i*) for *i* = 1, *n*
5. `IDIAG`(*i*+1) > `IDIAG`(*i*)+*i* and `ITRAN(1)` = 0 for *i* = 1, *n*
6. `IDIAG`(*i*) > `IDIAG`(*i*-1)+*i* and `ITRAN(1)` = 1 for *i* = 2, *n*
7. `ITRAN(1)` ≠ 0 or 1
8. `ITRAN(2)` ≠ 0 or 1
9. `ITRAN(3)` ≠ -1 or 1
10. *naux* Error 2015 is recoverable or *naux*≠0, and is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows how to convert a 9 by 9 symmetric sparse matrix ***A*** from diagonal-out skyline storage mode to profile-in skyline storage mode. Matrix ***A*** is:

```
┌                                                    ┐
| 11.0  12.0  13.0  14.0   0.0   0.0   0.0   0.0   0.0 |
| 12.0  22.0  23.0  24.0  25.0  26.0   0.0  28.0   0.0 |
| 13.0  23.0  33.0  34.0  35.0  36.0   0.0  38.0   0.0 |
| 14.0  24.0  34.0  44.0  45.0  46.0   0.0  48.0   0.0 |
|  0.0  25.0  35.0  45.0  55.0  56.0  57.0  58.0   0.0 |
|  0.0  26.0  36.0  46.0  56.0  66.0  67.0  68.0  69.0 |
|  0.0   0.0   0.0   0.0  57.0  67.0  77.0  78.0  79.0 |
|  0.0  28.0  38.0  48.0  58.0  68.0  78.0  88.0  89.0 |
|  0.0   0.0   0.0   0.0   0.0  69.0  79.0  89.0  99.0 |
└                                                    ┘
```

Assuming that DSKFS last performed a factorization on matrix ***A***, the direction of the transformation is negative; that is, `ITRAN(3)` is -1. This provides the best performance here.

Call Statement and Input:

```
             N   A   NA   IDIAG   ITRAN   AUX   NAUX
             |   |   |      |       |      |     |
CALL DSKTRN( 9 , A , 33 , IDIAG , ITRAN , AUX ,  9  )
```

```
A        =  (11.0, 22.0, 12.0, 33.0, 23.0, 13.0, 44.0, 34.0, 24.0,
             14.0, 55.0, 45.0, 35.0, 25.0, 66.0, 56.0, 46.0, 36.0,
             26.0, 77.0, 67.0, 57.0, 88.0, 78.0, 68.0, 58.0, 48.0,
             38.0, 28.0, 99.0, 89.0, 79.0, 69.0)
IDIAG    =  (1, 2, 4, 7, 11, 15, 20, 23, 30, 34)
ITRAN    =  (0, 1, -1)
```

Output:

```
A        =  (11.0, 12.0, 22.0, 13.0, 23.0, 33.0, 14.0, 24.0, 34.0,
             44.0, 25.0, 35.0, 45.0, 55.0, 26.0, 36.0, 46.0, 56.0,
             66.0, 57.0, 67.0, 77.0, 28.0, 38.0, 48.0, 58.0, 68.0,
             78.0, 88.0, 69.0, 79.0, 89.0, 99.0)
IDIAG    =  (1, 3, 6, 10, 14, 19, 22, 29, 33, 34)
```

Example 2

This example shows how to convert the same 9 by 9 symmetric sparse matrix ***A*** in Example 1 from profile-in skyline storage mode to diagonal-out skyline storage mode.

Assuming that DSKFS last performed a solve on matrix ***A***, the direction of the transformation is positive; that is, `ITRAN(3)` is 1. This provides the best performance here.

Call Statement and Input:

```
             N   A   NA   IDIAG   ITRAN   AUX   NAUX
             |   |   |      |       |      |     |
CALL DSKTRN( 9 , A , 33 , IDIAG , ITRAN , AUX ,  9
```

`A` =(same as output A in Example 1)
`IDIAG` =(same as output `IDIAG` in Example 1)
`ITRAN = (1, 0, 1)`

Output:

`A` =(same as input A in Example 1)
`IDIAG` =(same as input `IDIAG` in Example 1)

Appendix A. Basic Linear Algebra Subprograms (BLAS) and C BLAS (CBLAS)

The following information lists the ESSL subprograms corresponding to a subprogram in the standard set of BLAS and CBLAS.

Level 1 BLAS

Table 269: Level 1 BLAS Included in ESSL.

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram
"ISAMAX, IDAMAX, ICAMAX, and IZAMAX (Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude)" on page 204	ISAMAX ICAMAX cblas_isamax cblas_idamax	IDAMAX IZAMAX cblas_icamax cblas_izamax
"SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector)" on page 214	SASUM SCASUM cblas_sasum cblas_scasum	DASUM DZASUM cblas_dasum cblas_dcasum
"SAXPY, DAXPY, CAXPY, and ZAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y)" on page 216	SAXPY CAXPY cblas_saxpy cblas_caxpy	DAXPY ZAXPY cblas_daxpy cblas_zaxpy
"SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)" on page 223	SCOPY CCOPY cblas_scopy cblas_ccopy	DCOPY ZCOPY cblas_dcopy cblas_zcopy
"SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDOTC (Dot Product of Two Vectors)" on page 225	SDOT CDOTU CDOTC cblas_sdot cblas_cdotu_sub cblas_cdotc_sub	DDOT ZDOTU ZDOTC cblas_ddot cblas_zdotu_sub cblas_zdotc_sub
"SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow)" on page 238	SNRM2 SCNRM2 cblas_snrm2 cblas_scnrm2	DNRM2 DZNRM2 cblas_dnrm2 cblas_dznrm2
"SROTG, DROTG, CROTG, and ZROTG (Construct a Givens Plane Rotation)" on page 244	SROTG CROTG cblas_srotg cblas_crotg	DROTG ZROTG cblas_drotg cblas_zrotg

Table 269: Level 1 BLAS Included in ESSL. (continued)

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram
"SROT, DROT, CROT, ZROT, CSROT, and ZDROT (Apply a Plane Rotation)" on page 250	SROT CROT CSROT cblas_srot cblas_crot cblas_csrot	DROT ZROT ZDROT cblas_drot cblas_zrot cblas_zdrot
"SROTMG and DROTMG (Construct a modified Givens Transformation)" on page 254	SROTMG cblas_srotmg	DROTMG cblas_drotmg
"SROTM and DROTM (Apply a modified Givens Transformation)" on page 256	SROTM cblas_srotm	DROTM cblas_drotm
"SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, and ZDSCAL (Multiply a Vector X by a Scalar and Store in the Vector X)" on page 259	SSCAL CSCAL CSSCAL cblas_sscal cblas_cscal cblas_csscal	DSCAL ZSCAL ZDSCAL cblas_dscal cblas_zscal cblas_zdscal
"SSWAP, DSWAP, CSWAP, and ZSWAP (Interchange the Elements of Two Vectors)" on page 262	SSWAP CSWAP cblas_sswap cblas_cswap	DSWAP ZSWAP cblas_dswap cblas_zswap

Level 2 BLAS

Table 270: Level 2 BLAS Included in ESSL.

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram
"SGEMV, DGEMV, CGEMV, ZGEMV, SGEMX, DGEMX, SGEMTX, and DGEMTX (Matrix-Vector Product for a General Matrix, Its Transpose, or Its Conjugate Transpose)" on page 295	SGEMV CGEMV cblas_sgemv cblas_cgemv	DGEMV ZGEMV cblas_dgemv cblas_zgemv
"SGER, DGER, CGERU, ZGERU, CGERC, and ZGERC (Rank-One Update of a General Matrix)" on page 306	SGER CGERU CGERC cblas_sger cblas_cgeru cblas_cgerc	DGER ZGERU ZGERC cblas_dger cblas_zgeru cblas_zgerc

Table 270: Level 2 BLAS Included in ESSL. (continued)

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram
"SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)" on page 313	SSPMV CHPMV SSYMV CHEMV cblas_sspmv cblas_chpmv cblas_ssymv cblas_chemv	DSPMV ZHPMV DSYMV ZHEMV cblas_dspmv cblas_zhpmv cblas_dsymv cblas_zhemv
"SSPR, DSPR, CHPR, ZHPR, SSYR, DSYR, CHER, ZHER, SSLR1, and DSLR1 (Rank-One Update of a Real Symmetric or Complex Hermitian Matrix)" on page 321	SSPR CHPR SSYR CHER cblas_sspr cblas_chpr cblas_ssyr cblas_cher	DSPR ZHPR DSYR ZHER cblas_dspr cblas_zhpr cblas_dsyr cblas_zher
"SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, ZHER2, SSLR2, and DSLR2 (Rank-Two Update of a Real Symmetric or Complex Hermitian Matrix) " on page 328	SSPR2 CHPR2 SSYR2 CHER2 cblas_sspr2 cblas_chpr2 cblas_ssyr2 cblas_cher2	DSPR2 ZHPR2 DSYR2 ZHER2 cblas_dspr2 cblas_zhpr2 cblas_dsyr2 cblas_zher2
"SGBMV, DGBMV, CGBMV, and ZGBMV (Matrix-Vector Product for a General Band Matrix, Its Transpose, or Its Conjugate Transpose)" on page 336	SGBMV CGBMV cblas_sgbmv cblas_cgbmv	DGBMV ZGBMV cblas_dgbmv cblas_zgbmv
"SSBMV, DSBMV, CHBMV, and ZHBMV (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Band Matrix)" on page 343	SSBMV CHBMV cblas_ssbmv cblas_chbmv	DSBMV ZHBMV cblas_dsbmv cblas_zhbmv
"STRMV, DTRMV, CTRMV, ZTRMV, STPMV, DTPMV, CTPMV, and ZTPMV (Matrix-Vector Product for a Triangular Matrix, Its Transpose, or Its Conjugate Transpose)" on page 347	STPMV CTPMV STRMV CTRMV cblas_stpmv cblas_ctpmv cblas_strmv cblas_ctrmv	DTPMV ZTPMV DTRMV ZTRMV cblas_dtpmv cblas_ztpmv cblas_dtrmv cblas_ztrmv

Table 270: Level 2 BLAS Included in ESSL. (continued)

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram
"STRSV, DTRSV, CTRSV, ZTRSV, STPSV, DTPSV, CTPSV, and ZTPSV (Solution of a Triangular System of Equations with a Single Right-Hand Side)" on page 354	STPSV CTPSV STRSV CTRSV cblas_stpsv cblas_ctpsv cblas_strsv cblas_ctrsv	DTPSV ZTPSV DTRSV ZTRSV cblas_dtpsv cblas_ztpsv cblas_dtrsv cblas_ztrsv
"STBMV, DTBMV, CTBMV, and ZTBMV (Matrix-Vector Product for a Triangular Band Matrix, Its Transpose, or Its Conjugate Transpose)" on page 360	STBMV CTBMV cblas_stbmv cblas_ctbmv	DTBMV ZTBMV cblas_dtbmv cblas_ztbmv
"STBSV, DTBSV, CTBSV, and ZTBSV (Triangular Band Equation Solve)" on page 365	STBSV CTBSV cblas_stbsv cblas_ctbsv	DTBSV ZTBSV cblas_dtbsv cblas_ztbsv

Level 3 BLAS

Table 271: Level 3 BLAS Included in ESSL.

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram
"SGEMM, DGEMM, CGEMM, and ZGEMM (Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes)" on page 413	SGEMM CGEMM cblas_sgemm cblas_cgemm	DGEMM ZGEMM cblas_dgemm cblas_zgemm
"SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM (Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian)" on page 422	SSYMM CSYMM CHEMM cblas_ssymm cblas_csymm cblas_chemm	DSYMM ZSYMM ZHEMM cblas_dsymm cblas_zsymm cblas_zhemm
"STRMM, DTRMM, CTRMM, and ZTRMM (Triangular Matrix-Matrix Product)" on page 430	STRMM CTRMM cblas_strmm cblas_ctrmm	DTRMM ZTRMM cblas_dtrmm cblas_ztrmm
"STRSM, DTRSM, CTRSM, and ZTRSM (Solution of Triangular Systems of Equations with Multiple Right-Hand Sides)" on page 437	STRSM CTRSM cblas_strsm cblas_ctrsm	DTRSM ZTRSM cblas_dtrsm cblas_ztrsm

Table 271: Level 3 BLAS Included in ESSL. (continued)

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram
"SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK (Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)" on page 445	SSYRK CSYRK CHERK cblas_ssyrk cblas_csyrk cblas_cherk	DSYRK ZSYRK ZHERK cblas_dsyrk cblas_zsyrk cblas_zherk
"SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K (Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)" on page 451	SSYR2K CSYR2K CHER2K cblas_ssyr2k cblas_csyr2k cblas_cher2k	DSYR2K ZSYR2K ZHER2K cblas_dsyr2k cblas_zsyr2k cblas_zher2k

Note: xerbla is not used by ESSL but is included in the ESSL libraries for use in other programming environments (for example, Python).

CBLAS Subroutines

Note the following about the CBLAS subroutines:

- The definitions of the CBLAS data types and prototypes are included in the ESSL header files. In the beginning of your program, before you call any of the CBLAS subroutines, you must code the following statement for the ESSL header file:

```
#include <essl.h>
```

- CBLAS enumerated types have changed to type definitions for ESSL Version 5 Release 5 and changes are required if you used enumerated types. See "Migrating Programs from ESSL for Linux on Power Version 5 Release 4 to Version 5 Release 5" on page 179 for the required changes to your existing programs
- For information about CBLAS calling sequences, see "Syntax" on page xxxi and ["11" on page 1253].

Appendix B. LAPACK and LAPACKE

The following table lists the ESSL subroutines corresponding to subroutines in the standard set of LAPACK and LAPACKE.

Table 272: LAPACK and LAPACKE subroutines included in ESSL.

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
SGESV CGESV LAPACKE_sgesv LAPACKE_cgesv	DGESV ZGESV LAPACKE_dgesv LAPACKE_zgesv	"SGESV, DGESV, CGESV, ZGESV (General Matrix Factorization and Multiple Right-Hand Side Solve)" on page 480
SGETRF CGETRF LAPACKE_sgetrf LAPACKE_cgetrf	DGETRF ZGETRF LAPACKE_dgetrf LAPACKE_zgetrf	"SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)" on page 484
SGETRS CGETRS LAPACKE_sgetrs LAPACKE_cgetrs	DGETRS ZGETRS LAPACKE_dgetrs LAPACKE_zgetrs	"SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)" on page 488
SGECON CGECON LAPACKE_sgecon LAPACKE_cgecon	DGECON ZGECON LAPACKE_dgecon LAPACKE_zgecon	"SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)" on page 504
SGETRI CGETRI LAPACKE_sgetri LAPACKE_cgetri	DGETRI ZGETRI LAPACKE_dgetri LAPACKE_zgetri	"SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)" on page 511
SLANGE CLANGE LAPACKE_slange LAPACKE_clange	DLANGE ZLANGE LAPACKE_dlange LAPACKE_zlange	"SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)" on page 518
SPPSV CPPSV LAPACKE_sppsv LAPACKE_cppsv	DPPSV ZPPSV LAPACKE_dppsv LAPACKE_zppsv	"SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)" on page 521
SPOSV CPOSV LAPACKE_sposv LAPACKE_cposv	DPOSV ZPOSV LAPACKE_dposv LAPACKE_zposv	"SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)" on page 527

Table 272: LAPACK and LAPACKE subroutines included in ESSL. (continued)

Short-Precision Subprogram	Long-Precision Subprogram	Descriptive Name and Location
SPOTRF CPOTRF SPPTRF CPPTRF LAPACKE_spotrf LAPACKE_cpotrf LAPACKE_spptrf LAPACKE_cpptrf	DPOTRF ZPOTRF DPPTRF ZPPTRF LAPACKE_dpotrf LAPACKE_zpotrf LAPACKE_dpptrf LAPACKE_zpptrf	"SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)" on page 533
SPOTRS CPOTRS SPPTRS CPPTRS LAPACKE_spotrs LAPACKE_cpotrs LAPACKE_spptrs LAPACKE_cpptrs	DPOTRS ZPOTRS DPPTRS ZPPTRS LAPACKE_dpotrs LAPACKE_zpotrs LAPACKE_dpptrs LAPACKE_zpptrs	"SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS, CPPTRS, and ZPPTRS (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)" on page 545
SPOCON CPOCON SPPCON CPPCON LAPACKE_spocon LAPACKE_cpocon LAPACKE_sppcon LAPACKE_cppcon	DPOCON ZPOCON DPPCON ZPPCON LAPACKE_dpocon LAPACKE_zpocon LAPACKE_dppcon LAPACKE_zppcon	"SPOCON, DPOCON, CPOCON, ZPOCON, SPPCON, DPPCON, CPPCON, and ZPPCON (Estimate the Reciprocal of the Condition Number of a Positive Definite Real Symmetric or Complex Hermitian Matrix)" on page 555
SPOTRI CPOTRI SPPTRI CPPTRI LAPACKE_spotri LAPACKE_cpotri LAPACKE_spptri LAPACKE_cpptri	DPOTRI ZPOTRI DPPTRI ZPPTRI LAPACKE_dpotri LAPACKE_zpotri LAPACKE_dpptri LAPACKE_zpptri	"SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPPTRI, DPPTRI, CPPTRI, ZPPTRI, SPPICD, and DPPICD (Positive Definite Real Symmetric or Complex Hermitian Matrix Inverse, Condition Number Reciprocal, and Determinant)" on page 569
SLANSY >|CLANSY|< CLANHE SLANSP CLANHP LAPACKE_slansy >|LAPACKE_clansy |< LAPACKE_clanhe LAPACKE_slansp LAPACKE_clanhp	DLANSY >|ZLANSY|< ZLANHE DLANSP ZLANHP LAPACKE_dlansy >|LAPACKE_zlansy |< LAPACKE_zlanhe LAPACKE_dlansp LAPACKE_zlanhp	"SLANSY, DLANSY, CLANSY, ZLANSY, CLANHE, ZLANHE, SLANSP, DLANSP, CLANHP, and ZLANHP (Real Symmetric or Complex Hermitian Matrix Norm)" on page 579

Table 272: LAPACK and LAPACKE subroutines included in ESSL. (continued)

Short-Precision Subprogram	Long-Precision Subprogram	Descriptive Name and Location
SSYSV CSYSV CHESV SSPSV CSPSV CHPSV LAPACKE_ssysv LAPACKE_csysv LAPACKE_chesv LAPACKE_sspsv LAPACKE_cspsv LAPACKE_chpsv	DSYSV ZSYSV ZHESV DSPSV ZSPSV ZHPSV LAPACKE_dsysv LAPACKE_zsysv LAPACKE_zhesv LAPACKE_dspsv LAPACKE_zspsv LAPACKE_zhpsv	"SSYSV, DSYSV, CSYSV, ZSYSV, CHESV, ZHESV, SSPSV, DSPSV, CSPSV, ZSPSV, CHPSV, and ZHPSV (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)" on page 584
SSYTRF CSYTRF CHETRF SSPTRF CSPTRF CHPTRF LAPACKE_ssytrf LAPACKE_csytrf LAPACKE_chetrf LAPACKE_ssptrf LAPACKE_csptrf LAPACKE_chptrf	DSYTRF ZSYTRF ZHETRF DSPTRF ZSPTRF ZHPTRF LAPACKE_dsytrf LAPACKE_zsytrf LAPACKE_zhetrf LAPACKE_dsptrf LAPACKE_zsptrf LAPACKE_zhptrf	"SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, and ZHPTRF (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization)" on page 593
SSYTRS CSYTRS CHETRS SSPTRS CSPTRS CHPTRS LAPACKE_ssytrs LAPACKE_csytrs LAPACKE_chetrs LAPACKE_ssptrs LAPACKE_csptrs LAPACKE_chptrs	DSYTRS ZSYTRS ZHETRS DSPTRS ZSPTRS ZHPTRS LAPACKE_dsytrs LAPACKE_zsytrs LAPACKE_zhetrs LAPACKE_dsptrs LAPACKE_zsptrs LAPACKE_zhptrs	"SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, ZHETRS, SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, and ZHPTRS (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)" on page 601
STRTRI STPTRI CTRTRI CTPTRI LAPACKE_strtri LAPACKE_stptri LAPACKE_ctrtri LAPACKE_ctptri	DTRTRI DTPTRI ZTRTRI ZTPTRI LAPACKE_dtrtri LAPACKE_dtptri LAPACKE_ztrtri LAPACKE_ztptri	"STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)" on page 621

Table 272: LAPACK and LAPACKE subroutines included in ESSL. (continued)

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
SLANTR CLANTR SLANTP CLANTP LAPACKE_slantr LAPACKE_clantr LAPACKE_slantp LAPACKE_clantp	DLANTR ZLANTR DLANTP ZLANTP LAPACKE_dlantr LAPACKE_zlantr LAPACKE_dlantp LAPACKE_zlantp	"SLANTR, DLANTR, CLANTR, ZLANTR, SLANTP, DLANTP, CLANTP, and ZLANTP (Trapezoidal or Triangular Matrix Norm)" on page 628
SGBSV CGBSV LAPACKE_sgbsv LAPACKE_cgbsv	DGBSV ZGBSV LAPACKE_dgbsv LAPACKE_zgbsv	"SGBSV, DGBSV, CGBSV, and ZGBSV (General Band Matrix Factorization and Multiple Right-Hand Side Solve)" on page 635
SGBTRF CGBTRF LAPACKE_sgbtrf LAPACKE_cgbtrf	DGBTRF ZGBTRF LAPACKE_dgbtrf LAPACKE_zgbtrf	"SGBTRF, DGBTRF, CGBTRF and ZGBTRF (General Band Matrix Factorization)" on page 639
SGBTRS CGBTRS LAPACKE_sgbtrs LAPACKE_cgbtrs	DGBTRS ZGBTRS LAPACKE_dgbtrs LAPACKE_zgbtrs	"SGBTRS, DGBTRS, CGBTRS, and ZGBTRS (General Band Matrix Multiple Right-Hand Side Solve)" on page 643
SPBSV CPBSV LAPACKE_spbsv LAPACKE_cpbsv	DPBSV ZPBSV LAPACKE_dpbsv LAPACKE_zpbsv	"SPBSV, DPBSV, CPBSV, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization and Multiple Right-Hand Side Solve)" on page 651
SPBTRF CPBTRF LAPACKE_spbtrf LAPACKE_cpbtrf	DPBTRF ZPBTRF LAPACKE_dpbtrf LAPACKE_zpbtrf	"SPBTRF, DPBTRF, CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)" on page 656
SPBTRS CPBTRS LAPACKE_spbtrs LAPACKE_cpbtrs	DPBTRS ZPBTRS LAPACKE_dpbtrs LAPACKE_zpbtrs	"SPBTRS, DPBTRS, CPBTRS, and ZPBTRS (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Multiple Right-Hand Side Solve)" on page 661
SGTSV CGTSV LAPACKE_sgtsv LAPACKE_cgtsv	DGTSV ZGTSV LAPACKE_dgtsv LAPACKE_zgtsv	"SGTSV, DGTSV, CGTSV, and ZGTSV (General Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)" on page 666
SGTTRF CGTTRF LAPACKE_sgttrf LAPACKE_cgttrf	DGTTRF ZGTTRF LAPACKE_dgttrf LAPACKE_zgttrf	"SGTTRF, DGTTRF, CGTTRF, and ZGTTRF (General Tridiagonal Matrix Factorization)" on page 670

Table 272: LAPACK and LAPACKE subroutines included in ESSL. (continued)

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
SGTTRS CGTTRS LAPACKE_sgttrs LAPACKE_cgttrs	DGTTRS ZGTTRS LAPACKE_dgttrs LAPACKE_zgttrs	"SGTTRS, DGTTRS, CGTTRS, and ZGTTRS (General Tridiagonal Matrix Multiple Right-Hand Side Solve)" on page 674
SPTSV CPTSV LAPACKE_sptsv LAPACKE_cptsv	DPTSV ZPTSV LAPACKE_dptsv LAPACKE_zptsv	"SPTSV, DPTSV, CPTSV, and ZPTSV (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)" on page 679
SPTTRF CPTTRF LAPACKE_spttrf LAPACKE_cpttrf	DPTTRF ZPTTRF LAPACKE_dpttrf LAPACKE_zpttrf	"SPTTRF, DPTTRF, CPTTRF, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)" on page 684
SPTTRS CPTTRS LAPACKE_spttrs LAPACKE_cpttrs	DPTTRS ZPTTRS LAPACKE_dpttrs LAPACKE_zpttrs	"SPTTRS, DPTTRS, CPTTRS, and ZPTTRS (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Multiple Right-Hand Solve)" on page 687
SGESVD CGESVD SGESDD CGESDD LAPACKE_sgesvd LAPACKE_cgesvd LAPACKE_sgesdd LAPACKE_cgesdd	DGESVD ZGESVD DGESDD ZGESDD LAPACKE_dgesvd LAPACKE_zgesvd LAPACKE_dgesdd LAPACKE_zgesdd	"SGESVD, DGESVD, CGESVD, ZGESVD, SGESDD, DGESDD, CGESDD, and ZGESDD (Singular Value Decomposition for a General Matrix)" on page 794
SGEQRF CGEQRF LAPACKE_sgeqrf LAPACKE_cgeqrf	DGEQRF ZGEQRF LAPACKE_dgeqrf LAPACKE_zgeqrf	"SGEQRF, DGEQRF, CGEQRF, and ZGEQRF (General Matrix QR Factorization)" on page 807
SGELS CGELS LAPACKE_sgels LAPACKE_cgels	DGELS ZGELS LAPACKE_dgels LAPACKE_zgels	"SGELS, DGELS, CGELS, and ZGELS (Linear Least Squares Solution for a General Matrix)" on page 812
SGELSD CGELSD LAPACKE_sgelsd LAPACKE_cgelsd	DGELSD ZGELSD LAPACKE_dgelsd LAPACKE_zgelsd	"SGELSD, DGELSD, CGELSD, and ZGELSD (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)" on page 822

Table 272: LAPACK and LAPACKE subroutines included in ESSL. (continued)

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
SGEEV CGEEV SGEEVX CGEEVX LAPACKE_sgeev LAPACKE_cgeev LAPACKE_sgeevx LAPACKE_cgeevx	DGEEV ZGEEV DGEEVX ZGEEVX LAPACKE_dgeev LAPACKE_zgeev LAPACKE_dgeevx LAPACKE_zgeevx	"SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)" on page 850
SSYEV CHEEV SSPEVX CHPEVX SSYEVX CHEEVX >|SSYEVR CHEEVR|< LAPACKE_ssyev LAPACKE_cheev LAPACKE_sspevx LAPACKE_chpevx LAPACKE_ssyevx LAPACKE_cheevx >|LAPACKE_ssyevr LAPACKE_cheevr|<	DYSEV ZHEEV DSPEVX ZHPEVX DSYEVX ZHEEVX >|DSYEVR ZHEEVR|< LAPACKE_dsyev LAPACKE_zheev LAPACKE_dspevx LAPACKE_zhpevx LAPACKE_dsyevx LAPACKE_zheevx >|LAPACKE_dsyevr LAPACKE_zheevr|<	"SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, ZHEEVX, SSYEVR, DSYEVR, CHEEVR, and ZHEEVR (Eigenvalues, and Optionally, the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix) " on page 867
SSPEVD CHPEVD SSYEVD CHEEVD LAPACKE_sspevd LAPACKE_chpevd LAPACKE_ssyevd LAPACKE_cheevd	DSPEVD ZHPEVD DSYEVD ZHEEVD LAPACKE_dspevd LAPACKE_zhpevd LAPACKE_dsyevd LAPACKE_zheevd	"SSPEVD, DSPEVD, CHPEVD, ZHPEVD, SSYEVD, DSYEVD, CHEEVD, and ZHEEVD (Eigenvalues and, Optionally the Eigenvectors, of a Real Symmetric or Complex Hermitian Matrix Using a Divide-and-Conquer Algorithm) " on page 893
SGGEV CGGEV SGGEVX CGGEVX LAPACKE_sggev LAPACKE_cggev LAPACKE_sggevx LAPACKE_cggevx	DGGEV ZGGEV DGGEVX ZGGEVX LAPACKE_dggev LAPACKE_zggev LAPACKE_dggevx LAPACKE_zggevx	"SGGEV, DGGEV, CGGEV, ZGGEV, SGGEVX, DGGEVX, CGGEVX, and ZGGEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix Generalized Eigenproblem)" on page 905

Table 272: LAPACK and LAPACKE subroutines included in ESSL. (continued)

Short-Precision Subprogram	**Long-Precision Subprogram**	**Descriptive Name and Location**
SSYGVD CHEGVD LAPACKE_ssygvd LAPACKE_chegvd	DSYGVD ZHEGVD LAPACKE_dsygvd LAPACKE_zhegvd	"SSYGVD, DSYGVD, CHEGVD, ZHEGVD (Eigenvalues, and Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem Using a Divide-and-Conquer Algorithm)" on page 922
SSYGV CHEGV SSPGVX CHPGVX SSYGVX CHEGVX LAPACKE_ssygv LAPACKE_chegv LAPACKE_sspgvx LAPACKE_chpgvx LAPACKE_ssygvx LAPACKE_chegvx	DSYGV ZHEGV DSPGVX ZHPGVX DSYGVX ZHEGVX LAPACKE_dsygv LAPACKE_zhegv LAPACKE_dspgvx LAPACKE_zhpgvx LAPACKE_dsygvx LAPACKE_zhegvx	"SSYGV, DSYGV, CHEGV, ZHEGV, SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX (Eigenvalues, and Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem)" on page 931
SLAMCH LAPACKE_slamch	DLAMCH LAPACKE_dlamch	"SLAMCH and DLAMCH (Determine machine parameters for floating-point arithmetic)" on page 1210

LAPACKE Subroutines

Note the following about the LAPACKE subroutines:

- ESSL includes only the high-level C Interface to LAPACK.
- The definitions of the LAPACKE data types and prototypes are included in the ESSL header files. In the beginning of your program, before you call any of the LAPACKE subroutines, you must code the following statement for the ESSL header file:

```
#include <essl.h>
```

- The optional LAPACKE NaN checking is not supported.
- ESSL supports only C99 complex data types for C programs and C99 or STL complex data types for C++ programs. There is no support provided for custom complex data types. The default for ESSL is to use C99 complex datatypes for C programs and STL complex data types for C++ programs. If you wish to use C99 data types for C++ programs code the #include statement for the C99 complex floating point types (#include <complex.h>) in your program prior to coding the #include statement for the ESSL header file.
- For information about LAPACKE calling sequences, see "Syntax" on page xxxi

Using LAPACK with ESSL

ESSL contains a subset of LAPACK subroutines. If you want to use some of the LAPACK subroutines that are not included in ESSL, you can create a modified LAPACK library, in which the existing LAPACK subroutines that are contained in ESSL are removed.

The following example script shows the steps to create a modified Netlib LAPACK library:

```
#!/bin/bash
# Assumptions
#     liblapack.a is in the current directory
#     ESSL libs are in /usr/lib64
#
# First Pass - get the symbols in ESSL and LAPACK
nm liblapack.a           | grep " T " | awk '{ print $3 }' | sort -n > symbols.in.lapack
nm /usr/lib64/libessl.so | grep " T " | awk '{ print $3 }' | sort -n > symbols.in.essl
diff -y symbols.in.lapack symbols.in.essl | grep -v -e '>' -e '|' -e '<'  | awk '{ print $1 }' > duplicate.symbols
wc -l duplicate.symbols

# Second Pass - remove the ESSL symbols from the LAPACK library
cp -p liblapack.a liblapackforessl.a

for SYMBOL in $(cat duplicate.symbols | awk '{ print $1 }')
do
   fn=$SYMBOL
   unders="no"
   echo $SYMBOL | grep "_" > /dev/null && unders="yes"
   if [ "$unders" = "yes" ]
   then
    fn=`echo $SYMBOL | awk -F'_' '{print $1}'`
   fi
   ar d liblapackforessl.a $fn.o
done

# Check it worked
nm liblapackforessl.a           | grep " T " | awk '{ print $3 }' | sort -n > symbols.in.updatedlapack
diff -y symbols.in.updatedlapack symbols.in.essl | grep -v -e '>' -e '|' -e '<' | awk '{ print $1 }' >
duplicate.symbols.inupdatedlapack
wc -l duplicate.symbols.inupdatedlapack
```

|<

Appendix C. FFTW Version 3.1.2 to ESSL Wrapper Libraries

The following information lists the FFTW Version 3.1.2 wrappers that can be used for calling functions from the ESSL libraries.

Documentation for FFTW Version 3.1.2 can be found at the following URL:

http://www.fftw.org

Additional information about the FFTW Wrapper libraries can be found in the following file:

```
/opt/ibmmath/essl/version.release/FFTW3/README
```

C and Fortran Wrappers

The following tables list the available C and Fortran wrappers.

Table 273: List of available C and Fortran wrappers.

Category	C Wrapper	Fortran Wrapper
Plan usage	fftw_execute fftwf_execute fftw_destroy_plan fftwf_destroy_plan fftw_cleanup fftwf_cleanup	DFFTW_EXECUTE SFFTW_EXECUTE DFFTW_DESTROY_PLAN SFFTW_DESTROY_PLAN DFFTW_CLEANUP SFFTW_CLEANUP
Basic interface (Complex DFTs)	fftw_plan_dft_1d fftwf_plan_dft_1d fftw_plan_dft_2d fftwf_plan_dft_2d fftw_plan_dft_3d fftwf_plan_dft_3d fftw_plan_dft fftwf_plan_dft	DFFTW_PLAN_DFT_1D SFFTW_PLAN_DFT_1D DFFTW_PLAN_DFT_2D SFFTW_PLAN_DFT_2D DFFTW_PLAN_DFT_3D SFFTW_PLAN_DFT_3D DFFTW_PLAN_DFT SFFTW_PLAN_DFT

Table 273: List of available C and Fortran wrappers. (continued)

Category	**C Wrapper**	**Fortran Wrapper**
Basic interface (Real-data DFTs)	fftw_plan_dft_r2c_1d fftwf_plan_dft_r2c_1d fftw_plan_dft_r2c_2d fftwf_plan_dft_r2c_2d fftw_plan_dft_r2c_3d fftwf_plan_dft_r2c_3d fftw_plan_dft_r2c fftwf_plan_dft_r2c fftw_plan_dft_c2r_1d fftwf_plan_dft_c2r_1d fftw_plan_dft_c2r_2d fftwf_plan_dft_c2r_2d fftw_plan_dft_c2r_3d fftwf_plan_dft_c2r_3d fftw_plan_dft_c2r fftwf_plan_dft_c2r	DFFTW_PLAN_DFT_R2C_1D SFFTW_PLAN_DFT_R2C_1D DFFTW_PLAN_DFT_R2C_2D SFFTW_PLAN_DFT_R2C_2D DFFTW_PLAN_DFT_R2C_3D SFFTW_PLAN_DFT_R2C_3D DFFTW_PLAN_DFT_R2C SFFTW_PLAN_DFT_R2C DFFTW_PLAN_DFT_C2R_1D SFFTW_PLAN_DFT_C2R_1D DFFTW_PLAN_DFT_C2R_2D SFFTW_PLAN_DFT_C2R_2D DFFTW_PLAN_DFT_C2R_3D SFFTW_PLAN_DFT_C2R_3D DFFTW_PLAN_DFT_C2R SFFTW_PLAN_DFT_C2R
Advanced interface (Complex DFTs)	fftw_plan_many_dft fftwf_plan_many_dft	DFFTW_PLAN_MANY_DFT SFFTW_PLAN_MANY_DFT
Advanced interface (Real-data DFTs)	fftw_plan_many_dft_r2c fftwf_plan_many_dft_r2c fftw_plan_many_dft_c2r fftwf_plan_many_dft_c2r	DFFTW_PLAN_MANY_DFT_R2C SFFTW_PLAN_MANY_DFT_R2C DFFTW_PLAN_MANY_DFT_C2R SFFTW_PLAN_MANY_DFT_C2R
Guru interface (Complex DFTs)	fftw_plan_guru_dft fftwf_plan_guru_dft fftw_execute_dft fftwf_execute_dft	DFFTW_PLAN_GURU_DFT SFFTW_PLAN_GURU_DFT DFFTW_EXECUTE_DFT SFFTW_EXECUTE_DFT
Guru interface (Real-data DFTs)	fftw_plan_guru_dft_r2c fftwf_plan_guru_dft_r2c fftw_plan_guru_dft_c2r fftwf_plan_guru_dft_c2r fftw_execute_dft_r2c fftwf_execute_dft_r2c fftw_execute_dft_c2r fftwf_execute_dft_c2r	DFFTW_PLAN_GURU_DFT_R2C SFFTW_PLAN_GURU_DFT_R2C DFFTW_PLAN_GURU_DFT_C2R SFFTW_PLAN_GURU_DFT_C2R DFFTW_EXECUTE_DFT_R2C SFFTW_EXECUTE_DFT_R2C DFFTW_EXECUTE_DFT_C2R SFFTW_EXECUTE_DFT_C2R
Memory allocation	fftw_malloc fftwf_malloc fftw_free fftwf_free	(Not applicable)

Use of the ESSL FFTW Wrapper library has the following restrictions:

- No functions for real-to-real transforms are provided.
- No wrappers for the FFTW Wisdom functions are provided.
- The following flags are treated as equivalent: FFTW_ESTIMATE, FFTW_MEASURE, FFTW_PATIENT, FFTW_EXHAUSTIVE

- The FFTW Wrapper libraries use the same method to specify SMP parallelism as does ESSL instead of using the fftw_threads_init, fftw_plan_with_threads and fftw_cleanup_threads functions. These functions are not provided as part of the FFTW Wrapper libraries.

Using the FFTW Wrapper libraries

Applications using the FFTW Wrapper library can be linked with either the ESSL Serial Library or the ESSL SMP library in the 32-bit integer, 64-bit pointer environment.

>|Global subroutine names with or without trailing underscores are supported for the Fortran wrappers. |<

For additional information about how to use the FFTW Wrapper libraries, see Chapter 5, “Processing Your Program,” on page 169.

Building the FFTW Wrapper libraries

The C and Fortran wrappers are provided as source code to be compiled using the IBM C/C++ Compiler or the gcc compiler. If ESSL was installed in the default location, the source code, header files and makefiles can be found in the /opt/ibmmath/essl/*version.release*/FFTW3 directory.

Note: By default, the FFTW Wrapper libraries are installed in `/usr/local/lib64`. If you wish to install them to `/usr/local/lib` instead, you can change the value of LIBSUBDIR in Makefile or `Makefile.gcc`.

To build and install the FFTW Wrapper libraries using IBM XL C:

1. Change to a writable directory with approximately 512kb of free space.
2. Do one of the following:
   - Enter:

     cp /opt/ibmmath/essl/*version.release*/FFTW3/src/Makefile.gcc
     make install

     —or—
   - Enter:

     make -f /opt/ibmmath/essl/*version.release*/FFTW3/src/Makefile install

To build and install the FFTW Wrapper libraries using gcc:

1. Change to a writable directory with approximately 512kb of free space.
2. Do one of the following:
   - Enter:

     cp /opt/ibmmath/essl/*version.release*/FFTW3/src/Makefile.gcc
     make -f ./Makefile.gcc install
   - Enter:

     make -f /opt/ibmmath/essl/*version.release*/FFTW3/src/Makefile.gcc install

Appendix D. Using ESSL with netlib-java and Python

The ESSL libraries provide the BLAS and CBLAS subroutines which can be used by some other programming languages such as:

- netlib-java
- Python (Numpy and Scipy)

This appendix shows how to use the ESSL libraries with these programming languages, assuming you installed the ESSL packages with a symbolic link in the /usr directory. If you installed the ESSL package without creating the symbolic link in the /usr directory, you must set the correct path for the ESSL runtime libraries, as shown in the example below:

```
>|/opt/ibmmath/essl/6.1/lib64/libessl.so|<
```

Setting the BLAS runtime library for netlib-java and Python

Do one of the following to set up the BLAS runtime library for netlib-java and Python:

- To run with the 32-bit integer, 64-bit pointer environment ESSL Serial library as root set, the symbolic link as follows:

  ```
  ln -s /usr/lib64/libessl.so /usr/lib64/libblas.so.3
  ```

- To run with the 32-bit integer, 64-bit pointer environment ESSL SMP library as root, set the symbolic link as follows:

  ```
  ln -s /usr/lib64/libesslsmp.so /usr/lib64/libblas.so.3
  ```

- To run with the 32-bit integer, 64-bit pointer environment ESSL SMP CUDA library, as root set the symbolic link as follows:

  ```
  ln  -s /usr/lib64/libesslsmpcuda.so /usr/lib64/libblas.so.3
  ```

  Set the options as suggested "Using the ESSL SMP CUDA Library" on page 42 to get the best performance.

Apache Spark and netlib-java Example

netlib-java is a JAVA API used by Apache Spark for linear algebra subroutines such as BLAS, LAPACK, and ARPACK. Use the instructions listed at the following URL to enable the Native Optimized BLAS/CBLAS library for netlib-java:

```
https://developer.ibm.com/linuxonpower/2016/10/13/enable-system-native-blas-library-for-netlib-java-on-openpower-linux-
systems-2/
```

You can then run the following sample Java application using ESSL:

```
import com.github.fommil.netlib.BLAS;
public class ddot_example {
  public static void main(String[] args) {
    System.out.println(BLAS.getInstance().getClass().getName());
    double[] x = {1.0, 2.0, -3.0, 4.0, 5.0};
    double[] y = {9.0, 8.0, 7.0, -6.0, 5.0};

    int n;
    int nx = x.length;
    int ny = y.length;

    if (nx != ny) {
      System.err.println("Size of array X is not the same as size of array Y.\nAborting...");
      System.exit(-1);
    }

    n = nx;
    System.out.println("Answer = " + BLAS.getInstance().ddot(n, x, 1, y, 1));
  }
}
```

Figure 16: ddot_example.java

Compile and run the example code as follows:

```
javac –cp <PATH_TO_netlib-java Jar files> ddot_example.java
java –cp <PATH_TO_netlib-java Jar files>:. ddot_example
```

Figure 17 on page 1244 shows the output:

```
$ java -cp /mnt/workspace/essl_netlib_example.jar:. ddot_example

Oct 13, 2016 2:45:11 AM com.github.fommil.jni.JniLoader liberalLoad

INFO: successfully loaded /tmp/jniloader4688291511028652171netlib-native_system-linux-
ppc64le.so
com.github.fommil.netlib.NativeSystemBLAS

Answer = 5.0
```

Figure 17: Output from ddot_example.java

Scalca and netlib-java Example

MLlib uses the linear algebra package Breeze (see `http//www.scalanlp.org/`), which depends on netlib-java (see `http:github.com/fommil/netlib-java` for optimized numerical processing. Once you have enabled the system native BLAS library such as ESSL for netlib-java, you can run your application using ESSL, as shown in the example below:

```
import com.github.fommil.netlib.BLAS.{getInstance => blas}

println(blas.getClass().getName())


var x = Array(1.0, 2.0, -3.0, 4.0, 5.0)

var y = Array(9.0, 8.0, 7.0, -6.0, 5.0)


var nx = x.length

var ny = y.length

if (nx != ny) {

println("Size of array X is not the same as size of array Y.\nAborting...")

System.exit(1)

}

var n = nx


val res = blas.ddot(n, x, 1, y, 1)

println("Answer = " + res)

System.exit(0)
```

Figure 18: ddot_example.scala

Figure 19 on page 1245 shows the output:

```
16/10/13 03:25:46 INFO JniLoader: successfully loaded
/tmp/jniloader5596681969452523481netlib-native_system-linux-ppc64le.so
com.github.fommil.netlib.NativeSystemBLAS
x: Array[Double] = Array(1.0, 2.0, -3.0, 4.0, 5.0)
y: Array[Double] = Array(9.0, 8.0, 7.0, -6.0, 5.0)
nx: Int = 5
ny: Int = 5
n: Int = 5
res: Double = 5.0
Answer = 5.0
```

Figure 19: Ouput from ddot_example.scala

Python numpy Example

numpy is an open-source add-on module to Python that provides common mathematical and numerical subroutines in pre-compiled, fast functions. After setting the symbolic link for the BLAS library, you can also run the python application using ESSL, as shown in the example below.

```
#!/usr/bin/python
import numpy as np

x=np.array([ 1.0, 2.0, -3.0, 4.0, 5.0 ])
y=np.array([ 9.0, 8.0, 7.0, -6.0, 5.0 ])
print np.dot(x, y)
```

Figure 20: ddot_example.py

Figure 21 on page 1246 shows the output:

```
16/10/13 03:25:46 INFO JniLoader: successfully loaded
/tmp/jniloader5596681969452523481netlib-native_system-linux-ppc64le.so
com.github.fommil.netlib.NativeSystemBLAS
x: Array[Double] = Array(1.0, 2.0, -3.0, 4.0, 5.0)
y: Array[Double] = Array(9.0, 8.0, 7.0, -6.0, 5.0)
nx: Int = 5
ny: Int = 5
n: Int = 5
res: Double = 5.0
Answer = 5.0
```

Figure 21: Ouput from ddot_example.py

Accessibility features for ESSL

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility Features

The following list includes the major accessibility features in IBM ESSL. These features support:

- Keyboard-only operation that uses standard Microsoft Windows navigation keys.
- Interfaces that are commonly used by screen readers.
- Keys that are tactilely discernible and do not activate just by touching them.
- Industry-standard devices for ports and connectors.
- The attachment of alternative input and output devices.

The ESSL documentation that is located in the IBM Knowledge Center is accessibility-enabled. The accessibility features of IBM Knowledge Center are described in the Accessibility topic.

IBM and accessibility

For more information about the commitment that IBM has to accessibility, see IBM Accessibility.

Notices

Trademarks

Software update protocol

Privacy policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as the customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.

Bibliography

References

Text books and articles covering the mathematical aspects of ESSL are listed here, as well as several software libraries available from other companies. They are listed alphabetically as follows:

- Publications are listed by the author's name. IBM publications that include an order number, other than an *IBM Technical Report* can be ordered through the Subscription Library Services System (SLSS). The non-IBM publications listed here should be obtained through publishers, bookstores, or professional computing organizations.
- Software libraries are listed by their product name. Each reference includes the names, addresses, and phone numbers of the companies from which they can be obtained.

Each citation is shown as a number enclosed in square brackets. It indicates the number of the item listed in the bibliography. For example, reference [1] cites the first item listed below.

1. Agarwal, R. C. Dec. 1984. "An Efficient Formulation of the Mixed-Radix FFT Algorithm." *Proceedings of the International Conference on Computers, Systems, and Signal Processing*, 769–772. Bangalore, India.
2. Agarwal, R. C. August 1988. "A Vector and Parallel Implementation of the FFT Algorithm on the IBM 3090." *Proceedings from the IFIP WG 2.5 (International Federation for Information Processing Working Conference 5)* , Stanford University.
3. This is the new one
4. Agarwal, R. C. 1989. "A Vector and Parallel Implementation of the FFT Algorithm on the IBM 3090." *Aspects of Computation on Asynchronous Parallel Processors* , 45–54. Edited by M. H. Wright. Elsevier Science Publishers, New York, N. Y.
5. Agarwal, R. C.; Cooley, J. W. March 1986. "Fourier Transform and Convolution Subroutines for the IBM 3090 Vector Facility." *IBM Journal of Research and Development* , 30(2):145–162 (Order no. G322-0146).
6. Agarwal, R. C.; Cooley, J. W. September 1987. "Vectorized Mixed-Radix Discrete Fourier Transform Algorithms" *IEEE Proceedings* , 75:1283–1292.
7. Agarwal, R.; Cooley, J.; Gustavson F.; Shearer J.; Slishman G.; Tuckerman B. March 1986. "New Scalar and Vector Elementary Functions for the IBM System/370." *IBM Journal of Research and Development* , 30(2):126–144 (Order no. G322-0146).
8. Agarwal, R.; Gustavson F.; Zubair, M. May 1994. "An Efficient Parallel Algorithm for the 3-D FFT NAS Parallel Benchmark." *Proceedings of IEEE SHPCC 94* :129–133.
9. Anderson, E.; Bai, Z.; Bischof, C.; Demmel, J.; Dongarra, J.; DuCroz, J.; Greenbaum, A.; Hammarling, S.; McKenney, A.; Ostrouchov, S.; Sorensen, D. 1999. *LAPACK User's Guide* (third edition), SIAM Publications, Philadelphia, Pa.

   For more information, see:

   ```
   http://www.netlib.org/lapack/index.html
   ```
10. Anderson, E.; Bai, Z.; Bischof, C.; Demmel, J.; Dongarra, J.; DuCroz, J.; Greenbaum, A.; Hammarling, S.; McKenney, A.; Sorensen, D. May 1990. *LAPACK: A Portable Linear Algebra Library for High-Performance Computers*. University of Tennessee, Technical Report CS-90-105.
11. Basic Linear Algebra Subprograms Technical (BLAST) Forum. August 21, 2001."C Interface to the Legacy BLAS." *Basic Linear Algebra Subprograms Technical (BLAST) Forum*. 180 – 195. University of Tennessee.
12. Bathe, K.; Wilson, E. L. 1976. *Numerical Methods in Finite Element Analysis* , 249–258.

13. Bluestein, L. I. 1968. "A linear filtering approach to the computation of the discrete Fourier transform." *Northeast Electronics Research and Engineering Meeting Record* 10, 218-219.
14. Box, G. E. P.; Muller, Mervin E. 1958. "A Note on the Generation of Random Normal Deviates." *Annals of Mathematical Statistics* 29(2), 610-611.
15. Braman, K.; Byers, R.; Mathias, R. 2002 "The Multi-Shift QR Algorithm, Part I: Maintaining Well-focused Shifts and Level 3 Performance." *SIAM Journal on Matrix Analysis and Applications* 23(4):929–947.
16. Braman, K.; Byers, R.; Mathias, R. 2002 "The Multi-Shift QR Algorithm, Part II: Aggressive Early Deflation Maintaining Well-focused Shifts, and Level 3 Performance." *SIAM Journal on Matrix Analysis and Applications* 23(4):948–973.
17. Brayton, R. K.; Gustavson F. G.; Willoughby, R. A.; 1970. "Some Results on Sparse Matrices." *Mathematics of Computation* , 24(112):937–954.
18. Borodin, A.; Munro, I. 1975. *The Computational Complexity of Algebraic and Numeric Problems* American Elsevier, New York, N. Y.
19. Bunch, James R.; Kaufman, Linda. 1977. "Some Stable Methods for Calculating Inertia and Solving Symmetric Linear Systems"*Mathematics of Computation*, 31(137):163-179
20. Carey, G. F.; Oden, J. T. 1984. *Finite Elements: Computational Aspects, Vol 3* , 144–147. Prentice Hall, Englewood Cliffs, N. J.
21. Chan, T. F. March 1982. "An Improved Algorithm for Computing the Singular Value Decomposition." *ACM Transactions on Mathematical Software* 8(1):72–83.
22. Cline, A. K.; Moler, C. B.; Stewart, G. W.; Wilkinson, J. H. 1979. "An Estimate for the Condition Number of a Matrix." *SIAM Journal of Numerical Analysis* 16:368–375.
23. Conte, S. D.; DeBoor, C. 1972. *Elementary Numerical Analysis: An Algorithmic Approach®* (second edition), McGraw-Hill, New York, N. Y.
24. Cooley, J. W. 1976. "Fast Fourier Transform." *Encyclopedia of Computer Sciences* Edited by A. Ralston. Auerbach Publishers.
25. Cooley, J. W.; Lewis, P. A. W.; Welch, P. D. June 1967. "Application of the Fast Fourier Transform to Computation of Fourier Integrals, Fourier Series, and Convolution Integrals." *IEEE Transactions Audio Electroacoustics* AU-15:79–84.
26. Cooley, J. W.; Lewis, P. A. W.; Welch, P. D. June 1967. "Historical Notes on the Fast Fourier Transform." *IEEE Transactions Audio Electroacoustics* AU-15:76–79. (Also published Oct. 1967 in *Proceedings of IEEE* 55(10):1675–1677.)
27. Cooley, J. W.; Lewis, P. A. W.; Welch, P. D. March 1969. "The Fast Fourier Transform Algorithm and its Applications." *IEEE Transactions on Education* E12:27–34.
28. Cooley, J. W.; Lewis, P. A. W.; Welch, P. D. June 1969. "The Finite Fast Fourier Transform." *IEEE Transactions Audio Electroacoustics* AU-17:77–85.
29. Cooley, J. W.; Lewis, P. A. W.; Welch, P. D. July 1970. "The Fast Fourier Transform: Programming Considerations in the Calculation of Sine, Cosine, and LaPlace Transforms." *Journal of Sound Vibration and Analysis* 12(3):315–337.
30. Cooley, J. W.; Lewis, P. A. W.; Welch, P. D. July 1970. "The Application of the Fast Fourier Transform Algorithm to the Estimation of Spectra and Cross-Spectra." *Journal of Sound Vibration and Analysis* 12(3):339–352.
31. Cooley, J. W.; Lewis, P. A. W.; Welch, P. D. 1977. "Statistical Methods for Digital Computers." *Mathematical Methods for Digital Computers* Chapter 14. Edited by Ensein, Ralston and Wilf, Wiley-Interscience. John Wiley, New York.
32. Cooley, J. W.; Tukey, J. W. April 1965. "An Algorithm for the Machine Calculation of Complex Fourier Series." *Mathematics of Computation* 19:297.
33. Dahlquist, G.; Bjorck, A.; (Translated by Anderson, N.). 1974. *Numerical Methods* , Prentice Hall, Englewoods Cliffs, N. J. (For skyline subroutines, see 169–170.)
34. Davis, P. J.; Rabinowitz, P. 1984. *Methods of Numerical Integration* , (second edition), Academic Press, Orlando, Florida.

35. Demmel, J.; Kahan, W. February 1988. "Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy", LAPACK Working Note 3, ANL, MCS-TM-110.

    You can dowload this document from the following URL:

    ```
    http://www.netlib.org/lapack/lawns/lawn03.ps
    ```

36. Delsarte, P.; Genin, Y. V. June 1986. "The Split Levinson Algorithm." *IEEE Transactions on Acoustics, Speech, and Signal Processing* ASSP-34(3):472.

37. >|Dhillon, Inderjit S.; Parlett, Beresford N.; Vomel, Christof. "LAPACK Working Notes 162" *The Design and Implementation of the MRRR Algorithm* 200.

    You can download this document from:

    ```
    http://www.netlib.org/lapack/lawnspdf/lawn162.pdf
    ```

    |<

38. Di Chio, P.; Filippone, S. January 1992. "A Stable Partition Sorting Algorithm." *Report No. ICE-0045* IBM European Center for Scientific and Engineering Computing, Rome, Italy.

39. Dodson, D. S.; Lewis, J. G. Jan. 1985. "Proposed Sparse Extensions to the Basic Linear Algebra Subprograms." *ACM SIGNUM Newsletter* , 20(1).

40. Dongarra, J. J. July 1997. "Performance of Various Computers Using Standard Linear Equations Software." University of Tennessee, CS-89-85.

    You can download this document from:

    ```
    http://www.netlib.org/benchmark/performance.ps
    ```

41. Dongarra, J. J.; Bunch, J. R.; Moler C. B.; Stewart, G. W. 1986. *LINPACK User's Guide* , SIAM Publications, Philadelphia, Pa.

    For more information, see:

    ```
    http://www.netlib.org/linpack/index.html
    ```

42. Dongarra, J. J.; DuCroz, J.; Hammarling, S.; Duff, I. March 1990. "A Set of Level 3 Basic Linear Algebra Subprograms." *ACM Transactions on Mathematical Software* , 16(1):1–17.

43. Dongarra, J. J.; DuCroz, J.; Hammarling, S.; Duff, I. March 1990. "Algorithm 679. A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs." *ACM Tranactions on Mathematical Software* , 16(1):18–28.

44. Dongarra, J. J.; DuCroz, J.; Hammarling, S.; Hanson, R. J. March 1988. "An Extended Set of Fortran Basic Linear Algebra Subprograms." *ACM Transactions on Mathematical Software* , 14(1):1–17.

45. Dongarra, J. J.; DuCroz, J.; Hammarling, S.; Hanson, R. J. March 1988. "Algorithm 656. An Extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs." *ACM Tranactions on Mathematical Software* , 14(1):18–32.

46. Dongarra, J. J.; Duff, I. S.; Sorensen, D. C.; Van der Vorst, H. 1991. *Solving Linear Systems on Vector and Shared Memory Computers* , SIAM Publications, ISBN 0-89871-270-X.

47. Dongarra, J. J.; Eisenstat, S. C. May 1983. "Squeezing the Most Out of an Algorithm in Cray Fortran." *Technical Memorandum 9* Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439.

48. Dongarra, J. J.; Gustavson, F. G.; Karp, A. Jan. 1984. "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine." *SIAM Review* , 26(1).

49. Dongarra, J. J.; Kaufman, L.; Hammarling, S. Jan. 1985. "Squeezing the Most Out of Eigenvalue Solvers on High-Performance Computers." *Technical Memorandum 46* Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439.

50. Dongarra, J. J.; Kolatis M. October 1994. "Call Conversion Interface (CCI) for LAPACK/ESSL." LAPACK Working Note 82, Department of Computer Science University of Tennessee, Knoxville, Tennessee.

    You can download this document from:

    ```
    http://www.netlib.org/lapack/lawns/lawn82.ps
    ```

51. Dongarra, J. J.; Kolatis M. May 1994. "IBM RS/6000-550 & -590 Performance for Selected Routines in ESSL/LAPACK/NAG/IMSL", LAPACK Working Note 71, Department of Computer Science University of Tennessee, Knoxville, Tennessee.

    You can download this document from:

    ```
    http://www.netlib.org/lapack/lawns/lawn71.ps
    ```

52. Dongarra, J. J; Meuer, H. W.; Strohmaier, E. June 1997. "Top500 Supercomputer Sites." University of Tennessee, UT-CS-97-365.; University of Mannheim, RUM 50/97.

    You can view this document from:

    ```
    http://www.netlib.org/benchmark/top500.html
    ```

53. Dongarra, J. J.; Moler, C. B. August 1983. "EISPACK—A Package for Solving Matrix Eigenvalue Problems." *Technical Memorandum 12* Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439.
54. Dongarra, J. J.; Moler, C. B; Bunch, J. R.; Stewart, G. W. 1979. *LINPACK Users' Guide* , SIAM, Philadelphia, Pa.
55. Dubrulle, A. A. 1971. "QR Algorithm with Implicit Shift." IBM licensed program: PL/MATH.
56. Dubrulle, A. A. November 1979. "The Design of Matrix Algorithms for Fortran and Virtual Storage." *IBM Palo Alto Scientific Center Technical Report* (Order no. G320-3396).
57. Dubrulle, A. A. November 1988. "A Version of EISPACK for the IBM 3090VF", *IBM Palo Alto Scientific Center Technical Report* (Order no. G320-3510).
58. Duff, I. S.; Erisman, A. M.; Reid, J. K. 1986. *Direct Methods for Sparse Matrices* Oxford University Press (Claredon), Oxford. (For skyline subroutines, see 151–153.)
59. Eisenstat, S. C. March 1981. "Efficient Implementation of a Class of Preconditioned Conjugate Gradient Methods." *SIAM Journal of Scientific Statistical Computing* , 2(1).
60. EISPACK software library; National Energy Software Center, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439 (312-972-7250); International Mathematical and Statistical Libraries, Inc., Sixth Floor, GNB Building, 7500 Bellaire Boulevard, Houston, Texas 77036 (713-772-1927)
61. Elmroth, E.; Gustavson, F. "Applying Recursion to Serial and Parallel QR Factorization Leads to Better Performance." To be Published.*IBM J. Res. Develop. 44, No. 5*.
62. Elmroth, E.; Gustavson, F. "A High-Performance Algorithm for the Linear Least Squares Problem on SMP Systems." Submitted for Publication. *Lecture Notes in Computer Science* Springer-Verlag, Berlin, 2000.
63. Elmroth, E.; Gustavson, F. June 1998. "New Serial and Parallel Recursive *QR* Factorization Algorithms for SMP Systems." *Applied Parallel Computing Large Scale Scientific and Industrial Problems* , 4th International Workshop, PARA'98 Umea, Sweden, June 14-17, 1998 Proceedings:120—128.
64. Filippone, S.; Santangelo, P.; Vitaletti M. Nov. 1990. "A Vectorized Long-Period Shift Register Random Number Generation." *Proceedings of Supercomputing '90* , 676–684, New York.
65. Forsythe, G. E.; Malcolm, M. A. 1977. *Computer Methods for Mathematical Computations* , Prentice Hall, Englewoods Cliffs, N. J.
66. Forsythe, G.E.; Moler, C. 1967. *Computer Solution of Linear Algebra Systems* , Prentice Hall, Englewoods Cliffs, N. J.

67. Francis, J. G. F. 1961. "The QR Tranformation A Unitary Analogue to the LR Transformation—Part 1", *The Computer Journal Volume 4 Number 3*, 265-271, British Computer Society, London.
68. Francis, J. G. F. 1962. "The QR Tranformation—Part 2", *The Computer Journal Volume 4 Number 4*, 332–345, British Computer Society, London.
69. Freund, R. W. July 28, 1992. "Transpose-Free Quasi-Minimal Residual Methods for Non-Hermitian Linear Systems." *Numerical Analysis Manuscript 92-07* , AT&T Bell Laboratories. (To appear in *SIAM Journal of Scientific Statistical Computing* , 1993, Vol. 14.)
70. Gans, D. 1969. *Transformations and Geometries* Appleton Century Crofts, New York.
71. Garbow, B. S.; Boyle, J. M.; Dongarra, J. J.; Moler, C. B. 1977. "Matrix Eigensystem Routines." *EISPACK Guide Extension Lecture Notes in Computer Science, Vol. 51* Springer-Verlag, New York, Heidelberg, Berlin.
72. George, A.; Liu, J. W. 1981. "Computer Solution of Large Sparse Positive Definite Systems." *Series in Computational Mathematics* Prentice-Hall, Englewood Cliffs, New Jersey.
73. Gerald, C. F.; Wheatley, P. O. 1985. *Applied Numerical Analysis* (third edition), Addison-Wesley, Reading, Mass.
74. Gill, P. E.; Miller, G. R. 1972. "An Algorithm for the Integration of Unequally Spaced Data." *Computer Journal* 15:80–83.
75. Golub, G. H.; Van Loan, C. F. 1996. *Matrix Computations* , John Hopkins University Press, Baltimore, Maryland.
76. Gregory, R. T.; Karney, D. L. 1969. *A Collection of Matrices for Testing Computational Algorithms* , Wiley-Interscience, New York, London, Sydney, Toronto.
77. Grimes, R. C.; Kincaid, D. R.; Young, D. M. 1979. *ITPACK 2.0 User's Guide* , CNA-150. Center for Numerical Analysis, University of Texas at Austin.
78. Gustavson, Fred.; Alexander Karaivanov, Minka I. Marinova, Jerzy Wasniewski, Plamen Yalamov. "A new block packed storage for symmetric indefinite matrices." *Lecture Notes® in Computer Science* Fifth International Workshop, Bergen, Norway.
79. Gustavson, F.G. Nov. 1997. "Recursion leads to automatic variable blocking for dense linear-algebra algorithms." *IBM Journal of Research and Development*, Volume 41 Number 6:737—755.
80. Gustavson, F.G. Jan. 1997. "High Performance Linear Algebra Algorithms Using New Generalized Data Structures for Matrices." *IBM Journal of Research and Development*, Volume 47 Number 1.
81. Gustavson, F.; Henriksson, A.; Jonsson, I.; Kagstrom, B.; Ling, P. June 1998. "Recursive Blocked Data Formats and BLAS's for Dense Linear Algebra Algorithms." *Applied Parallel Computing Large Scale Scientific and Industrial Problems* , 4th International Workshop, PARA'98 Umea, Sweden, June 14-17, 1998 Proceedings:195—215.
82. Hageman, L. A.; Young, D. M.. 1981. *Applied Iterative Methods* Academic Press, New York, N. Y.
83. Higham, N. J. 1996. *Accuracy and Stability of Numerical Algorithms* , SIAM Publications, Philadelphia, Pa.
84. Higham, N. J. December 1988. *Fortran Codes for Estimating the One-Norm of a Real or Complex Matrix, with Application to Condition Estimating* ACM Transactions on Mathematical Software, 14(4):381–396.
85. Jennings, A. 1977. *Matrix Computation for Engineers and Scientists* , 153–158, John Wiley and Sons, Ltd., New York, N. Y.
86. Kagstrom, B.; Ling, P.; Van Loan, C. 1993. "Portable High Performance GEMM-Based Level 3 BLAS", *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing* , 339–346. Edited by: R. Sincovec, D. Keyes, M. Leize, L. Petzold, and D. Reed. SIAM Publications.
87. Kagstrom, B. 1993. "A Direct Method for Reordering Eigenvalues in the Generalized Real Schur Form of a Regular Matrix Pair (A,B)", *Linear Algebra for Large Scale and Real-Time Applications* , 195–218. Edited by: Moonen, M.S., Golub, Gene H., de Moor, B.L. Kluwer Academic Publishing.
88. Kagstrom, B.; Poromaa, P. 1994. "Computing Eigenspaces with Specified Eigenvalues of a Regular Matrix Pair (A, B) and Condition Estimation: Theory, Algorithms and Software", *Report UMINF - 94.04* , University of Umea.

89. Kincaid, D. R.; Oppe, T. C.; Respess, J. R.; Young, D. M. 1984. *ITPACKV 2C User's Guide* , CNA-191. Center for Numerical Analysis, University of Texas at Austin.
90. Kirkpatrick, S.; Stoll, E. P. 1981. "A Very Fast Shift-Register Sequence Random Number Generation." *Journal of Computational Physics* , 40:517–526.
91. Knuth, D. E. 1973. *The Art of Computer Programming, Vol. 3: Sorting and Searching* , Addison-Wesley, Reading, Mass.
92. Knuth, D. E. 1981. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms* , (second edition), Addison-Wesley, Reading, Mass.
93. Lambiotte, J. J.; Voigt, R. G. December 1975. "The Solution of Tridiagonal Linear Systems on the CDC STAR-100 Computer." *ACM Transactions on Mathematical Software* 1(4):308–329.
94. Lawson, C. L.; Hanson, R. J. 1974. *Solving Least Squares Problems* Prentice-Hall, Englewood Cliffs, New Jersey.
95. Lawson, C. L.; Hanson, R. J.; Kincaid, D. R.; Krough, F. T. Sept. 1979. "Basic Linear Algebra Subprograms for Fortran Usage." *ACM Transactions on Mathematical Software* 5(3):308–323.
96. Lewis, P. A. W.; Goodman, A. S.; Miller, J. M. 1969. "A Pseudo-Random Number Generator for the System/360." *IBM System Journal* , 8(2).
97. Matsumoto, M.; Nishimura, T., 1998. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator." *ACM Transactions on Modeling and Computer Simulations*, 8(1):3-30.
98. Mutsuo, S.; Makoto, M. 2006. "SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator" *Monte Carlo and Quasi-Monte Carlo Methods*, 2006: 607-622.
99. Mutsuo, S.; Makoto, M. 2009. "A PRNG Specialized in Double Precision Floating Point Number Using an Affine Transition" *Monte Carlo and Quasi-Monte Carlo Methods*, 2009: 589-602.
100. McCracken, D. D.; Dorn, W. S. 1964. *Numerical Methods and Fortran Programming* , John Wiley and Sons, New York.
101. Melhem, R. 1987. "Toward Efficient Implementation of Preconditioned Conjugate Gradient Methods on Vector Supercomputers." *Journal of Supercomputer Applications* , Vol. 1.
102. Moler, C. B.; Stewart, G. W. 1973. "An Algorithm for the Generalized Matrix Eigenvalue Problem." *SIAM Journal of Numerical Analysis* , 10:241–256.
103. Nichols, B.; Farrell, J.; Buttlar, D. 1996. *Pthreads Programming: Using POSIX Threads* O'Reilly & Associates, Inc.
104. Oppenheim, A. V.; Schafer, R. W. 1975. *Digital Signal Processing* Prentice-Hall, Englewood Cliffs, New Jersey.
105. Oppenheim, A. V.; Weinstein, C. August 1972. "Effects of Finite Register Length in Digital Filtering and the Fast Fourier Transform." *IEEE Proceedings* , AU-17:209–215.
106. Parlett, B.; Marques, O. "An implementation of the dqds Algorithm (Positive Case)," *LAPACK Working Note 155*.

     You can download this document from:

     ```
     http://www.netlib.org/lapack/lawns/lawn155.ps
     ```
107. Saad, Y.; Schultz, M. H. 1986. "GMRES: A Generalized Minimum Residual Algorithm for Solving Nonsymmetric Linear Systems." *SIAM Journal of Scientific and Statistical Computing* , 7:856–869. Philadelphia, Pa.
108. Smith, B. T.; Boyle, J. M.; Dongarra, J. J.; Garbow, B. S.; Ikebe, Y.; Klema, V. C.; Moler, C. B. 1976. "Matrix Eigensystem Routines." *EISPACK Guide Lecture Notes in Computer Science, Vol. 6* Springer-Verlag, New York, Heidelberg, Berlin.
109. Sonneveld; Wesseling; DeZeeuv. 1985. *Multigrid and Conjugate Gradient Methods as Convergence Acceleration Techniques in Multigrid Methods for Integral and Differential Equations* , 117–167. Edited by D.J. Paddon and M. Holstein. Oxford University Press (Claredon), Oxford.

110. Sonneveld, P. January 1989. "CGS, a Fast Lanczos-Type Solver for Nonsymmetric Linear Systems." *SIAM Journal of Scientific and Statistical Computing* , 10(1):36–52.

111. Stewart, G. 1973. *Introduction to Matrix Computations* Academic Press, New York, N. Y.

112. Stewart, G. W. 1976. "The Economical Storage of Plane Rotations." *Numerische Mathematik* , 25(2):137–139.

113. Stroud, A. H.; Secrest, D. 1966. *Gaussian Quadrature Formulas* Prentice-Hall, Englewood Cliffs, New Jersey.

114. Suhl, U. H.; Aittoniemi, L. 1987. "Computing Sparse LU-Factorization for Large-Scale Linear Programming Bases." *Report Number 58* Freie University, Berlin.

115. Tausworthe, R. C. 1965. "Random Numbers Generated by Linear Recurrence Modulo Two." *Mathematical Computing, Vol. 19*

116. Van der Vorst, H. A. 1992. "Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems." *SIAM Journal of Scientific Statistical Computing* , 13:631–644.

117. Weinstein, C. September 1969. "Round-off Noise in Floating Point Fast Fourier Transform Calculation." *IEEE Transactions on Audio Electroacoustics* AU-17:209–215.

118. Wilkinson, J. H. 1965. *The Algebraic Eigenvalue Problem* , Oxford University Press (Claredon), Oxford.

119. Wilkinson, J. H. 1963. *Rounding Errors in Algebraic Processes* , Prentice-Hall, Englewood Cliffs, New Jersey.

120. Wilkinson, J. H.; Reinsch, C. 1971. *Handbook for Automatic Computation, Vol. II, Linear Algebra* , Springer-Verlag, New York, Heidelberg, Berlin.

121. Wilkonson, J.H. 1979. "Kronecker's Canonical Form and the QZ Algorithm" *Linear Algebra and its Applications* , 28:285–303.

122. Zierler, N. 1969 "Primitive Trinomials Whose Degree Is a Mersenne Exponent." *Information and Control* , 15:67–69.

123. Zlatev, Z. 1980. "On Some Pivotal Strategies in Gaussian Elimination by Sparse Technique." *SIAM Journal of Numerical Analysis* , 17(1):18–30.

Index