DIGITAL Extended Math Library

Reference Guide

Order Number: AA–RFTPA–TE

December 1998

This document describes the DIGITAL Extended Math Library (DXML). DXML is a set of high-performance mathematical subprograms designed for use in many different types of scientific and engineering applications. This document is a guide to using DXML and provides reference information for each of the subprograms in the library.

DIGITAL Equipment Corporation
Maynard, Massachusetts

December 1998

This document is available on CD–ROM.

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

Level 1 BLAS Subprograms

Level I BLAS Extensions Subprograms

7 Using the Sparse Level 1 BLAS Subprograms

Sparse Level 1 BLAS Subprograms

8 Using the Level 2 BLAS Subprograms

Level 2 BLAS Subprograms

9 Using the Level 3 BLAS Subprograms

Level 3 BLAS Subroutines

10 Using LAPACK Subprograms

11 Using the Signal Processing Subprograms

Fast Fourier Transform Subprograms

Cosine and Sine Transform Subprograms

Convolution and Correlation Subprograms

Filter Subprograms

12 Using the Iterative Solvers for Sparse Linear Systems

Sparse Iterative Solver Subprograms

13 Using the Direct Solvers for Sparse Linear Systems

Sparse Direct Solver Subprograms

14 Using the VLIB Routines

VLIB Routines

15 Using Random Number Generator Subprograms

Random Number Generator Subprograms

16 Using Sort Subprograms

Sort Subprograms

A Bibliography

Index

Examples

Figures

Tables

Preface

The Digital Extended Math Library (DXML) is a collection of high performance subprograms that perform different types of mathematical operations. DXML is primarily used with Fortran programs, but it can be used with many other languages. This book provides descriptions of the subprograms in the DXML library. It does not describe specific applications of the subprograms.

Intended Audience

This book is a guide for scientists, mathematicians, engineers, computer scientists, and programmers who want to write applications that call DXML subprograms.

To use this book, you need an understanding of computer concepts, knowledge of computer programming, and knowledge of mathematics in the areas of DXML computations.

Chapter Descriptions

This manual includes the following chapters and appendices:

- Chapter 1 provides an introduction to DXML.
- Chapter 2 explains how to prepare and store program data.
- Chapter 3 explains how to code an application program.
- Chapter 4 describes the parallel library.
- Chapter 5 explains how to compile and link an application program.
- Chapter 6 describes how to use the Level 1 BLAS subprograms and extensions and is followed by the descriptions of the BLAS Level 1 and Level 1 Extensions subprograms.
- Chapter 7 describes how to use the Sparse Level 1 BLAS subprograms, and is followed by the descriptions of the Sparse BLAS Level 1 supbprograms.
- Chapter 8 describes how to use the Level 2 BLAS subprograms, and is followed by the descriptions of the BLAS Level 2 subprograms.
- Chapter 9 describes how to use the Level 3 BLAS subprograms, and is followed by the descriptions of the BLAS Level 2 subprograms.
- Chapter 10 provides an overview of the LAPACK library of subroutines.
- Chapter 11 describes how to use the signal processing subprograms, and is followed by the descriptions of the signal processing subprograms.
- Chapter 12 describes how to use the Iterative Solvers for Sparse Linear Systems, and is followed by the descriptions of the sparse iterative solver subprograms.

- Chapter 13 describes how to use Direct Solvers for Sparse Linear Systems, and is followed by the descriptions of the sparse iterative solver subprograms.
- Chapter 14 describes how to use the VLIB subprograms, and is followed by the descriptions of the VLIB subprograms.
- Chapter 15 describes how to use the random number generator subprograms, and is followed by the descriptions of the random number generator subprograms.
- Chapter 16 describes how to use the sort subprograms, and is followed by the descriptions of the sort subprograms.
- Appendix A, Bibliography, provides resource information about the mathematical operations covered by DXML.

Associated Documentation

In addition to this manual, the following DXML documentation is available:

- Installation information - located in the *DIGITAL Fortran Installation Guide for DIGITAL UNIX Systems*
- Online release notes - ascii text file
- Online version of this manual - .PDF file
- Online HELP - UNIX manpages for DXML

Additionally, the following related documentation is recommended:

- *DIGITAL Fortran 90 User Manual for DIGITAL UNIX Systems*
- *DIGITAL Fortran Language Reference Manual*
- *LAPACK Users' Guide* (available from SIAM)

About LAPACK

To make use of the LAPACK library, DIGITAL recommends the purchase of the major documentation of LAPACK, in book form, published by the Society for Industrial and Applied Math (SIAM) in 1995:

LAPACK Users' Guide, 2nd Edition, by E. Anderson et al,
SIAM
3600 University City Science Center
Philadelphia PA 19104-2688
ISBN: 0-89871-345-5
Tel: 1-800-447-SIAM
email: service@siam.org

This documentation is also available on the internet in a format viewable by a web browser. To view this book on the internet, use the following URL:

http://www.netlib.org/lapack/lug/lapack_lug.html

About DXML Manpages

DXML contains the following hierarchy of manpages:

- A top-level manpage (dxml) consisting of a product overview and a list of the manpages that describe DXML subcomponents
- A manpage for each DXML subcomponent that describes its functionality and lists the manpages that describe its subprograms

- A manpage for each subprogram that provides details of its use and the operations it implements

Refer to the section of this Preface titled "Using Manpages" if you need information about accessing or using these manpages.

Conventions Used in this Book

In this book, OpenVMS refers to the DIGITAL OpenVMS operating system, and UNIX refers to the DIGITAL UNIX operating system. References to Windows NT includes Windows NT for Intel and Windows NT for Alpha, unless otherwise stated.

This book also uses the documentation conventions summarized in the following tables.

Table 1 General Documentation Conventions

Convention	Meaning
Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x* or GOLD *x*	A sequence such as PF1 *x* or GOLD *x* indicates that you must first press and release the key labeled PF1 or GOLD and then press and release another key or a pointing device button. GOLD key sequences can also have a slash (/), dash (–), or underscore (_) as a delimiter in EVE commands.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[ ]	In command format descriptions, brackets indicate optional elements. You can choose one, none, or all of the options. (Brackets are not optional in the syntax of a substring specification in an assignment statement.)
{ }	In command format descriptions, braces indicate a required choice of options; you must choose one of the options listed.

(continued on next page)

Table 1 (Cont.) General Documentation Conventions

Convention	Meaning
bold	Bold type is used to emphasize a word or phrase or to indicate math variables. In text, it represents the introduction of a new term or the name of an argument, an attribute, or a reason. In examples, it shows user input.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *device-name* contains up to five alphanumeric characters).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace type`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
`MONOSPACE TYPE`	Monospaced uppercase characters are used for lines of code, commands, and command qualifiers.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.
*	An asterisk means that no value is stored at that location in the array.

Table 2 Documentation Conventions for Math

Item	Example	Description	Usage
Vector	x	Lowercase italic	The vector x has six elements.
Vector element	x_2	Lowercase italic with one subscript indicating position	The second element of the vector x is x_2.
Matrix	A	Uppercase italic	The matrix A has three rows and four columns.
Matrix element	a_{23}	Lowercase italic with two subscripts indicating position	Element a_{23} is in the second row and third column of A.
Scalar quantity specifying length or count	m by n	Lowercase italic	A is an m by n band matrix with kl subdiagonals and ku superdiagonals.

Table 3 Documentation Conventions for Programming

Item	Example	Description	Usage
Array	A or X	Uppercase	The matrix A is stored in the array A, and the vector x is stored in the array X.
Array element	A(1,2) or X(3)	Uppercase with numbers in parentheses indicating position	A(1,2) is the element in the first row and second column of the array A. X(3) is the third element in the array X.
Arguments mentioned in text	**n** or **A**	Bold	The data length is specified by the **n** argument.

Table 4 Symbols and Expressions Used

Symbol or Expression	Description
α, β, a	Greek and English letters denoting scalar values
$\lvert incx \rvert$	Absolute value of $incx$
$A \leftarrow B$	Matrix A is replaced by matrix B
B^T y^T	Transpose of the matrix B; transpose of the vector y
B^{-1}	Inverse of the matrix B
B^{-T}	Inverse of the transpose of matrix B
B^m	Product of matrix B, m times
$\overline{B}$ $\overline{y}$	Complex conjugate of the matrix B; complex conjugate of the vector y
$B^H y^H$	Complex conjugate transpose of the matrix B; complex conjugate transpose of the vector y
$\overline{b}_{ij}$ $\overline{y}_i$	Complex conjugate of the matrix element b_{ij}; complex conjugate of the vector element y_i
$\sum_{i=1}^{n} x_i$	Sum of the elements x_1 to x_n
$\min \{x_j\}$	Minimum element in the vector x
$\max \{x_j\}$	Maximum element in the vector x

Using the Reference Sections

The following information may be helpful when using the reference sections contained in this book.

Each component of the DXML library is described in its own reference section. Each reference section consists of an introductory discussion followed by reference pages that describe the functionality, calling sequences, and parameters of each routine. Many of the routines come in four variations, indicated by the prefix in its name:

S	single-precision
D	double-precision

C	single-complex
Z	double-complex

The description of each routine combines the purpose and attributes of all four variations. When the discussion applies to all four versions, a leading underscore character is used for a prefix.

In this book, references to real*4, real*8, complex*8, and complex*16 refer to single-precision, double-precision, single-complex, and double-complex, as follows:

- real*4 = single-precision
- real*8 = double-precision
- complex*8 = single-complex
- complex*16 = double-complex

The following conventions indicate any differences among the four variations:

- Data types
  If a parameter's data type is the same for all variations of the routine, the data type is listed once:

  integer*4

  If a parameter has a different data type when used with each of the variations, the parameter's data type is documented in the following way:

  real*4 | real*8 | complex*8 | complex*16

  This indicates that the parameter must be single-precision when calling routines with the prefix S, double-precision when calling routines with the prefix D, single-complex when calling routines with the prefix C, and double-complex when calling routines with the prefix Z.

- Parameters

  For some routines, a variation requires additional parameters. This is indicated in the calling sequence in the following way:

  {S,D}xxxx(..., ..., ...)

  {C,Z}xxxx(..., rwork, ...)

- Ordering of routines

  In general, the BLAS routines are sorted alphabetically, However, some routines that have the same logical function are documented together even though they have different names. This occurs when the real version of the routine deals with a symmetric matrix (and so has SY in its name) and the complex version deals with a Hermitian matrix (HE). The signal processing, iterative solver, and direct solver routines are grouped according to the type of mathematical task.

Using Manpages

Online reference information for DXML is available in the form of manpages. Symbolic links to the manpages are installed in the */usr/share/man/man3* directory at installation time.

Use the *man* command, with the manpage name, to display a DXML manpage. For example, to display the product overview - which gives a brief summary of the contents of DXML, a list of subcomponents, and pointers to related information - enter the following command:

```
man dxml
```

To display the manpage that describes a subcomponent, such as BLAS 3, and its list of associated subprograms, use the name of the subcomponent, as follows:

```
man blas3
```

To display the manpage that provides a description of a subprogram, such as SAXPY, use the name of the subprogram, as follows:

```
man saxpy
```

Note About LAPACK Manpages

To display the manpage that gives an overview of LAPACK, and lists LAPACK routines and the operations they perform, use the following command:

```
man lapack
```

LAPACK routines have a separate manpage for each data type. When you use the *man* command for an LAPACK routine, the routine name should have the correct data type. For example, to display the manpage for the DGETRF routine, use the command:

```
man dgetrf
```

To display the CGETRF routine, use the command:

```
man cgetrf
```

Sending DIGITAL Your Comments

DIGITAL welcomes your comments on this product and on its documentation. You can send comments to us in the following ways:

- Internet electronic mail: *DXML@DIGITAL.COM*
- FAX: *603-884-0120, Attention: DXML Team, ZK02-3/Q18*
- A letter sent to the following address:

  DIGITAL Equipment Corporation
  High Performance Computing Group (DXML), ZK02-3/Q18
  110 Spit Brook Road
  Nashua, N.H. 03062-2698
  USA

If you have suggestions for improving particular sections, or find errors, please indicate the title, order number, and section number.

Getting Help from DIGITAL

If you have a customer support contract and have comments or questions about DXML software, you contact DIGITAL's Customer Support Center (CSC), preferably using electronic means such as DSNlink. In the United States, customers can call the CSC at 1-800-354-9000.

1
Introduction to DXML

Digital Extended Math Library (DXML) is a collection of high-performance, computationally-intensive mathematical subprograms designed for use in many different types of scientific and engineering applications. DXML subprograms are callable from any programming language.

1.1 Overview

DXML's subprograms cover the areas of Basic Linear Algebra, Linear System and Eigenproblem Solvers, Sparse Linear System Solvers, Sorting, Random Number Generation, and Signal Processing.

Where appropriate, each subprogram has a version to support each combination of real or complex arithmetic and single or double precision. The supported floating point format is IEEE.

- **Basic Linear Algebra Subprograms -** The Basic Linear Algebra Subprograms (BLAS) library includes the industry-standard Basic Linear Algebra Subprograms for Level 1 (vector-vector (BLAS1)), Level 2 (matrix-vector (BLAS2)), and Level 3 (matrix-matrix (BLAS3)). Also included are subprograms for BLAS Level 1 Extensions, Sparse BLAS Level 1, and Array Math Functions (VLIB).
- **Signal Processing Subprograms -** The Signal Processing library provides a basic set of signal processing functions. Included are one-, two-, and three-dimensional Fast Fourier Transforms (FFT), group FFTs, Cosine/Sine Transforms (FCT/FST), Convolution, Correlation, and Digital Filters.
- **Sparse Linear System Subprograms -** The Sparse Linear System library provides both direct and iterative sparse linear system solvers. The direct solver package supports both symmetric and nonsymmetric sparse matrices stored using the skyline storage scheme. The iterative solver package contains a basic set of storage schemes, preconditioners, and iterative solvers. The design of this package is modular and matrix-free, allowing future expansion and easy modification by users.
- **LAPACK subprograms -** The Linear System and Eigenproblem Solver library provides the complete LAPACK package developed by a consortium of university and government laboratories. LAPACK is an industry-standard subprogram package offering an extensive set of linear system and eigenproblem solvers. LAPACK uses blocked algorithms that are better suited to most modern architectures, particularly ones with memory hierarchies. LAPACK will supersede LINPACK and EISPACK for most users.

1.2 Parallel Library Support for Symmetric Multiprocessing

DXML supports symmetric multiprocessing (SMP) for improved performance on platforms that support SMP. Key BLAS Level 2 and 3 routines, the LAPACK GETRF and POTRF routines, the sparse iterative solvers, the skyline solvers, and the FFT routines have been modified to execute in parallel if run on SMP hardware. These parallel routines along with the other serial routines are supplied in an alternative library.

The user may choose to link with either the parallel (" -ldxmlp ") library, or the serial (" -ldxml ") library, depending on whether SMP support is required, since each library contains the complete set of routines. The parallel DXML library achieves its parallelization using OpenMP.

1.3 Cray SciLib Support (SCIPORT)

SCIPORT is Digital Equipment Corporation's implementation of the Cray Reasearch scientific numerical library, SciLib. SCIPORT provides 64-bit single-precision and 64-bit integer interfaces to underlying DXML routines for Cray users porting programs to Alpha systems running DIGITAL UNIX. SCIPORT also provides an equivalent version of almost all Cray Math Library and CF77 (Cray Fortran 77) Math intrinsic routines.

In order to be completely source code compatible with SciLib, the SCIPORT library calling sequence supports 64-bit integers passed by reference. However, internally, SCIPORT used 32 bit integers. Consequently, some run-time uses of SciLib are not supported by SCIPORT.

SCIPORT provides the following:

- 64-bit versions of all Cray SciLib single-precision BLAS Level 1, Level 2, and Level 3 routines
- All Cray SciLib LAPACK routines
- All Cray SciLib Special Linear System Solver routines
- All Cray SciLib Signal Processing routines
- All Cray SciLib Sorting and Searching routines

These routines are completely interchangeable with their Cray SciLib counterparts, up to the runtime limit on integer size - and with the exception of the ORDERS routine, require no program changes to function correctly. Due to endian differences of machine architecture, special considerations must be given when the ORDERS routine is used to sort multi-byte character strings.

1.4 Calling DXML from Programming Languages

DXML subprograms are callable from most programming languages. However, DXML subprograms follow Fortran conventions and assume a Fortran standard for the passing of arguments and for the storing of data. Unless specifically noted, all non-character arguments are passed by reference. Data in arrays is stored column by column.

If you are programming in a language other than Fortran, consult the specific language's user guide and reference manual for information about how that language stores and passes data. You may be required to set up your data differently from the way you normally would when using that language.

1.5 How DXML Achieves High Performance

DXML relies on the following design techniques to achieve high performance:

- Computational constructs maximize the use of available instructions and promote pipelined use of functional units.
- Where appropriate, selected routines are available in parallel and serial, to offer additional performance on multiprocessor (SMP) systems.
- The hierarchical memory system is efficiently managed by enhancing the data locality of reference:
  - Data in registers is often reused to minimize load and store operations.
  - The cache is managed efficiently to maximize the locality of reference and data reuse. For example, the algorithms are structured to operate on sub-blocks of arrays that are sized to remain in the cache until all operations involving the data in the sub-block are complete.
  - The algorithms minimize Translation Buffer misses and page faults.
- Unity increment (or stride) is used wherever possible.

1.6 DXML's Accuracy

To obtain the highest performance from processors, the operations in DXML subprograms have been reordered to take advantage of processor-level parallelism.

As a result of this reordering, some subprograms may have an arithmetic evaluation order different from the conventional evaluation order. This difference in the order of floating point operations introduces round-off errors that imply the subprograms can return results that are not bit-for-bit identical to the results obtained when the computation is in the conventional order. However, for well-conditioned problems, these round-off errors should be insignificant.

Significant round-off errors in application code that is otherwise correct indicates that the problem is most likely not correctly conditioned. The errors could be the result of inappropriate scaling during the mathematical formulation of the problem or an unstable solution algorithm. Re-examine the mathematical analysis of the problem and the stability characteristics of the solution algorithm.

2

Preparing and Storing Program Data

This chapter discusses how to prepare and store program data. The data your program uses can be classified as one of two kinds: scalar or array data. The following topics are covered in this chapter:

- Scalar data and array data (Section 2.1, Section 2.2, and Section 2.3)
- Ways that scalar data can be stored in arrays (Section 2.4)
- Storing program data (Section 2.5)
- Fortran arrays (Section 2.6)

2.1 Scalar Data

A single data item, having one value, is known as a scalar. A scalar can be a single piece of data, or it can be an element in an array. A scalar can be passed to a subprogram as input or returned as output to your application program.

Scalar data can be of different types such as character, integer, single-precision real, double-precision real, single-precision complex, and double-precision complex.

2.1.1 Fortran Data Types in DXML

The Fortran data types that can be passed to DXML subprograms are shown in Table 2–1.

Table 2–1 Fortran Data Types

Data Type	Fortran Equivalent	Definition
Character	CHARACTER*1	A single character such as “n”, “t”, or “C”.
Character string	CHARACTER*(*)	A sequence of one or more characters.
Logical	LOGICAL*4	A logical value: TRUE or FALSE.
Integer	INTEGER*4	A number such as +8 or −136.
Single-precision real	REAL*4	A single-precision floating-point number.
Double-precision real	REAL*8	A double-precision floating-point number.
Single-precision complex	COMPLEX*8	Two floating-point numbers that together represent a complex number. Each number is REAL*4.
Double-precision complex	COMPLEX*16	Two floating-point numbers that together represent a complex number. Each number is REAL*8.

2.2 Data Representation in Fortran

When data is passed to a DXML subprogram, it must conform to the Fortran standard. The following sections describe the Fortran data types used in DXML subprograms and illustrate how data with these types is stored in memory.

2.2.1 CHARACTER*1 and CHARACTER*(*) Representation

A character string is a contiguous sequence of bytes in memory. A character string is specified by a descriptor containing two attributes: the address of the first byte of the string, and the length of the string in bytes.

2.2.2 LOGICAL Representation

Logical values are stored in four contiguous bytes, starting on an arbitrary byte boundary. The low-order bit (bit 0) determines the value. If bit 0 is set, the value is TRUE. If bit 0 is clear, the value is FALSE.

2.2.3 INTEGER*4 Representation

INTEGER*4 values are stored in two's complement representation and lie in the range -2147483648 to 2147483647. Each value is stored in four contiguous bytes, aligned on an arbitrary byte boundary.

2.2.4 REAL Floating-Point Representations

The Digital UNIX architecture defines two real floating-point data types that conform to IEEE standard data formats. DXML supports the following two formats:

- S_floating (REAL*4)

  The value of S_floating data is in the approximate normalized range $1.18 * 10^{-38}$ to $3.4 * 10^{38}$. The precision is approximately one part in 2^{23} or seven decimal digits.

- T_floating (REAL*8)

  The value of T_floating data is in the approximate range $2.23 * 10^{-308}$ to $1.8 * 10^{308}$. The precision is approximately one part in 2^{52} or 15 decimal digits.

2.2.5 COMPLEX Floating-Point Representations

The Digital UNIX architecture defines two complex floating-point data types. DXML supports the following two formats:

- U_floating complex (COMPLEX*8)
- V_floating double complex (COMPLEX*16)

2.3 Array Data

Array data can be thought of as many pieces of data grouped together in one unit called an array. Each piece of data is called an element of the array. Each element is a scalar and all elements are of the same data type.

An array can have one or more dimensions. A column or row of numbers is a one-dimensional array. A one-dimensional array is usually represented as a column or row of elements in parentheses. For example:

$$(3.1,\ 2.2,\ 1.3,\ 2.2,\ 3.1)$$

To locate a value in this array, you specify its position within the parentheses.

A table with rows and columns of figures is a two-dimensional array. A two-dimensional array is usually represented as rows and columns enclosed in square brackets:

$$\begin{bmatrix} 3.1 & 4.3 & 9.0 \\ 1.1 & 4.0 & 11.7 \end{bmatrix}$$

To locate a value in this array, you specify its position within the brackets by specifying its row number and then its column number.

From the user's perspective, an array is a group of contiguous storage locations associated with a single symbolic name, such as A, the array name. The individual storage locations (the array elements) are referred to by a number or a series of numbers in parentheses after the array name. A(1) is the first element of a one-dimensional array A. A(3,2) is the element in the third row and second column of a two-dimensional array A.

An array can contain data structures such as vectors and matrices. The way vector or matrix elements are separated in array storage is defined by stride and leading dimension arguments passed to DXML subprograms. See Section 2.6 for information on array storage techniques.

An array can be passed to a DXML subprogram as input, it can be returned as output to your application program, or it can be used by the subprogram as both input and output. In the latter case, some input data would be overwritten and therefore lost.

2.4 Requirements for Array Storage

Not all programming languages use the same storage techniques to store arrays. Some programming languages, such as Fortran, store arrays in memory in column-major order, storing the first column, then the second column, and so on. Other languages, such as C, store arrays in row-major order, storing the first row, then the second row, and so on.

DXML assumes that array elements are stored in column-major order when processing data. Use Fortran conventions as described in Section 2.6 for arrays passed to a DXML subprogram.

If you are calling DXML subprograms from languages other than Fortran, you must set up your data so that Fortran conventions for array storage can be applied. For information about calling subprograms from other languages, see Chapter 3.

2.5 Storing Program Data

DXML subprograms perform operations on two particular kinds of data structures: vectors and matrices. This section defines and describes vectors and matrices and discusses the various ways of storing vectors and matrices in arrays.

2.5.1 Vectors

A vector is a one-dimensional ordered collection of numbers, either real or complex. A real vector contains real numbers, and a complex vector contains complex numbers.

A vector can be represented symbolically as a column of numbers or as a row of numbers. For a column of numbers, use the following vector notation for a vector x with n elements:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

For a row of numbers, use the following vector notation for a vector y with n elements:

$$y = [\, y_1 \quad y_2 \quad y_3 \quad \ldots \quad y_n \,]$$

2.5.2 Transpose and Conjugate Transpose of a Vector

The transpose of a vector changes a column vector to a row vector or a row vector to a column vector. Use the following notation for a vector x and its transpose x^T:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad x^T = [\, x_1 \quad x_2 \quad x_3 \quad \ldots \quad x_n \,]$$

A complex number c is defined as $c = a + bi$, where a and b are real and $i = \sqrt{-1}$. The complex conjugate $\overline{c}$ is obtained by replacing i by $-i$:

$$\overline{c} = a - bi$$

If a vector x has elements that are complex numbers, the conjugate transpose of the vector x, denoted by x^H, is the vector that changes each element of $x_j = a_j + b_j i$ to its complex conjugate $\overline{x}_j = a_j - b_j i$ and then transposes it:

$$x^H = [\, \overline{x}_1 \quad \overline{x}_2 \quad \overline{x}_3 \quad \ldots \quad \overline{x}_n \,]$$

2.5.3 Storing a Vector in an Array

DXML provides various storage schemes to store vectors in arrays. You can find information about vector storage schemes in the following chapters:

- Defining and storing a vector in an array (See Section 6.2.1 and Section 6.2.2).
- Storing a sparse vector (See Section 7.2.1 and Section 7.2.2).
- Storing vectors for signal processing subprograms (See Section 11.1.2).

2.5.4 Matrices

A matrix is a two-dimensional ordered collection of numbers, either real or complex. A matrix A with m rows and n columns, an m by n matrix, is represented in the following way:

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

The elements of the matrix are represented as a_{ij} where $i = 1, ..., m$ and $j = 1, ..., n$. A square matrix with n rows and n columns is called a matrix of order n.

2.5.5 Transpose and Conjugate Transpose of a Matrix

The transpose of a matrix A, denoted by A^T, is formed by taking the ith row of A and making it the ith column of A^T. The columns of A become the rows of A^T. If A is an m by n matrix, A^T is an n by m matrix. Use the following notation for a matrix A and its transpose:

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

$$A^T = \begin{bmatrix} a_{11} & \cdots & a_{m1} \\ \vdots & \vdots & \vdots \\ a_{1n} & \cdots & a_{mn} \end{bmatrix}$$

The effect of transposing a matrix is to flip the matrix across its main diagonal. The element in row i, column j of A^T comes from row j, column i of A:

$$A^T_{ij} = A_{ji}$$

Taking the conjugate transpose of a matrix A that contains complex numbers is an operation on the matrix that changes each element of the matrix to its complex conjugate and then transposes the matrix:

$$A^H = \begin{bmatrix} \overline{a}_{11} & \cdots & \overline{a}_{n1} \\ \vdots & \vdots & \vdots \\ \overline{a}_{1n} & \cdots & \overline{a}_{mn} \end{bmatrix}$$

The effect of finding the complex conjugate of a matrix is to get the complex conjugate of each element of the matrix and then flip the matrix across its main diagonal. The element in row i, column j of A^H is the complex conjugate of the element in row j, column i of A:

$$A^H_{ij} = \overline{A}_{ji}$$

2.5.6 Storing a Matrix in an Array

DXML provides various storage schemes to store matrices in arrays. You can find information about matrix storage schemes in the following chapters:

- Defining and storing a matrix in an array (see Section 8.2.1).
- Symmetric and Hermitian matrices (see Section 8.2.2 and Section 8.2.3).
- Triangular matrix (see Section 8.2.4 and Section 8.2.5).
- General band matrices (see Section 8.2.6 and Section 8.2.7).
- Real symmetric band matrices and complex hermitian band matrices (see Section 8.2.8 and Section 8.2.9).
- Upper and lower triangular band matrices (see Section 8.2.10 and Section 8.2.11).
- Sparse matrices for iterative solvers (see Section 12.3 and Section 12.3.1).
- Sparse matrices stored using skyline storage scheme (see Section 13.4).

2.6 Fortran Arrays

A Fortran array can have from one to seven dimensions. An array is specified by the name of the array, the number of dimensions in the array, and the number of elements in each dimension. DXML operates on one- to three-dimensional arrays.

You must state the size of each array explicitly in your Fortran program. Use a DIMENSION statement, or preferably, a specific data type statement (such as REAL*4 or COMPLEX*8) for each array. Fortran arrays are always stored in memory as a linear sequence of values.

2.6.1 One-Dimensional Fortran Array Storage

A one-dimensional Fortran array is stored with its first element in the first storage location, the second element in the second storage location, and so on until the last element is in the last storage location.

For example, consider the one-dimensional array A with 4 elements shown in (2–1):

$$A = (A(1), A(2), A(3), A(4)) \qquad (2–1)$$

The array A has its elements stored as shown in Table 2–2.

Table 2–2 One-Dimensional Fortran Array Storage

Storage Location	Array Element
1	A(1)
2	A(2)
3	A(3)
4	A(4)

2.6.2 Two-Dimensional Fortran Array Storage

The elements of a two-dimensional array are stored column by column, so that the left subscripts vary most rapidly and the right subscripts vary least rapidly. The elements of the first column are stored, then the elements of suceeding columns are stored, until the elements in the last column are stored. This mode of storage is called column-major order.

For example, consider the two-dimensional array A with 12 elements shown in Table 2–3:

$$\mathrm{A} = \begin{bmatrix} \mathrm{A}(1,1) & \mathrm{A}(1,2) & \mathrm{A}(1,3) \\ \mathrm{A}(2,1) & \mathrm{A}(2,2) & \mathrm{A}(2,3) \\ \mathrm{A}(3,1) & \mathrm{A}(3,2) & \mathrm{A}(3,3) \\ \mathrm{A}(4,1) & \mathrm{A}(4,2) & \mathrm{A}(4,3) \end{bmatrix}$$

The array A has its elements stored as shown in Table 2–3.

Table 2–3 Two-Dimensional Fortran Array Storage

Storage Location	Array Element
1	A(1,1) (column 1 starts)
2	A(2,1)
3	A(3,1)
4	A(4,1)
5	A(1,2) (column 2 starts)
6	A(2,2)
7	A(3,2)
8	A(4,2)
9	A(1,3) (column 3 starts)
10	A(2,3)
11	A(3,3)
12	A(4,3)

2.6.3 Array Elements

All the elements of an array have the same data type: real or complex, single- or double-precision. The size of the storage locations for the elements depends on the data type of the array. Single-precision real (REAL*4, S_floating) data requires 4 bytes of storage; double-precision real (REAL*8, T_floating) data requires 8 bytes of storage.

Because a complex number is an ordered pair of two real numbers, (a, b) or $a + ib$, where $i = \sqrt{-1}$, storing a complex number requires two storage locations, one location for each part of the complex number. Single-precision complex (COMPLEX*8, S_floating complex) data requires 8 bytes of storage; double-precision complex (COMPLEX*16, T_floating complex) data requires 16 bytes of storage.

3
Coding an Application Program

As you code your application program, you need to remember certain information. This chapter provides information about the following topics:

- Guidelines for choosing the appropriate subprogram version (Section 3.1)
- Subprogram calling sequences and argument descriptions (Section 3.2)
- Calling subroutines and functions in Fortran (Section 3.3)
- Calling subroutines in other languages (Section 3.4)
- Error handling (Section 3.5)

3.1 Selecting the Appropriate Subprogram Version

DXML contains several versions of most subprograms. These versions are for operations performed in either real or complex arithmetic and either single-precision or double-precision arithmetic. You must use the DXML subprogram that operates on the type of data that you are using.

The naming convention for the subprogram identifies the type of data it works with. The first or second letter shows the type of data on which the subprogram operates. The letters shown (S, D, C, and Z) represent the following:

S Single-precision real data
D Double-precision real data
C Single-precision complex data
Z Double-precision complex data

See Chapter 5 for information on the compile and link procedure.

3.1.1 Subprogram Data Structure and Storage Method

DXML subprograms operate on vectors and matrices. For the subprograms that operate on matrices, different kinds of matrices use different storage schemes. When you use a DXML subprogram, consider the type of matrix used in your application and the data structure used to store it.

The storage methods described in Chapter 2 apply to Fortran and other languages that store arrays in column-major order. See Section 3.4 for techniques to use for languages that store arrays in row-major order.

3.1.2 Improving Performance

You have several options for improving the performance of your application subprogram:

- Use higher level BLAS subprograms where applicable. For example, use a Level 3 BLAS subprogram rather than a sequence of calls to Level 2 subprograms.

- Use subprograms that perform more than one computation rather than subprograms that perform a single computation.
- Use an increment or stride of 1. Performance is better if the elements of a vector are stored close to each other.
- In a few cases, the difference between two subprograms is that one performs scaling. Since performance is better when no scaling is done, use the subprogram without scaling whenever possible.

3.2 Calling Sequences

Each of the DXML subprograms has a specific calling sequence. The calling sequences for each subprogram are described in this manual.

These descriptions specify the correct syntax for the argument list; whether an argument is an input argument, an output argument, or both; required numerical values for arguments; and specific actions that might be taken by the subprogram.

Each subprogram is described using a structured format:

Name
Overview
Format
Function Value (if applicable)
Arguments
Description
Example

The **Arguments** section provides detailed information about each subprogram argument, such as the argument name, the Fortran data type, the information the argument passes to or returns from the subprogram, and the acceptable values of the argument. All arguments in the calling sequences are required arguments.

The terms *On entry* and *On exit* are used in each argument description to show whether the argument is an input argument, an output argument, or both:

- An input argument has a value on entry and is unchanged on exit.
  An input argument passes information from the application program to the subprogram.
- An output argument has no value on entry and is overwritten on exit.
  An output argument passes information from the subprogram back to the application program.
- An argument that is used for both input and output has a value on entry that is overwritten on exit by the output value. An argument used for both input and output passes information both to the subprogram and back to the application program. If you want to keep the input data, save it before calling the subprogram.

To avoid errors and the possible termination of a program's execution, be sure input data is of the correct type. Do not mix single-precision data and double-precision data. Also, character values must be one of the allowed characters and numeric values have to be within the specified range.

3.2.1 Passing of Arguments

In Fortran, a character argument can be longer than its corresponding dummy argument. For example, some BLAS subroutines have the arguments with the data type CHARACTER*1. One such argument is the **trans** argument, which is used to select the form of the input matrix. The value 'T' can be passed as 'TRANSPOSE'.

Some signal processing subroutines have arguments with the data type CHARACTER*(*). For example, the value 'F' for the argument **direction** can be passed as 'FORWARD'.

3.2.2 Implicit and Explicit Arguments

Arguments can be coded either implicitly or explicitly. For example, consider the Level 3 subprogram SSYMM. The following programs are equivalent.

```
REAL*4 A(20,20), B(30,40), C(30,50), ALPHA, BETA
SIDE = 'L'
UPLO = 'U'
M = 10
N = 20
ALPHA = 2.0
BETA = 3.0
LDA = 20
LDB = 30
LDC = 30
CALL SSYMM(SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
```

```
REAL*4 A(20,20), B(30,40), C(30,50)
CALL SSYMM('L','U',10,20,2.0,A,20,B,30,3.0,C,30)
```

3.2.3 Expanding Argument Lists

DXML provides the ability to expand the argument lists of each DXML subprogram from within an editor. This capability is convenient when creating or modifying code that frequently calls DXML subprograms and is available for the EMACS editor.

EMACS Editor

Use the following to expand the parameter list of a subroutine:

1. At the EMACS command line, load the file /usr/share/dxml.ml.
2. Enter ABBREV mode.
3. Enter the subroutine name and then press the SPACE bar.

3.3 Calling Subroutines and Functions in Fortran

A few DXML subprograms return a scalar. These subprograms are functions, and they are called as functions by coding a function reference. First, declare the data type of the returned value and the subprogram name, and then code the function reference, as shown in the following generic example:

```
INTEGER*4 function_value, subprogram_name
function_value = subprogram_name (argument_1,argument_2, . . .  ,argument_n)
```

For example, the Level 1 BLAS subprogram ISAMAX returns the index of the element of vector x having the largest absolute value:

```
   .
   .
   .
INTEGER*4 IAMAX, ISAMAX
   .
   .
   .
IAMAX=ISAMAX(N,X,INCX)
```

For the signal processing routines, the declaration of function type is done by including either "dxmldef.h" or "DXMLDEF.FOR" in your code. All arguments of function subprograms are input arguments, which are unchanged on exit. The value is returned to the function value.

Most subprograms return a vector or a matrix. These subprograms are subroutines and they are called as subroutines with a CALL statement.

```
CALL subroutine_name (argument_1,argument_2, . . .  ,argument_n)
```

For example, the call statement for the subroutine SSET looks like the following:

```
CALL SSET(N,A,X,INCX)
```

Each subroutine has an output argument that is overwritten on exit and contains the output information.

3.3.1 Fortran Program Example

The following is an example of a Fortran program that makes a call to saxpy:

```
integer n,incx,incy
real x(5),y(5),alpha

DATA  x /2.0,4.0,6.0,8.0,10.0/
DATA  y /5*1.0/
incx = 1
incy = 1
alpha = 2.0
n = 5

call saxpy (n,alpha,x,incx,y,incy)

write (6,*) y

stop
end
```

3.4 Using DXML from Non-Fortran Programming Languages

If your application involves only one-dimensional arrays (for example, one-dimensional FFTs), call DXML routines as described in the calling standard. However, two-dimensional (and higher) arrays are not covered by most calling standards. High-level languages have different ways of storing the elements of a two-dimensional matrix in a two-dimensional array.

DXML requires that arrays be stored in column-major order the way Fortran does. If you are writing applications in languages such as ADA or C, and you want to call DXML routines, you must consider how to ensure that array data is passed and processed correctly to obtain the highest performance.

The most direct and least error-prone method of calling DXML routines from another language, is to write a matrix transpose routine in that language, and to use transposed (column-stored) matrices in calls to DXML routines. The application program can subsequently transpose the results of calls to DXML when appropriate, to return to the row-major format of the calling language.

In some cases, using matrix identities is a shortcut, at a small cost in program complexity, as shown in the following two cases of the same operation (to compute the row-stored product of two matrices):

A and B are n by n matrices, stored in row-major order. To compute their product, $C = AB$:

1. Transpose A. Store it in the array AT.
2. Transpose B. Store it in the array BT.
3. Invoke the matrix-multiply routine to compute $D = AB$, column-stored.

   ```
   CALL SGEMM('N','N',N,N,N,1.0,AT,N,BT,N,0.0,D,N)
   ```

4. Transpose D to get C, the row-stored version of the result.

The shortcut uses the matrix identity, as in (3–1):

$$(B^T A^T)^T = AB \tag{3–1}$$

From a Fortran point of view, row-major storage of the matrices A and B is simply the matrices A^T and B^T. Therefore, the same row-major product can be computed using the following procedure:

1. Invoke the matrix-multiply routine.

   ```
   SGEMM('N','N',N,N,N,1.0,B,N,A,N,0.0,C,N)
   ```

This routine computes $B^T A^T$, column-stored, that is $C = (B^T A^T)^T = AB$, the row-stored result.

A third way to achieve the same product is somewhat slower because of the way memory is accessed:

1. Invoke the matrix-multiply routine to compute $(A^T)^T(B^T)^T$, column-stored, that is $D = C^T$.

   ```
   SGEMM('T','T',N,N,N,1.0,A,N,B,N,0.0,D,N)
   ```

2. Transpose D to get C, the row-stored version of the result.

3.4.1 Calling DXML from C Programs

In addition to the differences in storage of multi-dimensional arrays between Fortran and C, the following changes may also be required for a C program that calls DXML subroutines.

1. In Fortran, a two-dimensional array declared as:

   ```
   DOUBLE PRECISION A(LDA,N)
   ```

   is a contiguous piece of LDA × N double precision words in memory stored in a column major order. However, a similar declaration in C:

   ```
   double A[LDA][N]);
   ```

is LDA pointers to rows of length N. As these pointers can be anywhere, there is no guarantee that the rows of the array are contiguous. To interface a C program with a DXML routine that expects the memory locations to be contiguous, the array A should be declared as

```
double A[LDA*N];
```

or allocated as contiguous memory locations using malloc.

2. On Digital UNIX, you must append an underscore at the end of each subroutine or function subprogram name.

3.4.2 C Program Example

The following example illustrates the calculation of a matrix-vector product using the DXML routine DGEMV. The matrix is stored using the row-major storage of C. The column-major storage of the DXML Fortran routine is implicitly taken into account by calculating $A^T * x$, instead of $A * x$.

```
#include <stdio.h>
#include <stdlib.h>

#define dgemv dgemv_
#define max_size  10

extern void dgemv(char *, int *, int*, double *,
    double [], int *, double[], int *, double *,
    double[], int *);

int main()
{
    double *a, *b, *x;
    double alpha, beta;

    int length, lda, incx, i, j;

    length = 3;
    lda = max_size;
    incx = 1;

    alpha = 1.0;
    beta = 0.0;

    a = (double *)malloc(max_size*max_size*sizeof(double));
    b = (double *)malloc(max_size*sizeof(double));
    x = (double *)malloc(max_size*sizeof(double));

    for (i=0; i<length; i++)
    {
        for (j=0; j<length; j++)
            a[max_size*i+j] = (double)(2*i+j);
        x[i] = 1.0;
    }

    printf("  matrix:\n");
    for (i=0; i<length; i++)
    {
        for (j=0; j<length; j++)
            printf("   %6.2f  ", a[i*max_size+j]);
        printf("\n");
    }

    printf("\n  vector:\n");
    for (j=0; j<length; j++)
        printf("   %6.2f  \n", x[j]);

    dgemv("T", &length, &length, &alpha, a, &lda, x, &incx,
          &beta, b, &incx);
```

```
    printf("\n  matrix times vector: \n");
    for (i = 0; i < length; i++)
        printf ("    %.2f\n",b[i]);

    free (a);
    free (b);
    free (x);
}  /* end of main() */
```

Additional examples illustrating the use of DXML routines from a C program can be found on-line at /usr/examples/dxml.

3.5 Error Handling

Some errors are common to all portions of the DXML library. Other errors are unique to a particular library within DXML. This section describes general information about how errors are handled. See the appropriate chapters for more details about error handling for specific subprograms.

3.5.1 Internal Exceptions

Under certain extreme conditions, such as passing numbers on the verge of underflow or overflow, you can receive an internal exception error message. If this happens, you should check arguments for valid range.

When underflow occurs, the number is replaced by a zero, and execution continues. No error message is provided. When overflow occurs, you receive a message directed to the devices or files defined as *stdout* and *stderr*, and execution terminates. Check the subprogram arguments for valid range.

Internal exceptions also occur if you have a shorter array than that specified by a data length argument. In this case, you receive an error message, since you are trying to address a location outside the bounds of the array. Check the length of the arrays used or the value denoting the length of the arrays.

4
Using the Parallel Library

DXML includes a parallel library that can yield dramatic performance improvements on symmetric multiprocessing (SMP) systems. The DXML parallel library contains the same set of subprograms as the serial library except a subset of the subprograms has been parallelized. The parallel subprograms have names and calling parameters that are identical to the serial versions.

You do not have to make any changes in your source code to use the parallel library. Simply link your source code with the parallel library, and DXML automatically supplies the parallel subprograms.

Where parallel versions are unavailable, DXML supplies the serial versions. At run time you declare the number of processors on your SMP system with environment variables. For complete information on the procedure, see Section 5.3.

The following subcomponents of the DXML library contain key subprograms that have been modified to execute in parallel if run on SMP hardware:

- Level 2 BLAS
- Level 3 BLAS
- LAPACK
- Signal Processing
- Iterative Solvers
- Skyline Solvers

See Section 4.1 for a list of the parallel subprograms.

In some cases, a parallel subprogram in one subcomponent of the library benefits the subprograms in another subcomponent. For example, LAPACK subprograms that call the parallel versions of Level 2 or Level 3 BLAS subprograms show performance improvement. See Section 4.2.3 for details. Additionally, some Level 2 and Level 3 serial BLAS subprograms benefit by calling other BLAS subprograms that are parallel. See Section 4.2.2 for details.

4.1 DXML Parallel Subprograms

Table 4–1 lists parallel subprograms by subcomponent.

Table 4–1 Parallel Subprograms

Name	Subcomponent
{S,D,C,Z}GEMV	Level 2 BLAS (Chapter 8)
{S,D,C,Z}GEMM	Level 3 BLAS (Chapter 9)
{S,D,C,Z}GETRF	LAPACK manpage computational routines
{S,D,C,Z}POTRF	"
{S,D,C,Z}FFT	Signal Processing (Chapter 11)
{S,D,C,Z}FFT_INIT	"
{S,D,C,Z}FFT_APPLY	"
{S,D,C,Z}FFT_EXIT	"
{S,D,C,Z}FFT_2D	"
{S,D,C,Z}FFT_INIT_2D	"
{S,D,C,Z}FFT_APPLY_2D	"
{S,D,C,Z}FFT_EXIT_2D	"
{S,D,C,Z}FFT_3D	"
{S,D,C,Z}FFT_INIT_3D	"
{S,D,C,Z}FFT_APPLY_3D	"
{S,D,C,Z}FFT_EXIT_3D	"
DITSOL_DRIVER	Iterative Solver (Chapter 12)
DMATVEC_DRIVER	"
DPCONDL_DRIVER	"
DPCONDR_DRIVER	"
DITSOL_PBCG	"
DITSOL_PCG	"
DITSOL_PCGS	"
DITSOL_PGMRES	"
DITSOL_PLSCG	"
DITSOL_PTFQMR	"
DMATVEC_GENR	Iterative Solver (Chapter 12) continued

(continued on next page)

Table 4–1 (Cont.) Parallel Subprograms

Name	Subcomponent
DMATVEC_SDIA	"
DMATVEC_UDIA	"
DCREATE_DIAG_GENR	"
DCREATE_DIAG_SDIA	"
DCREATE_DIAG_UDIA	"
DAPPLY_DIAG_ALL	"
DCREATE_POLY_GENR	"
DCREATE_POLY_SDIA	"
DCREATE_POLY_UDIA	"
DAPPLY_POLY_GENR	"
DAPPLY_POLY_SDIA	"
DAPPLY_POLY_UDIA	"
DSSKYF	Skyline Solvers (Chapter 13)
DUSKYF	"

4.2 Performance Considerations for Parallel Execution

The following sections point out considerations about performance improvement using parallel subprograms.

4.2.1 Single Processor Systems

The parallel version of a subprogram may not run as quickly as the serial version on a single processor system. Overhead due to either the parallel processing software or changes in the implementation of the algorithm to make it parallelizable can cause reduced performance.

4.2.2 Level 2 and Level 3 BLAS Subprograms

Some Level 2 and most Level 3 BLAS serial subprograms use other subprograms that have parallel versions. For example, most Level 3 BLAS subprograms use the corresponding parallelized {S,D,C,Z}GEMM subprograms. Specifically, DSYMM uses DGEMM, CTRMM uses CGEMM, and STRSM uses both SGEMM and SGEMV. Moreover, some Level 3 BLAS subprograms use the corresponding parallelized {S,D,C,Z}GEMV subprograms.

In the case of Level 2 BLAS, many subprograms use {S,D,C,Z}GEMV. For example, ZTRSV uses ZGEMV. To the extent serial subprograms use parallel versions of other subprograms, they will run faster in a multi-processor environment, if you have linked your application to the parallel library.

4.2.3 LAPACK Subprograms

LAPACK subprograms that call the parallel versions of {S,D,C,Z}GEMM or {S,D,C,Z}GEMV will run faster in a multi-processor environment. For example, if you are using DGESV to solve a system of linear equations, linking your application to the parallel library results in improved performance.

4.2.4 Signal Processing Subprograms

All the Fast Fourier Transform subprograms in the signal processing subcomponent of the DXML library have parallel versions. The parallel versions run faster than the serial versions primarily when you use them with larger input data arrays, where the parallel initialization is small in comparison to the computations involved.

For example, using the parallel version of a one-dimensional FFT subprogram with an input array containing 8K or more elements generally results in a faster run time. When the input array contains less than 8K elements, the serial version often gives better results on an SMP system. DIGITAL recommends that you link your application with both libraries and compare the results.

4.2.5 Iterative Solver Subprograms

The performance improvement obtained by the use of parallel iterative solver subprograms is dependent not only on the system characteristics, but also on the properties of the linear system, such as the size of the matrix, the number of diagonals, the preconditioner used, and so on. In some cases, the performance can be improved by considering other options. For example, in the case of a symmetric matrix with few diagonals stored in the SDIA storage scheme, the parallel version may not yield the expected performance improvement. In this case, using the UDIA storage scheme, and trading off the extra memory required with the better parallelization properties of the UDIA scheme, may result in an overall reduction in the execution time.

4.2.6 Skyline Solver Subprograms

The factorization routines for symmetric and unsymmetric skyline matrices have been parallelized for all storage modes. Due to the overhead introduced during parallelization, these routines will run faster than the serial routines primarily when the problem size is large. Further, the performance improvement may depend on the profile of the matrix, with a uniform profile resulting in better load balancing, and consequently, better speedup on multi-processor systems.

5
Compiling and Linking an Application Program

Compiling and linking your application program with DXML is usually performed by a single command, the `f77` command for Fortran 77, the `f90` command for Fortran 90, and the `cc` command for C.

5.1 DXML Libraries

The DXML development kit provides three libraries:

- Serial shareable installed at /usr/shlib/libdxml.so
- Parallel shareable installed at /usr/shlib/libdxmlp.so
- Serial archive installed at /usr/lib/libdxml.a

The serial and the parallel shared libraries each contain a complete set of DXML routines called by identical routine names. In the parallel library some of these routines are parallelized to take advantage of additional CPUs in shared memory configurations; the remaining routines in the parallel library are the serial versions. See Section 4.1 for a complete list of DXML parallel routines.

The following sections show how to compile and link an application program to each of the DXML libraries. For more details about compiling and linking your application, see the reference (man) pages of `f77` and `cc`.

5.2 Compiling and Linking to the Serial Library

The following examples show how to compile and link to the serial shared library.

Fortran examples:

```
f77 program.f -ldxml
f90 program.f90 -ldxml
```

C example:

```
cc program.c -ldxml
```

5.3 Compiling and Linking to the Parallel Library

The following examples show how to compile and link to the parallel shared library.

Fortran examples:

```
f77 program.f -ldxmlp
f90 program.f90 -ldxmlp
```

C example:

```
cc program.c -ldxmlp
```

Parallel Execution Environment Variable

Before you can run programs compiled and linked with the parallel version of DXML, you must set the following environment variable:

OMP_NUM_THREADS <integer>

Replace <integer> with a positive number equal to the number of parallel threads to use. The number of parallel threads is usually equal to the number of processors on the system. Do not specify more parallel threads than there are processors to avoid performance degradation.

For example, to run a program on a multiprocessor system with three parallel threads and a thread stack size of 256K bytes, set the environment variables as follows:

```
setenv OMP_NUM_THREADS 3
```

5.4 Compiling and Linking to the Archive Library

The following examples show how to compile and link to the archive library.

Fortran examples:

```
f77 program.f /usr/lib/libdxml.a
```

```
f90 program.f90 /usr/lib/libdxml.a
```

C example:

```
cc  program.c /usr/lib/libdxml.a  -lfor
```

6

Using the Level 1 BLAS Subprograms and Extensions

The Level 1 BLAS (Basic Linear Algebra Subprograms) subprograms and the Extensions to the Level 1 BLAS subprograms perform vector-vector operations commonly occurring in many computational problems in linear algebra. This chapter provides information about the following topics:

- Operations performed by the Level 1 BLAS subprograms and their Extensions (Section 6.1)
- Vector storage (Section 6.2)
- Accuracy (Section 6.1.2)
- Subprogram naming conventions (Section 6.3)
- Subprogram summaries (Section 6.4)
- Calling Level 1 BLAS subprograms (Section 6.5)
- Arguments and definitions used in the subprograms (Sections 6.6 and 6.8)
- Error handling (Section 6.7)
- A look at a Level 1 Extensions subprogram and its use (Section 6.9)

A description of each Level 1 and Level 1 Extension subprogram follows this chapter.

6.1 Level 1 BLAS Operations

BLAS Level 1 operations work with vectors.

6.1.1 Types of Operations

The Level 1 BLAS subprograms and the Extensions usually operate on only one vector, but a few of the subprograms involve operations on two vectors. The subprograms can be classified into two types:

- Vector output is returned from a vector input.
  The results of these operations are independent of the order in which the elements of the vector are processed.
- Scalar output is returned from a vector input.
  The results of these reduction operations usually depend on the order in which the elements of the vector are processed.

6.1.2 Accuracy

Because of the efficient coding of the subprograms, in some cases, the results obtained might not match the results obtained using the conventional order of evaluation. Whenever this could happen, it is stated in the reference description for that subprogram.

6.2 Vector Storage

For the Level 1 BLAS and the Extensions subprograms, a vector is stored in a one-dimensional array.

6.2.1 Defining a Vector in an Array

A vector is usually stored in a one- or two-dimensional array. The elements of a vector are stored in order, but the elements are not necessarily contiguous.

If a vector is complex, each vector element has the form $a + bi$. For each complex element, two storage locations in succession are needed to store a and b. Therefore, storing a complex vector requires twice the number of storage locations as storing a real vector of the same precision.

An array can be much larger than a vector that is stored in the array. The storage of a vector is defined using three arguments in a DXML subprogram argument list:

- Vector length: Number of elements in the vector
- Vector location: Base address of the vector in the array
- Stride: Space, or increment, between consecutive elements in the array

These three arguments together specify which elements of an array are selected to become the vector.

6.2.1.1 Vector Length

To specify the length n of a vector, you specify an integer value for a length argument, such as the **n** argument. The length of a vector can be less than the length of the array that specifies the vector.

Vector length can also be thought of as the number of elements of the associated array that a subroutine will process. Processing continues until n elements have been processed.

6.2.1.2 Vector Location

The location of a vector is specified by the argument for the vector in the DXML subprogram argument list. Usually, an array such as X is declared, for example, X(1:20) or X(20). In this case, if you want to specify vector x as starting at the first element of an array X, the argument is specified as X(1) or X. If you want to specify vector x as starting at the fifth element of X, the argument is specified as X(5).

However, in an array X that is declared as X(3:20), with a lower bound and an upper bound given for the dimension, specifying vector x as starting at the fifth element of X means that the argument is specified as X(7).

For a two-dimensional array X that is declared as X(1:10,1:20) or X(10,20), specifying the vector x as starting at the seventh row and eleventh column of X means that the argument is specified as X(7,11).

Most of the examples shown in this manual assume that the lower bound in each dimension of an array is 1. Therefore, the lower bound is not specified, and the value of the upper bound is the number of elements in that dimension. So, a declaration of X(50) means X has 50 elements.

When vector elements are selected by the DXML subprogram, the starting point for the selection of vector elements is not always the location of the vector as specified by the argument passed to the DXML subroutine. Which element is the starting point for processing depends on whether the spacing parameter is positive, negative, or zero.

6.2.1.3 Stride of a Vector

The spacing parameter, called the increment or stride, indicates how to move from the starting point through the array to select the vector elements from the array. The increment is specified by an argument in the DXML subprogram argument list, such as the **incx** argument. Because one vector element does not necessarily immediately follow another, the increment specifies the size of the step (or stride) between elements.

The sign (+ or −) of the stride indicates the direction in which the vector elements are selected:

- Forward indexing

  The stride is positive. Vector elements are stored forward in the array in the order $x_1, x_2, ..., x_n$. As the vector element index increases, the array element index increases.

- Backward indexing

  The stride is negative. Vector elements are stored backward in the array, in the reverse order $x_n, x_{n-1}, ..., x_1$. As the vector element index increases, the array element index decreases.

The absolute value of the stride is the spacing between each element. An increment of 1 indicates that the vector elements are contiguous. An increment of 0 indicates that all the elements of a vector are selected from the same location in the array.

6.2.1.4 Selecting Vector Elements from an Array

DXML uses the stride to select elements from the array to construct the vector composed of these elements. The stride associates consecutive elements of the vector with equally spaced elements of the array.

When the stride is positive:

- The location specified by the argument for the vector is the location of the first element in the vector, x_1.
- The starting point for the selection of elements is at the first vector element.
- The indexing is forward, with the vector elements stored forward in the array.

For example, consider the array X declared as X(10) with X defined as shown in (6–1):

$$X = (10.0,\ 9.1,\ 8.2,\ 7.3,\ 6.4,\ 5.5,\ 4.6,\ 3.7,\ 2.8,\ 1.9) \qquad (6–1)$$

If you specify X, which means X(1), for the vector x, the first element processed is the first element of X, which is 10.0. If you specify X(3) for the vector x, the first element processed is the third element of X, which is 8.2.

To select the vector from the array, DXML adds the stride to the starting point and processes the number of elements you specify. For example, if the location of the vector is X(2), the stride is 2, and the vector length is 4, the vector is

$$x = (9.1,\ 7.3,\ 5.5,\ 3.7)$$

Processing begins at array element X(2), which is 9.1, and processing ends at array element X(8), which is 3.7.

If you are using a two-dimensional array for vector storage, remember that array elements are selected as they are stored, column by column. See Section 2.6.2, for this storage information.

For example, consider the array X declared as X(4,4) with X defined as shown in (6–2):

$$\mathrm{X} = \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 \\ 5.0 & 6.0 & 7.0 & 8.0 \\ 8.0 & 7.0 & 6.0 & 5.0 \\ 4.0 & 3.0 & 2.0 & 1.0 \end{bmatrix} \qquad (6\text{–}2)$$

If the location of the vector x is X(4,1), the stride is 3, and the vector length is 5, the vector is

$$x = (4.0,\ 7.0,\ 7.0,\ 4.0,\ 1.0)$$

When the increment is negative, vector elements are selected as follows:

- The location specified by the argument for the vector is the location of the last element in the vector, x_n.
- DXML calculates the starting point for the selection of elements by considering the location of the vector, the increment, and the number of elements to process.
- The indexing is backward. Vector elements are stored backwards in the array.

For example, consider the array X declared as X(12) with X defined as shown in (6–3):

$$\mathrm{X} = (1.0,\ 2.0,\ 3.0,\ 4.0,\ 5.0,\ 6.0,\ 7.0,\ 8.0,\ 9.0,\ 10.0,\ 11.0,\ 12.0) \qquad (6\text{–}3)$$

If the location of the vector is X(3), the increment is -2, and the vector length is 5, the vector is

$$x = (11.0,\ 9.0,\ 7.0,\ 5.0,\ 3.0)$$

In this case, processing begins at array element X(11), which is 11.0, and processing ends at array element X(3), which is 3.0.

When the increment is 0, the location specified by the argument such as the **x** argument, is the only array element used in the selection of the vector. Each element of the vector has the same value.

For example, consider the array X declared as X(6) with X defined as shown in (6–4):

$$\mathrm{X} = (1.0,\ 2.0,\ 3.0,\ 4.0,\ 5.0,\ 6.0) \qquad (6\text{–}4)$$

If the location of the vector is X(3), the increment is 0, and the number of elements to process is 5, the vector is

$$x = (3.0,\ 3.0,\ 3.0,\ 3.0,\ 3.0)$$

6.2.2 Storing a Vector in an Array

Suppose X is a real one-dimensional array of $ndim$ elements. Let vector x have length n and let $incx$ be the increment used to access the elements of vector x whose components x_i, $i = 1, \ldots, n$, are stored in X.

If $incx \geq 0$, and if the location of the vector is specified at the first element of the array, then x_i is stored in the array location as shown in (6–5):

$$\mathrm{X}(1 + (i - 1) * incx) \tag{6–5}$$

If $incx = 0$, and if the location of the vector is specified at the first element of the array, all the elements of the vector x are at the same array location, X(1).

If $incx < 0$, and if the location of the vector is specified at the first element of the array, then x_i is stored in the array location as shown in (6–6):

$$\mathrm{X}(1 + (n - i) * |incx|) \tag{6–6}$$

Therefore, $ndim$, the number of elements in the array, follows the condition shown in (6–7):

$$ndim \geq 1 + (n - 1) * |incx| \tag{6–7}$$

For the general case where the location of the vector in the array is at the point X(BP) rather than at the first element of the array, (6–8) or (6–9) can be used to find the position of each vector element x_i in a one-dimensional array.

For $incx \geq 0$,

$$X(\mathrm{BP} + (i - 1) * incx) \tag{6–8}$$

For $incx < 0$,

$$X(\mathrm{BP} + (n - i) * |incx|) \tag{6–9}$$

For example, suppose that $\mathrm{BP} = 3$, $ndim = 20$, and $n = 5$. Then a value of $incx = 2$ implies that x_1, x_2, x_3, x_4, and x_5 are stored in array elements X(3), X(5), X(7), X(9), and X(11). However, if $incx = -2$, then x_1, x_2, x_3, x_4, and x_5 are stored in array elements X(11), X(9), X(7), X(5), and X(3).

With a suitable choice for the location of a vector, you can operate on vectors that are embedded in other vectors or matrices. For example, consider an m by n matrix A, stored in an md by nd array.

The jth column of the matrix is a vector represented by:

base address:	A(1,*j*)
increment:	1
length:	*m*

The ith row of the matrix is a vector represented by:

base address:	A(*i*,1)
increment:	*md*
length:	*n*

The main diagonal of the matrix is a vector represented by:

base address:	A(1,1)
increment:	$md + 1$
length:	$\min(m,n)$

6.3 Naming Conventions

Table 6–1 shows the characters used in the names of the Level 1 BLAS and the Extensions and what the characters mean.

Table 6–1 Naming Conventions: Level 1 BLAS Subprograms

Character Group	Mnemonic	Meaning
First group	I	Computes the index of a particular vector element.
	No mnemonic	Computes the value of a particular vector element or performs an operation on one or more vectors.
Second group	S	Single-precision real data.
	D	Double-precision real data.
	C	Single-precision complex data.
	Z	Double-precision complex data.
Third group	A combination of letters at the end such as AMIN or AXPY	Type of computation such as Absolute (A) Minimum (MIN) or Scalar (A) Times a Vector (X) Plus (P) a Vector (Y).

For example, the name ICAMIN is the subprogram for computing the index of the element of a single-precision complex vector having the minimum absolute value.

6.4 Summary of Level 1 BLAS Subprograms

Tables 6–2 and 6–3 summarize the BLAS Level 1 subprograms and the extension subprograms.

Table 6–2 Summary of Level 1 BLAS Subprograms

Subprogram Name	Operation
ISAMAX	Calculates, in single-precision arithmetic, the index of the element of a real vector with maximum absolute value.
IDAMAX	Calculates, in double-precision arithmetic, the index of the element of a real vector with maximum absolute value.
ICAMAX	Calculates, in single-precision arithmetic, the index of the element of a complex vector with maximum absolute value.

(continued on next page)

Table 6–2 (Cont.) Summary of Level 1 BLAS Subprograms

Subprogram Name	Operation
IZAMAX	Calculates, in double-precision arithmetic, the index of the element of a complex vector with maximum absolute value.
SASUM	Calculates, in single-precision arithmetic, the sum of the absolute values of the elements of a real vector.
DASUM	Calculates, in double-precision arithmetic, the sum of the absolute values of the elements of a real vector.
SCASUM	Calculates, in single-precision arithmetic, the sum of the absolute values of the elements of a complex vector.
DZASUM	Calculates, in double-precision arithmetic, the sum of the absolute values of the elements of a complex vector.
SAXPY	Calculates, in single-precision arithmetic, the product of a real scalar and a real vector and adds the result to a real vector.
DAXPY	Calculates, in double-precision arithmetic, the product of a real scalar and a real vector and adds the result to a real vector.
CAXPY	Calculates, in single-precision arithmetic, the product of a complex scalar and a complex vector and adds the result to a complex vector.
ZAXPY	Calculates, in double-precision arithmetic, the product of a complex scalar and a complex vector and adds the result to a complex vector.
SCOPY	Copies a real, single-precision vector.
DCOPY	Copies a real, double-precision vector.
CCOPY	Copies a complex, single-precision vector.
ZCOPY	Copies a complex, double-precision vector.
SDOT	Calculates the inner product of two real, single-precision vectors.
DDOT	Calculates the inner product of two real, double-precision vectors.
DSDOT	Calculates the inner product of two real, single-precision vectors using double precision arithmetic operations and returns a double-precision result.
CDOTC	Calculates the conjugated inner product of two complex, single-precision vectors.
ZDOTC	Calculates the conjugated inner product of two complex, double-precision vectors.
CDOTU	Calculates the unconjugated inner product of two complex, single-precision vectors.
ZDOTU	Calculates the unconjugated inner product of two complex, double-precision vectors.
SDSDOT	Calculates the inner product of two real, single-precision vectors using double-precision arithmetic operations, adds the inner product result to a real single-precision scalar, and returns a single-precision value.

(continued on next page)

Table 6–2 (Cont.) Summary of Level 1 BLAS Subprograms

Subprogram Name	Operation
SNRM2	Calculates, in single-precision arithmetic, the square root of the sum of the squares of the elements of a real vector.
DNRM2	Calculates, in double-precision arithmetic, the square root of the sum of the squares of the elements of a real vector.
SCNRM2	Calculates, in single-precision arithmetic, the square root of the sum of the squares of the elements of a complex vector.
DZNRM2	Calculates, in double-precision arithmetic, the square root of the sum of the squares of the elements of a complex vector.
SROT	Applies a real Givens plane rotation to two real, single-precision vectors.
DROT	Applies a real Givens plane rotation to two real, double-precision vectors.
CROT	Applies a complex Givens plane rotation to two single-precision complex vectors.
ZROT	Applies a complex Givens plane rotation to two double-precision complex vectors.
CSROT	Applies a real Givens plane rotation to two complex, single-precision vectors.
ZDROT	Applies a real Givens plane rotation to two complex, double-precision vectors.
SROTM	Applies a modified Givens transformation to two real, single-precision vectors.
DROTM	Applies a modified Givens transformation to two real, double-precision vectors.
SROTG	Generates the real elements for a real, single-precision Givens plane rotation.
DROTG	Generates the elements for a real, double-precision Givens plane rotation.
CROTG	Generates the elements for a complex, single-precision Givens plane rotation.
ZROTG	Generates the real elements for a complex, double-precision Givens plane rotation.
SROTMG	Generates the real elements for a real, single-precision Givens transform.
DROTMG	Generates the real elements for a real, double-precision Givens transform.
SSCAL	Calculates, in single-precision arithmetic, the product of a real scalar and a real vector.

(continued on next page)

Table 6–2 (Cont.) Summary of Level 1 BLAS Subprograms

Subprogram Name	Operation
DSCAL	Calculates, in double-precision arithmetic, the product of a real scalar and a real vector.
CSCAL	Calculates, in single-precision arithmetic, the product of a complex scalar and a complex vector.
ZSCAL	Calculates, in double-precision arithmetic, the product of a complex scalar and a complex vector.
CSSCAL	Calculates, in single-precision arithmetic, the product of a real scalar and a complex vector.
ZDSCAL	Calculates, in double-precision arithmetic, the product of a real scalar and a complex vector.
SSWAP	Swaps the elements of two real, single-precision vectors.
DSWAP	Swaps the elements of two real, double-precision vectors.
CSWAP	Swaps the elements of two complex, single-precision vectors.
ZSWAP	Swaps the elements of two complex, double-precision vectors.

Table 6–3 Summary of Extensions to Level 1 BLAS Subprograms

Subprogram Name	Operation
ISAMIN	Calculates, in single-precision arithmetic, the index of the element of a real vector with minimum absolute value.
IDAMIN	Calculates, in double-precision arithmetic, the index of the element of a real vector with minimum absolute value.
ICAMIN	Calculates, in single-precision arithmetic, the index of the element of a complex vector with minimum absolute value.
IZAMIN	Calculates, in double-precision arithmetic, the index of the element of a complex vector with minimum absolute value.
ISMAX	Calculates, in single-precision arithmetic, the index of the real vector element with maximum value.
IDMAX	Calculates, in double-precision arithmetic, the index of the real vector element with maximum value.
ISMIN	Calculates, in single-precision arithmetic, the index of the real vector element with minimum value.
IDMIN	Calculates, in double-precision arithmetic, the index of the real vector element with minimum value.
SAMAX	Calculates, in single-precision arithmetic, the largest absolute value of the elements of a real vector.
DAMAX	Calculates, in double-precision arithmetic, the largest absolute value of the elements of a real vector.

(continued on next page)

Table 6–3 (Cont.) Summary of Extensions to Level 1 BLAS Subprograms

Subprogram Name	Operation
SCAMAX	Calculates, in single-precision arithmetic, the largest absolute value of the elements of a complex vector.
DZAMAX	Calculates, in double-precision arithmetic, the largest absolute value of the elements of a complex vector.
SAMIN	Calculates, in single-precision arithmetic, the smallest absolute value of the elements of a real vector.
DAMIN	Calculates, in double-precision arithmetic, the smallest absolute value of the elements of a real vector.
SCAMIN	Calculates, in single-precision arithmetic, the smallest absolute value of the elements of a complex vector.
DZAMIN	Calculates, in double-precision arithmetic, the smallest absolute value of the elements of a complex vector.
SMAX	Calculates, in single-precision arithmetic, the largest value of the elements of a real vector.
DMAX	Calculates, in double-precision arithmetic, the largest value of the elements of a real vector.
SMIN	Calculates, in single-precision arithmetic, the smallest value of the elements of a real vector.
DMIN	Calculates, in double-precision arithmetic, the smallest value of the elements of a real vector.
SNORM2	Calculates, in single-precision arithmetic, the square root of the sum of the squares of the elements of a real vector.
DNORM2	Calculates, in double-precision arithmetic, the square root of the sum of the squares of the elements of a real vector.
SCNORM2	Calculates, in single-precision arithmetic, the square root of the sum of the squares of the absolute value of the elements of a complex vector.
DZNORM2	Calculates, in double-precision arithmetic, the square root of the sum of the squares of the absolute value of the elements of a complex vector.
SNRSQ	Calculates, in single-precision arithmetic, the sum of the squares of the elements of a real vector.
DNRSQ	Calculates, in double-precision arithmetic, the sum of the squares of the elements of a real vector.
SCNRSQ	Calculates, in single-precision arithmetic, the sum of the squares of the absolute value of the elements of a complex vector.
DZNRSQ	Calculates, in double-precision arithmetic, the sum of the squares of the absolute value of the elements of a complex vector.

(continued on next page)

Table 6–3 (Cont.) Summary of Extensions to Level 1 BLAS Subprograms

Subprogram Name	Operation
SSET	For single-precision data, sets all the elements of a real vector equal to a real scalar.
DSET	For double-precision data, sets all the elements of a real vector equal to a real scalar.
CSET	For single-precision data, sets all the elements of a complex vector equal to a complex scalar.
ZSET	For double-precision data, sets all the elements of a complex vector equal to a complex scalar.
SSUM	Calculates, in single-precision arithmetic, the sum of the values of the elements of a real vector.
DSUM	Calculates, in double-precision arithmetic, the sum of the values of the elements of a real vector.
CSUM	Calculates, in single-precision arithmetic, the sum of the values of the elements of a complex vector.
ZSUM	Calculates, in double-precision arithmetic, the sum of the values of the elements of a complex vector.
SVCAL	Calculates, in single-precision arithmetic, the product of a real scalar and a real vector.
DVCAL	Calculates, in double-precision arithmetic, the product of a real scalar and a real vector.
CVCAL	Calculates, in single-precision arithmetic, the product of a complex scalar and a complex vector.
ZVCAL	Calculates, in double-precision arithmetic, the product of a complex scalar and a complex vector.
CSVCAL	Calculates, in single-precision arithmetic, the product of a real scalar and a complex vector.
ZDVCAL	Calculates, in double-precision arithmetic, the product of a real scalar and a complex vector.
SZAXPY	Calculates, in single-precision arithmetic, the product of a real scalar and a real vector and adds the result to a real vector.
DZAXPY	Calculates, in double-precision arithmetic, the product of a real scalar and a real vector and adds the result to a real vector.
CZAXPY	Calculates, in single-precision arithmetic, the product of a complex scalar and a complex vector and adds the result to a complex vector.
ZZAXPY	Calculates, in double-precision arithmetic, the product of a complex scalar and a complex vector and adds the result to a complex vector.

6.5 Calling Subprograms

The BLAS Level 1 and Extensions subprograms consist of both functions and subroutines:

- Functions

  Return a scalar
  Require a functions reference from a program
  Processing does not change arguments
  Documented with a **Function Value** section

- Subroutines

  Return a vector
  Require a CALL statement from a program
  Processing overwrites an output argument with the output vector
  No **Function Value** section

6.6 Argument Conventions

The subprograms use a list of arguments to specify the requirements and control the result of the subprogram. All arguments are required. The argument list is in the same order for each subprogram:

- Arguments that define the length of the input vectors
  The **n** argument specifies the length of the input vectors. The values $n < 0$, $n = 0$, and $n > 0$ are all allowed. However, for $n \leq 0$, either the output vector is unchanged or the function value is immediately set equal to a value specified previously.
- Arguments that specify the input scalar
  The **alpha** argument defines the input scalar.
- Arguments that describe the input and output vectors
  In addition to the **n** argument, the following arguments describe a vector:
  - The arguments **x**, **y**, and **z** define the location of the vectors x, y, and z in the array. In the usual case, the argument **x** specifies the location in the array as X(1), but the location can be specified at any other element of the array. An array can be much larger than the vector that it contains.
  - The arguments **incx**, **incy**, and **incz** provide the increment between the elements of the vector x, vector y, and vector z, respectively. The increment can be positive, negative, or zero. The vector can be stored forward or backward in the array.

Not every type of argument is needed by every subprogram.

6.7 Error Handling

The Level 1 BLAS subprograms assume that input parameters are correct and provide no feedback when problems occur. You must ensure that all input data for these subprograms is correct.

6.8 Definition of Absolute Value

Real subprograms that calculate an absolute value define the absolute value in the following way:

$$|x_j| = x_j \quad \text{if } x \text{ is positive}$$

$$|x_j| = -x_j \quad \text{if } x \text{ is negative}$$

Complex subprograms define the absolute value in a way that depends on the subprogram and its operations. The definitions are consistent with the definitions used in the BLAS Level 1 subprograms.

In some cases, the definition is the strict definition of the absolute value of a complex number, that is, the square root of the sum of the squares of the real part and the imaginary part:

$$|x_j| = \sqrt{a_j^2 + b_j^2} = \sqrt{\text{real}^2 + \text{imaginary}^2}$$

In other cases, the definition for the absolute value of a complex number is the absolute value of the real part plus the absolute value of the imaginary part:

$$|x_j| = |a_j| + |b_j| = |\text{real}| + |\text{imaginary}|$$

The subprogram name does not specify the definition used. Check the **Description** section of the subprogram reference description for the definition used for that subprogram.

6.9 A Look at a Level 1 Extensions Subprogram

To understand the meaning of the arguments, consider the subroutine SVCAL. SVCAL computes the product of a real scalar α and a real (n-element) vector x, and the result is returned in the vector y. SVCAL has the arguments **n**, **alpha**, **x**, **incx**, **y**, and **incy** as shown in the following code:

```
REAL*4 X(100), Y(200), ALPHA
INCX = 1
INCY = 2
ALPHA = 3.2
N = 100
CALL SVCAL(N,ALPHA,X,INCX,Y,INCY)
```

The argument **x** specifies the array X with 100 elements and specifies X(1) as the location of the vector x whose elements are embedded in X. Since **n** = 100, the vector also has 100 elements. The length of the array X is the same as the length of the vector x. The **incx** is positive, indicating the vector starts at the first array element. Because **incx** = 1, the vector elements are contiguous in the array. Since **incy** is 2, each element of the array is multiplied by 3.2 and stored in array Y, beginning at Y(1), in the locations Y(1), Y(3), Y(5), and so on.

As another example, if vector x has 20 elements, the starting point of the vector is X(1), and the elements are selected from the array X with an increment of 3, then the array X must have at least $(1 + (n-1)|incx|)$ or 58 elements to store the vector. The following code shows this case:

```
REAL*4 X(58), Y(200), ALPHA
INCX = 3
INCY = 2
ALPHA = 3.2
N = 20
CALL SVCAL(N,ALPHA,X,INCX,Y,INCY)
```

In this case, elements X(1), X(4), X(7), . . . , X(58) of the array are multiplied by 3.2 and are stored in array Y, beginning at Y(1), in the locations Y(1), Y(3), Y(5), . . . , Y(39).

When the increment is negative, the starting point for the vector selection is at the last element of the vector. Consider the following code where the increment is −2 and the starting point is specified as X(20):

```
REAL*4 X(100), Y(200), ALPHA
INCX = -2
INCY = 2
ALPHA = 3.2
N = 6
CALL SVCAL(N,ALPHA,X(20),INCX,Y,INCY)
```

The vector x has 6 elements. The elements selected to form the vector are X(30), X(28), X(26), X(24), X(22), and X(20). Each of these elements is multiplied by 3.2 and the results are stored in Y(1), Y(3), Y(5), Y(7), Y(9), and Y(11), respectively.

Level 1 BLAS Subprograms

This section provides descriptions of the Level 1 BLAS subprograms, combining real and complex versions of the subprograms.

A complex number has the form $a + bi$ where a is a real number called the real part, b is a real number called the imaginary part, and $i = \sqrt{-1}$. The vectors used in these subprograms represent a row or column of a matrix and can be indexed either forward or backward.

ISAMAX IDAMAX ICAMAX IZAMAX
Index of the Element of a Vector with Maximum Absolute Value

Format

I{S,D,C,Z}AMAX (n, x, incx)

Function Value

imax
integer*4
The index of the element of the vector x that is the largest in absolute value of all elements of the vector. If $n \leq 0$, **imax** returns the value 0.

Arguments

n
integer*4
On entry, the number of elements in the vector x.
On exit, **n** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n-1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** ≥ 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i-1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n-i) * |incx|)$.
On exit, **incx** is unchanged.

Description

These functions determine the first integer i among the elements of the vector x such that:

$$|x_i| = \max \left\{ |x_j|, j = 1, 2, ..., n \right\}$$

You can use these functions to obtain the pivots in Gaussian elimination.

For complex vectors, each element of the vector is a complex number. In these subprograms, the absolute value of a complex number is defined as the absolute value of the real part plus the absolute value of the imaginary part:

$$|x_j| = |a_j| + |b_j| = |\text{real}| + |\text{imaginary}|$$

If $incx < 0$, the result depends on how the program is processed. See the coding information in this document for a discussion of the possible results. If $incx = 0$, the computation is a time-consuming way of setting $imax = 1$.

Example

```
INTEGER*4 IMAX, N, INCX, ISAMAX
REAL*4 X(40)
INCX = 2
N = 20
IMAX = ISAMAX(N,X,INCX)
```

This Fortran code shows how to compute the index of a real vector element with maximum absolute value.

SASUM DASUM SCASUM DZASUM
Sum of the Absolute Values

Format

{S,D}ASUM (n, x, incx)

SCASUM (n, x, incx)

DZASUM (n, x, incx)

Function Value

sum
real*4 | real*8
The sum of the absolute values of the elements of the vector x.
If $n \leq 0$, **sum** returns the value 0.0.

Arguments

n
integer*4
On entry, the number of elements in the vector x.
On exit, **n** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n-1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** ≥ 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i-1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n-i) * |incx|)$.
On exit, **incx** is unchanged.

Description

The SASUM and DASUM functions compute the sum of the absolute values of the elements of a real vector x:

$$\sum_{i=1}^{n} |x_i| = |x_1| + |x_2| + \ldots + |x_n|$$

SCASUM and DZASUM compute the sum of the absolute values of the real and imaginary parts of the elements of a complex vector x:

$$\sum_{i=1}^{n}(|a_i| + |b_i|) = (|a_1| + |b_1|) + (|a_2| + |b_2|) + \ldots + (|a_n| + |b_n|)$$

where $x_i = (a_i, b_i)$ and $|x_i| = |a_i| + |b_i| = |\text{real}| + |\text{imaginary}|$

If $incx < 0$, the result is identical to using $|incx|$. If $incx = 0$, the computation is a time-consuming way of setting $\text{sum} = nx_1$.

Because of the efficient coding of these routines, rounding errors can cause the final result to differ from the result computed by a sequential evaluation of the sum of the elements of the vector.

Example

```
INTEGER*4 N, INCX
REAL*4 X(20), SUM, SASUM
INCX = 1
N = 20
SUM = SASUM(N,X,INCX)
```

This Fortran code shows how to compute the sum of the absolute values of the elements of the vector x.

SAXPY DAXPY CAXPY ZAXPY
Vector Plus the Product of a Scalar and a Vector

Format

{S,D,C,Z}AXPY (n, alpha, x, incx, y, incy)

Arguments

n
integer*4
On entry, the number of elements in the vectors x and y.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar multiplier α for the elements of the vector x.
On exit, **alpha** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** ≥ 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i - 1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n - i) * |incx|)$.
On exit, **incx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$, containing the elements of the vector y.

On exit, if $n \leq 0$ or $\alpha = 0$, **y** is unchanged. If $n > 0$, **y** is overwritten; y_i is replaced by $y_i + \alpha x_i$.

incy
integer*4
On entry, the increment for the array Y.
If **incy** > 0, vector y is stored forward in the array, so that y_i is stored in location $Y(1 + (i - 1) * incy)$.
If **incy** < 0, vector y is stored backward in the array, so that y_i is stored in location $Y(1 + (n - i) * |incy|)$.
On exit, **incy** is unchanged.

Description

The _AXPY functions compute the following scalar-vector product and sum:

$$y \leftarrow \alpha x + y$$

where α is a scalar, and x and y are vectors.

If any element of x or the scalar α share a memory location with an element of y, the results are unpredictable.

If $incx = 0$, the computation is a time-consuming way of adding the constant αx_1 to all the elements of y. The following chart shows the operation that results from the interaction of the values for arguments **incx** and **incy**:

	incx = 0	**incx** $\neq$ **0**
incy = 0	$y_1 = y_1 + n\alpha x_1$	$y_1 = y_1 + \sum_{i=1}^{n} \alpha x_i$
incy $\neq$ **0**	$y_i = y_i + \alpha x_1$	$y_i = y_i + \alpha x_i$

Example

```
INTEGER*4 N, INCX, INCY
REAL*4 X(20), Y(20), ALPHA
INCX = 1
INCY = 1
ALPHA = 2.0
N = 20
CALL SAXPY(N,ALPHA,X,INCX,Y,INCY)
```

This Fortran code shows how all elements of the real vector x are multiplied by 2.0, added to the elements of the real vector y, and the vector y is set equal to the result.

SCOPY DCOPY CCOPY ZCOPY
Copy of a Vector

Format

{S,D,C,Z}COPY (n, x, incx, y, incy)

Arguments

n
integer*4
On entry, the number of elements in the vector x.
On exit, **n** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** ≥ 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i - 1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n - i) * |incx|)$.
On exit, **incx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$.
On exit, if $n \leq 0$, **y** is unchanged. If $n > 0$, **y** is overwritten; y_i is replaced by x_i.

incy
integer*4
On entry, the increment for the array Y.
If **incy** ≥ 0, vector y is stored forward in the array, so that y_i is stored in location $Y(1 + (i - 1) * incy)$.
If **incy** < 0, vector y is stored backward in the array, so that y_i is stored in location $Y(1 + (n - i) * |incy|)$.
On exit, **incy** is unchanged.

Description

The _COPY subprograms copy the elements of the vector x to the vector y, performing the following operation:

$$y_i \leftarrow x_i$$

If $incx = 0$, each y_i is set to x_1. Therefore, you can use $incx = 0$ to initialize all elements to a constant.

If $incy = 0$, the computation is a time-consuming way of setting $y_1 = x_n$, the last referenced element of the vector x.

If $incy = -incx$, the vector x is stored in reverse order in y. In this case, the call format is as follows:

```
CALL SCOPY (N,X,INCX,Y,-INCX)
```

If any element of x shares a memory location with an element of y, the results are unpredictable, except for the following special case. It is possible to move the contents of a vector up or down within itself and not cause unpredictable results even though the same memory location is shared between input and output. To do this when $i > j$, call the subroutine with $incx = incy > 0$ as follows:

```
CALL SCOPY (N,X(I),INCX,X(J),INCX)
```

The call to SCOPY moves elements of the array X $x(i), x(i + 1 * incx), ..., x(i + (n - 1) * incx)$ to new elements of the array X $x(j), x(j + 1 * incx), ..., x(j + (n - 1) * incx)$. If $i < j$, specify a negative value for $incx$ and $incy$ in the call to the subroutine, as follows. The parts that do not overlap are unchanged.

```
CALL SCOPY (N,X(I),-INCX,X(J),-INCX)
```

Examples

1.
```
INTEGER*4 N, INCX, INCY
REAL*4 X(20), Y(20)
INCX = 1
INCY = 1
N = 20
CALL SCOPY(N,X,INCX,Y,INCY)
```

   The preceding Fortran code copies a vector x to a vector y.

2.
```
CALL SCOPY(N,X,-2,X(3),-2))
```

   The preceding call moves the contents of X(1),X(3),X(5), . . . , X(2N-1) to X(3),X(5), . . . , X(2N+1) and leaves the vector x unchanged.

3.
```
CALL SCOPY(99,X(2),1,X,1))
```

   The preceding call moves the contents of X(2),X(3), . . . , X(100) to X(1),X(2), . . . , X(99) and leaves x_{100} unchanged.

4.
```
CALL SCOPY(N,X,1,Y,-1))
```

   The preceding call moves the contents of X(1),X(2),X(3), . . . , X(N) to Y(N),Y(N-1), . . . , Y.

SDOT DDOT DSDOT CDOTC ZDOTC CDOTU ZDOTU
Inner Product of Two Vectors

Format

{S,D}DOT (n, x, incx, y, incy)

DSDOT (n, x, incx, y, incy)

{C,Z}DOT{C,U} (n, x, incx, y, incy)

Function Value

dotpr
real*4 | real*8 | complex*8 | complex*16
The dot product of the two vectors x and y.

For real vectors, if $n \leq 0$, **dotpr** returns the value 0.0.
For complex vectors, if $n \leq 0$, **dotpr** returns (0.0, 0.0).

Arguments

n
integer*4
On entry, the number of elements in the vectors x and y.
On exit, **n** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n-1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** ≥ 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i-1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n-i) * |incx|)$.
On exit, **incx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array Y of length at least $(1 + (n-1) * |incx|)$, containing the elements of the vector y.
On exit, **y** is unchanged.

incy
integer*4
On entry, the increment for the array Y.
If **incy** ≥ 0, vector y is stored forward in the array, so that y_i is stored in location $Y(1 + (i-1) * incy)$.
If **incy** < 0, vector y is stored backward in the array, so that y_i is stored in location $Y(1 + (n-i) * |incy|)$.
On exit, **incy** is unchanged.

Description

SDOT, DDOT, and DSDOT compute the dot product of two real vectors. CDOTC and ZDOTC compute the conjugated dot product of two complex vectors. CDOTU and ZDOTU compute the unconjugated dot product of two complex vectors.

SDOT, DDOT, DSDOT are functions that compute the dot product of two n-element real vectors, x and y:

$$x \cdot y = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n$$

The order of operations is different from the order in a sequential evaluation of the dot product. The final result can differ from the result of a sequential evaluation. The DSDOT function accepts single-precision input vectors but uses double-precision operations to compute a double-precision result.

CDOTC and ZDOTC are functions that compute the conjugated dot product of two complex vectors, x and y, that is, the complex conjugate of the first vector is used to compute the dot product.

Each element x_j of the vector x is a complex number and each element y_j of the vector y is a complex number. The conjugated dot product of two complex vectors, x and y, is expressed as follows:

$$\overline{x} \cdot y = \sum_{i=1}^{n} \overline{x}_i y_i = \overline{x}_1 y_1 + \overline{x}_2 y_2 + \ldots + \overline{x}_n y_n$$

For example, x and y each have two complex elements:

$$x = (1+i, 2-i), y = (3+i, 3+2i)$$

The conjugate of vector x is $\overline{x} = (1-i, 2+i)$, and the dot product is:

$$\overline{x} \cdot y = (1-i)(3+i) + (2+i)(3+2i) = (4-2i) + (4+7i) = (8+5i))$$

CDOTU and ZDOTU compute the unconjugated dot product of two complex vectors. The unconjugated dot product of two complex vectors, x and y, is expressed as follows:

$$x \cdot y = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n$$

For example, for the same complex vectors x and y:

$$x \cdot y = (1+i)(2+i) + (2-i)(3+2i) = (1+3i) + (8+i) = 9+4i$$

Example

```
INTEGER*4 INCX, INCY
REAL*4 X(20), Y(20), DOTPR, SDOT
INCX = 1
INCY = 1
N = 20
DOTPR = SDOT(N,X,INCX,Y,INCY)
```

This Fortran code shows how to compute the dot product of two vectors, x and y, and return the result in dotpr.

```
INTEGER*4 INCX, INCY
COMPLEX*8 X(20), Y(20), DOTPR, CDOTU
INCX = 1
INCY = 1
N = 20
DOTPR = CDOTU(N,X,INCX,Y,INCY)
```

This Fortran code shows how to compute the unconjugated dot product of two complex vectors, x and y, and return the result in dotpr.

SDSDOT
Product of Scaled Vector and Vector

Format

SDSDOT (n, alpha, x, incx, y, incy)

Function Value

dotpr
real*4
The sum of the scalar **alpha** and the dot product of vectors x and y.

If $n \leq 0$, **dotpr** returns the value in **alpha**.

Arguments

n
integer*4
On entry, the number of elements in the vectors x and y.
On exit, **n** is unchanged.

alpha
real*4
On entry, a scalar addend summed with the dot product of vectors x and y.
On exit, **alpha** is unchanged.

x
real*4
On entry, a one-dimensional array X of length at least $(1 + (n-1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** ≥ 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i-1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n-i) * |incx|)$.
On exit, **incx** is unchanged.

y
real*4
On entry, a one-dimensional array Y of length at least $(1 + (n-1) * |incx|)$, containing the elements of the vector y.
On exit, **y** is unchanged.

incy
integer*4
On entry, the increment for the array Y.
If **incy** ≥ 0, vector y is stored forward in the array, so that y_i is stored in location $Y(1 + (i-1) * incy)$.
If **incy** < 0, vector y is stored backward in the array, so that y_i is stored in location $Y(1 + (n-i) * |incy|)$.
On exit, **incy** is unchanged.

Description

SDSDOT computes the dot product of two single-precision real vectors, x and y, using double-precision arithmetic. The resulting dot product is added to the single-precision scalar value **alpha** to produce the single precision return value.

Example

```
INTEGER*4 INCX,INCY
REAL*8 X(20),Y(20),ALPHA
REAL*4 DOTPR,SDSDOT
INCX = 1
INCY = 1
ALPHA = 0.4
N = 20
DOTPR = SDSDOT(N,ALPHA,X,INCX,Y,INCY)
```

SNRM2 DNRM2 SCNRM2 DZNRM2
Square Root of Sum of the Squares of the Elements of a Vector

Format

{S,D}NRM2 (n, x, incx)

SCNRM2 (n, x, incx)

DZNRM2 (n, x, incx)

Function Value

e_norm
real*4 | real*8
The Euclidean norm of the vector x, that is, the square root of the conjugated dot product of x with itself.

If $n \leq 0$, **e_norm** returns the value 0.0.

Arguments

n
integer*4
On entry, the number of elements in the vector x.
On exit, **n** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n-1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** ≥ 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i-1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n-i) * |incx|)$.
On exit, **incx** is unchanged.

Description

SNRM2 and DNRM2 compute the Euclidean norm of a real vector; SCNRM2 and DZNRM2 compute the Euclidean norm of a complex vector. The Euclidean norm is the square root of the conjugated dot product of a vector with itself.

For real vectors:

$$\sqrt{\sum_{i=1}^{n} x_i^2} = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}$$

For complex vectors:

$$\sqrt{\sum_{i=1}^{n} \overline{x}_i * x_i} = \sqrt{(\overline{x}_1 * x_1) + (\overline{x}_2 * x_2) + ... + (\overline{x}_n * x_n)}$$

The order of operations is different from the order in a sequential evaluation of the Euclidean norm. The final result can differ from the result of a sequential evaluation.

If $incx < 0$, the result is identical to using $|incx|$. If $incx = 0$, the computation is a time-consuming way of setting $\text{e\_norm} = \sqrt{nx_1^2}$.

Example

```
INTEGER*4 INCX, N
REAL*4 X(20), E_NORM
INCX = 1
N = 20
E_NORM = SNRM2(N,X,INCX)
```

This Fortran code shows how to compute the Euclidean norm of a real vector.

SROT DROT CROT ZROT CSROT ZDROT
Apply Givens Plane Rotation

Format

{S,D,C,Z}ROT (n, x, incx, y, incy, c, s)

CSROT (n, x, incx, y, incy, c, s)

ZDROT (n, x, incx, y, incy, c, s)

Arguments

n
integer*4
On entry, the number of elements in the vectors x and y.
On exit, **n** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.
On exit, if $n \leq 0$ or if **c** is 1.0 and **s** is 0.0, **x** is unchanged. Otherwise, **x** is overwritten; X contains the rotated vector x.

incx
integer*4
On entry, the increment for the array X.
If **incx** ≥ 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i - 1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n - i) * |incx|)$.
On exit, **incx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$. Y contains the n elements of the vector y.

On exit, if $n \leq 0$ or if **c** is 1.0 and **s** is 0.0, **y** is unchanged. Otherwise, **y** is overwritten; Y contains the rotated vector y.

incy
integer*4
On entry, the increment for the array Y.
If **incy** ≥ 0, vector y is stored forward in the array, so that y_i is stored in location $Y(1 + (i - 1) * incy)$.
If **incy** < 0, vector y is stored backward in the array, so that y_i is stored in location $Y(1 + (n - i) * |incy|)$.
On exit, **incy** is unchanged.

c
real*4 | real*8
On entry, the first rotation element, that is, the cosine of the angle of rotation. The argument **c** is the first rotation element generated by the _ROTG subroutines.
On exit, **c** is unchanged.

s
real*4 | real*8 | complex*8 | complex*16
On entry, the second rotation element, that is, the sine of the angle of rotation. The argument **s** is the second rotation element generated by the _ROTG subroutines.
On exit, **s** is unchanged.

Description

SROT and DROT apply a real Givens plane rotation to each element in the pair of real vectors, x and y. CSROT and ZDROT apply a real Givens plane rotation to elements in the complex vectors, x and y. CROT and ZROT apply a complex Givens plane rotation to each element in the pair of complex vectors x and y.

The cosine and sine of the angle of rotation are c and s, respectively, and are provided by the BLAS Level 1 _ROTG subroutines.

The Givens plane rotation for SROT, DROT, CSROT, and ZDROT follows:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

The elements of the rotated vector x are $x_i \leftarrow cx_i + sy_i$.
The elements of the rotated vector y are $y_i \leftarrow -sx_i + cy_i$.

The Givens plane rotation for CROT and ZROT follows:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} c & s \\ -\overline{s} & c \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

The elements of the rotated vector x are $x_i \leftarrow cx_i + sy_i$.
The elements of the rotated vector y are $y_i \leftarrow -\overline{s}x_i + cy_i$.

If $n \leq 0$ or if $c = 1.0$ and $s = 0.0$, x and y are unchanged. If any element of x shares a memory location with an element of y, the results are unpredictable.

These subroutines can be used to introduce zeros selectively into a matrix.

Example

```
INTEGER*4 INCX, N
REAL X(20,20), A, B, C, S
INCX = 20
N = 20
A = X(1,1)
B = X(2,1)
CALL SROTG(A,B,C,S)
CALL SROT(N,X,INCX,X(2,1),INCX,C,S)
```

This Fortran code shows how to rotate the first two rows of a matrix and zero out the element in the first column of the second row.

SROTG DROTG CROTG ZROTG
Generate Elements for a Givens Plane Rotation

Format

{S,D,C.Z}ROTG (a, b, c, s)

Arguments

a
real*4 | real*8 | complex*8 | complex*16
On entry, the first element of the input vector.
On exit, **a** is overwritten with the rotated element r.

b
real*4 | real*8 | complex*8 | complex*16
On entry, the second element of the input vector. On exit, for SROTG and DROTG, **b** is overwritten with the reconstruction element z. For CROTG and ZROTG, **b** is unchanged.

c
real*4 | real*8
On entry, an unspecified variable.
On exit, **c** is overwritten with the first rotation element, that is, the cosine of the angle of rotation.

s
real*4 | real*8 | complex*8 | complex*16
On entry, an unspecified variable.
On exit, **s** is overwritten with the second rotation element, that is, the sine of the angle of rotation.

Description

The _ROTG subroutines construct a Givens plane rotation that eliminates the second element of a two-element vector and can be used to introduce zeros selectively into a matrix.

Using a and b to represent elements of an input real vector, the SROTG and DROTG functions calculate the elements c and s of an orthogonal matrix such that:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} * \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

Using a and b to represent elements of an input complex vector, the CROTG and ZROTG functions calculate the elements real c and complex s of an orthogonal matrix such that:

$$\begin{bmatrix} c & s \\ -\overline{s} & c \end{bmatrix} * \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

A real Givens plane rotation is constructed for values a and b by computing values for r, c, s, and z, as follows:

$r = p\sqrt{a^2 + b^2}$

$p = \text{SIGN}(a)$ if $|a| > |b|$
$p = \text{SIGN}(b)$ if $|a| \leq |b|$

$c = \frac{a}{r}$ if r is not equal to 0
$c = 1$ if $r = 0$

$s = \frac{b}{r}$ if r is not equal to 0
$s = 0$ if $r = 0$

$z = s$ if $|a| > |b|$
$z = \frac{1}{c}$ if $|a| \leq |b|$, c is not equal to 0, and r is not equal to 0.
$z = 1$ if $|a| \leq |b|$, $c = 0$, and r is not equal to 0.
$z = 0$ if $r = 0$

SROTG and DROTG can use the reconstruction element z to store the rotation elements for future use. The quantities c and s are reconstructed from z as follows:

For $|z| = 1, c = 0.0$ and $s = 1.0$

For $|z| < 1$, $c = \sqrt{1 - z^2}$ and $s = z$

For $|z| > 1, c = \frac{1}{z}$ and $s = \sqrt{1 - c^2}$

A complex Givens plane rotation is constructed for values a and b by computing values for real c, complex s and complex r, as follows:

$p = \sqrt{|a|^2 + |b|^2}$
$q = \frac{a}{|a|}$
$r = qp$ if $|a|$ is not equal to 0.
$r = b$ if $|a|$ is equal to 0.

$c = \frac{|a|}{p}$ if $|a|$ is not equal to 0.
$c = 0$ if $|a|$ is equal to 0.

$s = \frac{q\bar{b}}{p}$ if $|a|$ is not equal to 0.
$s = (1.0, 0.0)$ if $|a|$ is equal to 0.

The absolute value used in the previous definitions corresponds to the strict definition of the absolute value of a complex number.

The arguments **c** and **s** are passed to the _ROT subroutines.

Example

```
REAL*4 A, B, C, S
CALL SROTG(A,B,C,S)
```

This Fortran code shows how to generate the rotation elements for a vector of elements a and b.

SROTM DROTM
Apply Modified Givens Transformation

Format

{S,D}ROTM (n, x, incx, y, incy, param)

Arguments

n
integer*4
On entry, the number of elements in the vectors x and y.
On exit, **n** is unchanged.

x
real*4 | real*8
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.
On exit, if $n \leq 0$ or if PARAM(1) = (-2.0), **x** is unchanged. Otherwise, **x** is overwritten; X contains the rotated vector x.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i - 1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n - i) * |incx|)$.
On exit, **incx** is unchanged.

y
real*4 | real*8
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$. Y contains the n elements of the vector y.

On exit, if $n \leq 0$ or if PARAM(1) = (-2.0), **y** is unchanged. Otherwise, **y** is overwritten; Y contains the rotated vector y.

incy
integer*4
On entry, the increment for the array Y.
If **incy** > 0, vector y is stored forward in the array, so that y_i is stored in location $Y(1 + (i - 1) * incy)$.
If **incy** < 0, vector y is stored backward in the array, so that y_i is stored in location $Y(1 + (n - i) * |incy|)$.
On exit, **incy** is unchanged.

param
real*4 | real*8
On entry, an array defining the type of transform matrix H used:

PARAM(1) specifies the flag characteristic: -1.0, 0.0, 1.0, -2.0
PARAM(2) specifies H_{11} value
PARAM(3) specifies H_{21} value
PARAM(4) specifies H_{12} value
PARAM(5) specifies H_{22} value

On exit, **param** is unchanged.

Description

SROTM and DROTM apply a modified Givens transform to each element in the pair of real vectors, x and y, using the transformation matrix H as follows:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = H * \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Depending on the value of PARAM(1), the transformation matrix H is defined as follows:

- PARAM(1)= -1.0
$$\begin{pmatrix} H_{11} & H_{12} \\ H_{21} & H_{22} \end{pmatrix}$$
- PARAM(1)= 0.0
$$\begin{pmatrix} 1.0 & H_{12} \\ H_{21} & 1.0 \end{pmatrix}$$
- PARAM(1)= 1.0
$$\begin{pmatrix} H_{11} & 1.0 \\ -1.0 & H_{22} \end{pmatrix}$$
- PARAM(1)= -2.0
$$\begin{pmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{pmatrix}$$

The array PARAM is generated by a call to the routine _ROTMG.

Results are unpredictable if either **incx** or **incy** are zero.

Example

```
      INTEGER*4 INCA, N
      REAL A(10,10), D(10), SPARAM(5)
C
      INCA = 10
C
C INITIALIZE D TO 1.0
C
      DO I = 1, 10
         D(I) = 1.0
      END DO
C
C FOR EACH ROW OF THE MATRIX, ELIMINATE TO UPPER TRIANGULAR FORM
C
      DO I = 2, 10
C
C ELIMINATE A(I,J) USING ELEMENT A(J,J)
C
         JEND = I-1
         DO J = 1, JEND
            N = 10-J
            CALL SROTMG(D(J),D(I),A(J,J),A(I,J),SPARAM)
            CALL SROTM(N,A(J,J+1),INCA,A(I,J+1),INCA,SPARAM)
         ENDDO
C
      END DO
C
C APPLY ACCUMULATED SCALE FACTORS TO THE ROWS OF A
C
      DO I = 1, 10
         CALL SSCAL(11-I, SQRT(D(I)), A(I,I), INCA)
      END DO
```

This Fortran code shows how to reduce a 10 by 10 matrix to upper triangular form using the routine SROTMG and SROTM.

SROTMG DROTMG
Generate Elements for a Modified Givens Transform

Format

{S,D}ROTMG (d1, d2, x1, y1, param)

Arguments

d1
real*4 | real*8
On entry, the first scale factor for the modified Givens transform.
On exit, **d1** is updated.

d2
real*4 | real*8
On entry, the second scale factor for the modified Givens transform.
On exit, **d2** is updated.

x1
real*4 | real*8
On entry, the first element x_1 of the input vector.
On exit, **x1** is overwritten with the rotated element.

y1
real*4 | real*8
On entry, the second element y_1 of the input vector.
On exit, **y1** is unchanged.

param
real*4 | real*8
On entry, **param** is unspecified.
On exit, **param** contains an array defining the transform matrix H as follows:

PARAM(1) specifies the flag characteristic: -1.0, 0.0, 1.0, -2.0
PARAM(2) specifies H_{11} value
PARAM(3) specifies H_{21} value
PARAM(4) specifies H_{12} value
PARAM(5) specifies H_{22} value

Description

The _ROTMG subroutines construct a modified Givens transform that eliminates the second element of a two-element vector and can be used to introduce zeros selectively into a matrix. These routines use the modification due to Gentleman of the Givens plane rotations. This modification eliminates the square root from the construction of the plane rotation and reduces the operation count when the modified Givens rotation, rather than the standard Givens rotations are applied. In most applications, the scale factors d_1 and d_2 are initially set to 1 and then modified by _ROTMG as necessary.

Given real a and b in factored form:

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} d_1^{\frac{1}{2}} & 0 \\ 0 & d_2^{\frac{1}{2}} \end{bmatrix} * \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

SROTMG and DROTMG construct the modified Givens plane rotation, $\overline{d_1}$, $\overline{d_2}$ and

$$H = \begin{bmatrix} H_{11} & H_{12} \\ H_{21} & H_{22} \end{bmatrix}$$

such that

$$\begin{bmatrix} \frac{1}{d_1^2} & 0 \\ 0 & \frac{1}{d_2^2} \end{bmatrix} * H * \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = G * \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where G is a 2 by 2 Givens plane rotation matrix which annihilates b, and where H is chosen for numerical stability and computational efficiency.

The routine _ROTM applies the matrix H, as constructed by _ROTMG, to a pair of real vectors, x and y, each with n elements, as follows:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = H * \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

These vectors may be either rows or columns of matrices and the indexing of the vectors may be either forwards or backwards.

Depending on the value of IPARAM(1), the matrix H is defined as follows:

- PARAM(1)= -1.0

$$\begin{pmatrix} H_{11} & H_{12} \\ H_{21} & H_{22} \end{pmatrix}$$

- PARAM(1)= 0.0

$$\begin{pmatrix} 1.0 & H_{12} \\ H_{21} & 1.0 \end{pmatrix}$$

- PARAM(1)= 1.0

$$\begin{pmatrix} H_{11} & 1.0 \\ -1.0 & H_{22} \end{pmatrix}$$

- PARAM(1)= -2.0

$$\begin{pmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{pmatrix}$$

Note

The routines _ROTMG and _ROTM perform similar tasks to the routines _ROTG and _ROT, which construct and apply the standard Givens plane rotations. The modified Givens rotations reduce the operation count of constructing and applying the rotations at the cost of increased storage to represent the rotations.

Example

See the example for SROTM.

SSCAL DSCAL CSCAL ZSCAL, CSSCAL ZDSCAL
Product of a Scalar and a Vector

Format

{S,D,C,Z}SCAL (n, alpha, x, incx)

CSSCAL (n, alpha, x, incx)

ZDSCAL (n, alpha, x, incx)

Arguments

n
integer*4
On entry, the number of elements in the vector x.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar value used to multiply the elements of vector x.
On exit, **alpha** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.
On exit, if $n \leq 0$ or $\alpha = 1.0$, then **x** is unchanged. Otherwise, **x** is overwritten; x_i is replaced by αx_i.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i - 1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n - i) * |incx|)$.
If **incx** = 0, only the first element in the array is scaled.
On exit, **incx** is unchanged.

Description

These routines perform the following operation:

$$x \leftarrow \alpha x$$

SSCAL and DSCAL scale the elements of a real vector by computing the product of the vector and a real scalar αx. CSCAL and ZSCAL scale the elements of a complex vector by computing the product of the vector and a complex scalar α. CSSCAL and ZDCAL scale the elements of a complex vector by computing the product of the vector and a real scalar α.

If $n \leq 0$ or α = 1.0, x is unchanged.

If $incx < 0$, the result is identical to using $|incx|$.

If α = 0.0 or (0.0, 0.0), the computation is a time-consuming way of setting all elements of the vector x equal to zero. Use the BLAS Level 1 Extensions subroutines _SET to set all the elements of a vector to a scalar.

The _SCAL routines are similar to the BLAS Level 1 Extensions subroutines _VCAL routines, but the _VCAL routines use an output vector different from the input vector.

Example

```
INTEGER*4 INCX, N
COMPLEX*8 X(20), ALPHA
INCX = 1
ALPHA = (2.0, 1.0)
N = 20
CALL CSCAL(N,ALPHA,X,INCX)
```

This Fortran code shows how to scale a complex vector x by the complex scalar (2.0, 1.0).

SSWAP DSWAP CSWAP ZSWAP
Exchange the Elements of Two Vectors

Format

{S,D,C,Z}SWAP (n, x, incx, y, incy)

Arguments

n
integer*4
On entry, the number of elements in the vector x.
On exit, **n** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.
On exit, if $n \leq 0$, **x** is unchanged. If $n > 0$, **x** is overwritten; the elements in the array X that are the vector x are overwritten by the vector y.

incx
integer*4
On entry, the increment for the array X.
If **incx** ≥ 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i - 1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n - i) * |incx|)$.
On exit, **incx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$.

On exit, if $n \leq 0$, **y** is unchanged. If $n > 0$, **y** is overwritten; the elements in the array Y that are the vector y are overwritten by the vector x.

incy
integer*4
On entry, the increment for the array Y.
If **incy** ≥ 0, vector y is stored forward in the array, so that y_i is stored in location $Y(1 + (i - 1) * incy)$.
If **incy** < 0, vector y is stored backward in the array, so that y_i is stored in location $Y(1 + (n - i) * |incy|)$.
On exit, **incy** is unchanged.

Description

These subroutines swap n elements of the vector x with n elements of vector y:

$$x \leftrightarrow y$$

If any element of x shares a memory location with an element of y, the results are unpredictable.

If $n \leq 0$, x and y are unchanged.

You can use these subroutines to invert the storage of elements of a vector within itself. If $incx > 0$, each element x_i is moved from location $\mathrm{X}(1 + (i - 1) * incx)$ to location $\mathrm{X}(1 + (n - i) * incx)$. The following code fragment inverts the storage of elements of a vector within itself:

```
NN = N/2
LHALF = 1+(N-NN)*INCX
CALL SSWAP(NN,X,INCX,X(LHALF),-INCX)
```

Example

```
INTEGER*4 INCX, INCY, N
REAL*4 X(20), Y(20)
INCX = 1
INCY = 1
N = 20
CALL SSWAP(N,X,INCX,Y,INCY)
```

The preceding Fortran code swaps the contents of vectors x and y.

```
INCX = 1
INCY = -1
N = 50
CALL SSWAP(N,X,INCX,X(51),INCY)
```

The preceding Fortran code inverts the order of storage of the elements of x within itself; that is, it moves $x_1, \ldots, x_{100}$ to $x_{100}, \ldots, x_1$.

Level I BLAS Extensions Subprograms

This section provides descriptions of the Level 1 BLAS Extensions subprograms, combining real and complex versions of the subprograms.

A complex number has the form $a + bi$ where a is a real number called the real part, b is a real number called the imaginary part, and $i = \sqrt{minus > 1}$. The vectors used in these subprograms represent a row or column of a matrix and can be indexed either forward or backward.

ISAMIN IDAMIN ICAMIN IZAMIN
Index of the Element of a Vector with Minimum Absolute Value

Format

I{S,D,C,Z}AMIN (n, x, incx)

Function Value

imin
integer*4

The index of the first element of the vector x such that $\mathrm{X}(1+(imin-1)*|incx|)$ is the smallest in absolute value of all elements of the vector. If $n \leq 0$, **imin** returns the value 0.

Arguments

n
integer*4
On entry, the number of elements in the vector x.
On exit, **n** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1+(n-1)*|incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $\mathrm{X}(1+(i-1)*incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $\mathrm{X}(1+(n-i)*|incx|)$.
If **incx** = 0, only the first element is accessed.
On exit, **incx** is unchanged.

Description

These subprograms compute the index of the element of a vector having the minimum absolute value. They determine the first integer i of the vector x such that:

$$|x_i| = \min\left\{|x_j|, j = 1, 2, ..., n\right\}$$

For complex vectors, each element x_j is a complex number. In this subprogram, the absolute value of a complex number is defined as the absolute value of the real part of the complex number plus the absolute value of the imaginary part of the complex number:

$$|x_j| = |a_j| + |b_j| = |\mathrm{real}| + |\mathrm{imaginary}|$$

If $incx = 0$, the computation is a time-consuming way of setting $imin = 1$.

Example

```
INTEGER*4 N, INCX, IMIN, ISAMIN
REAL*4 X(40)
INCX = 2
N = 20
IMIN = ISAMIN(N,X,INCX)
```

This Fortran example shows how to compute the index of the vector element with minimum absolute value.

ISMAX IDMAX
Index of the Real Vector Element with Maximum Value

Format

I{S,D}MAX (n, x, incx)

Function Value

imax
integer*4

The index of the first element of the real vector x such that $X(1 + (imax - 1) * |incx|)$ is the largest of all elements of the vector.
If $n \leq 0$, **imax** returns the value 0.

Arguments

n
integer*4
On entry, the number of elements in the vector x.
On exit, **n** is unchanged.

x
real*4 | real*8
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the real vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i - 1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n - i) * |incx|)$.
If **incx** = 0, only the first element is accessed.
On exit, **incx** is unchanged.

Description

ISMAX and IDMAX determine the first integer i of vector x such that:

$$x_i = \max \{x_j, j = 1, 2, ..., n\}$$

If $incx = 0$, the computation is a time-consuming way of setting $imax$ = 1.

Example

```
INTEGER*4 N, INCX, IMAX, ISMAX
REAL*4 X(40)
INCX = 2
N = 20
IMAX = ISMAX(N,X,INCX)
```

This Fortran example shows how to compute the index of the vector element with maximum value.

ISMIN IDMIN
Index of the Real Vector Element with Minimum Value

Format

I{S,D}MIN (n, x, incx)

Function Value

imin
integer*4

The index of the first element of the real vector x such that $\mathrm{X}(1+(imin-1)*|incx|)$ is the smallest of all elements in the vector. If $n \leq 0$, **imin** returns the value 0.

Arguments

n
integer*4
On entry, the number of elements in the vector x.
On exit, **n** is unchanged.

x
real*4 | real*8
On entry, a one-dimensional array X of length at least $(1+(n-1)*|incx|)$, containing the elements of the real vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $\mathrm{X}(1+(i-1)*incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $\mathrm{X}(1+(n-i)*|incx|)$.
If **incx** = 0, only the first element is accessed.
On exit, **incx** is unchanged.

Description

ISMIN and IDMIN determine the first integer i such that:

$$x_i = \min\left\{x_j, j = 1, 2, ..., n\right\}$$

If $incx = 0$, the computation is a time-consuming way of setting $imin = 1$.

Example

```
INTEGER*4 N, INCX, IMIN, ISMIN
REAL*4 X(40)
INCX = 2
N = 20
IMIN = ISMIN(N,X,INCX)
```

This Fortran example shows how to compute the index of the vector element with minimum value.

SAMAX DAMAX SCAMAX DZAMAX
Maximum Absolute Value

Format

{S,D}AMAX (n, x, incx)

SCAMAX (n, x, incx)

DZAMAX (n, x, incx)

Function Value

amax
real*4 | real*8
The element of the vector with the largest absolute value. If $n \leq 0$, **amax** returns the value 0.0.

Arguments

n
integer*4
On entry, the number of elements in the vector x.
On exit, **n** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n-1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i-1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n-i) * |incx|)$.
If **incx** = 0, only the first element is accessed.
On exit, **incx** is unchanged.

Description

These functions determine the largest absolute value of the elements of a vector:

$$\max \left\{ |x_j|, j = 1, 2, ..., n \right\}$$

For complex vectors, each element is a complex number. In this subprogram, the absolute value of a complex number is defined as the absolute value of the real part of the complex number plus the absolute value of the imaginary part of the complex number:

$$|x_j| = |a_j| + |b_j| = |\text{real}| + |\text{imaginary}|$$

If $incx < 0$, the result is identical to using $|incx|$. If $incx = 0$, the computation is a time-consuming way of setting $amax = |x_1|$.

Example

```
REAL*4 SCAMAX, AMAX
INTEGER*4 N, INCX
COMPLEX*8 X(40)
INCX = 2
N = 20
AMAX = SCAMAX(N,X,INCX)
```

This Fortran example shows how to compute the element with the largest absolute value.

SAMIN DAMIN SCAMIN DZAMIN
Minimum Absolute Value

Format

{S,D}AMIN (n, x, incx)

SCAMIN (n, x, incx)

DZAMIN (n, x, incx)

Function Value

amin
real*4 | real*8
The element of the vector x with the smallest absolute value.
If $n \leq 0$, **amin** = 0.

Arguments

n
integer*4
On entry, the number of elements in the vector x.
On exit, **n** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n-1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i-1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n-i) * |incx|)$.
If **incx** = 0, only the first element is accessed.
On exit, **incx** is unchanged.

Description

These functions determine the smallest absolute value of the elements of a vector x:

$$\max \left\{ |x_j|, j = 1, 2, ..., n \right\}$$

For complex vectors, each element x_j is a complex number. In this subprogram, the absolute value of a complex number is defined as the absolute value of the real part of the complex number plus the absolute value of the imaginary part of the complex number:

$$|x_j| = |a_j| + |b_j| = |\text{real}| + |\text{imaginary}|$$

If $incx < 0$, the result is identical to using $|incx|$. If $incx = 0$, the computation is a time-consuming way of setting $amin = |x_1|$.

Example

```
INTEGER*4 N, INCX
REAL*4 X(400), AMIN, SAMIN
INCX = 3
N = 100
AMIN = SAMIN(N,X,INCX)
```

These Fortran examples show how to compute the element with the smallest absolute value.

SMAX DMAX
Largest Element in a Real Vector

Format

{S,D}MAX (n, x, incx)

Function Value

wmax
real*4 | real*8
The largest value among the elements of the real vector x.

If $n \leq 0$, **wmax** = 0.

Arguments

n
integer*4
On entry, the number of elements in real vector x.
On exit, **n** is unchanged.

x
real*4 | real*8
On entry, a one-dimensional array X of length at least $(1 + (n-1) * |incx|)$, containing the elements of real vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i-1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n-i) * |incx|)$.
If **incx** = 0, only the first element is accessed.
On exit, **incx** is unchanged.

Description

SMAX and DMAX are functions that determine the largest value among the real elements of a vector x:

$$\max \{x_j, j = 1, 2, ..., n\}$$

If $incx < 0$, the result is identical to using $|incx|$. If $incx = 0$, the computation is a time-consuming way of setting $wmax = x_1$.

Example

```
INTEGER*4 N, INCX
REAL*4 X(40), WMAX, SMAX
INCX = 2
N = 20
WMAX = SMAX(N,X,INCX)
```

This Fortran example shows how to compute the largest value of the elements of a vector x.

SMIN DMIN
Minimum Value of the Elements of a Real Vector

Format

{S,D}MIN (n, x, incx)

Function Value

wmin
real*4 | real*8
The smallest value of the elements of the real vector x.
If $n \leq 0$, **wmin** = 0.

Arguments

n
integer*4
On entry, the number of elements n in the vector x.
On exit, **n** is unchanged.

x
real*4 | real*8
On entry, a one-dimensional array X of length at least $(1 + (n-1) * |incx|)$. X contains the n elements of the real vector x.

On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i-1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n-i) * |incx|)$.
If **incx** = 0, only the first element is accessed.
On exit, **incx** is unchanged.

Description

SMIN and DMIN are functions that determine the smallest value of the elements of a vector x:

$$\min \{x_j, j = 1, 2, ..., n\}$$

If $incx < 0$, the result is identical to using $|incx|$. If $incx = 0$, the computation is a time-consuming way of setting $wmin = x_1$.

Example

```
INTEGER*4 N, INCX
REAL*4 X(40), WMIN, SMIN
INCX = 1
N = 30
WMIN = SMIN(N,X,INCX)
```

This Fortran example shows how to compute the smallest value of the elements of a vector x.

SNORM2 DNORM2 SCNORM2 DZNORM2
Square Root of Sum of the Squares of the Elements of a Vector

Format

{S,D}NORM2 (n, x, incx)

SCNORM2 (n, x, incx)

DZNORM2 (n, x, incx)

Function Value

sum
real*4 | real*8 | complex*8 | complex*16
The Euclidean norm of the vector x, that is, the square root of the sum of the squares of the elements of a real vector or the square root of the sum of the squares of the absolute value of the elements of the complex vector. If $n \leq 0$, **sum** = 0.0.

Arguments

n
integer*4
On entry, the number of elements of the vector x.
On exit, **n** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n-1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i-1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n-i) * |incx|)$.
If **incx** = 0, only the first element is accessed.
On exit, **incx** is unchanged.

Description

SNORM2 and DNORM2 compute the Euclidean norm of a real vector x. The Euclidean norm is the square root of the sum of the squares of the elements of the vector:

$$\sqrt{\sum_{i=1}^{n} x_i^2}$$

SCNORM2 and DZNORM2 compute the square root of the sum of the squares of the absolute value of the elements of a complex vector x:

$$\sqrt{\sum_{i=1}^{n} |x_i|^2}$$

For complex vectors, each element x_j is a complex number. In this subprogram, the absolute value of a complex number is defined as the square root of the sum of the squares of the real part and the imaginary part:

$$|x_j| = \sqrt{a_j^2 + b_j^2} = \sqrt{\text{real}^2 + \text{imaginary}^2}$$

If $incx < 0$, the result is identical to using $|incx|$. If $incx = 0$, the computation is a time-consuming way of setting $\text{sum} = \sqrt{nx_1^2}$ for real operations, and $\text{sum} = |x_i|\sqrt{n}$ for complex operations.

Because of efficient coding, rounding errors can cause the final result to differ from the result computed by a sequential evaluation of the Euclidean norm.

Unlike the _NRM2 and __NRM2 subprograms in BLAS Level 1, the _NORM2 and __NORM2 subprograms do not perform any special scaling to ensure that intermediate results do not overflow or underflow. Therefore, these routines must use an input vector x so that:

$$|\sqrt{min}| \leq |x_i| \leq |\sqrt{max}|$$

The largest value of x must not overflow when it is squared; the smallest value must not underflow when it is squared.

Example

```
INTEGER*4 N, INCX
REAL*4 X(20), SUM, SNORM2
INCX = 1
N = 20
SUM = SNORM2(N,X,INCX)
```

This Fortran example shows how to compute the Euclidean norm of the vector x.

SNRSQ DNRSQ SCNRSQ DZNRSQ
Sum of the Squares of the Elements of a Vector

Format

{S,D}NRSQ (n, x, incx)

SCNRSQ (n, x, incx)

DZNRSQ (n, x, incx)

Function Value

sum
real*4 | real*8
The sum of the squares of the elements of the real vector x.
The sum of the squares of the absolute value of the elements of the complex vector x.

If $n \leq 0$, **sum** returns the value 0.0.

Arguments

n
integer*4
On entry, the number of elements of the vector x.
On exit, **n** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n-1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i-1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n-i) * |incx|)$.
If **incx** = 0, only the first element is accessed.
On exit, **incx** is unchanged.

Description

SNRSQ and DNRSQ compute the sum of squares of the elements of a real vector. SCNRSQ and DZNRSQ compute the sum of squares of the absolute value of the elements of a complex vector.

SNRSQ and DNRSQ compute the total value of the square roots of each element in the real vector x:

$$\sum_{j=1}^{n} x_j^2$$

SCNRSQ and DZNRSQ compute the total value of the square roots of each element in the complex vector x, using the absolute value of each element:

$$\sum_{j=1}^{n} |x_j|^2$$

For complex vectors, each element x_j is a complex number. In this subprogram, the absolute value of a complex number is defined as the square root of the sum of the square of the real part and the square of the imaginary part:

$$|x_j| = \sqrt{a_j^2 + b_j^2} = \sqrt{\text{real}^2 + \text{imaginary}^2}$$

If $incx < 0$, the result is identical to using $|incx|$. If $incx = 0$, the computation is a time-consuming way of setting $\text{sum} = nx_1^2$.

Because of efficient coding, rounding errors can cause the final result to differ from the result computed by a sequential evaluation of the sum of the squares of the elements of the vector. Use these functions to obtain the square of the Euclidean norm instead of squaring the result obtained from the Level 1 routines SNRM2 and DNRM2. The computation is more accurate.

Example

```
INTEGER*4 N, INCX
REAL*4 X(20), SUM, SNRSQ
INCX = 1
N = 20
SUM = SNRSQ(N,X,INCX)
```

This Fortran example shows how to compute the sum of the squares of the elements of the vector x.

SSET DSET CSET ZSET
Set All Elements of a Vector to a Scalar

Format

{S,D,C,Z}SET (n, alpha, x, incx)

Arguments

n
integer*4
On entry, the number of elements in the vector.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar α value.
On exit, **alpha** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.

On exit, if $n \leq 0$, **x** is unchanged. If $n > 0$, **x** is overwritten by the updated x.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i - 1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n - i) * |incx|)$.
If **incx** = 0, only the first element is accessed.
On exit, **incx** is unchanged.

Description

The _SET subroutines change all elements of a vector to the same scalar value; each element x_i is replaced with α.

$$x_i \leftarrow \alpha$$

If $incx < 0$, the result is identical to using $|incx|$. If $incx = 0$, the computation is a time-consuming way of setting $x_1 = \alpha$.

Example

```
INTEGER*4 N, INCX
REAL*4 X(200), ALPHA
INCX = 2
ALPHA = 2.0
N = 50
CALL SSET(N,ALPHA,X,INCX)
```

This Fortran example shows how to set all elements of the vector x equal to 2.0.

SSUM DSUM CSUM ZSUM
Sum of the Values of the Elements of a Vector

Format

{S,D,C,Z}SUM (n, x, incx)

Function Value

sum
real*4 | real*8 | complex*8 | complex*16
The total of the values of the elements in the vector x. If $n \leq 0$, **sum** returns the value 0.0.

Arguments

n
integer*4
On entry, the number of elements in the vector x.
On exit, **n** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the n elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i - 1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n - i) * |incx|)$.
If **incx** = 0, only the first element is accessed.
On exit, **incx** is unchanged.

Description

The _SUM subprograms compute the total value of the elements of a vector, performing the following operation:

$$\sum_{i=1}^{n} x_i$$

Because of efficient coding, rounding errors can cause the final result to differ from the result computed by a sequential evaluation of the sum of the elements of the vector.

If $incx < 0$, the result is identical to using $|incx|$. If $incx = 0$, the computation is a time-consuming way of setting $\text{sum} = nx_1$.

Example

```
INTEGER*4 N, INCX
REAL*4 X(200), SUM, SSUM
INCX = 2
N = 50
SUM = SSUM(N,X,INCX)
```

This Fortran example shows how to compute the sum of the values of the elements of the vector x.

SVCAL DVCAL CVCAL ZVCAL CSVCAL, ZDVCAL
Product of a Scalar and a Vector

Format

{S,D,C,Z}VCAL (n, alpha, x, incx, y, incy)

CSVCAL (n, alpha, x, incx, y, incy)

ZDVCAL (n, alpha, x, incx, y, incy)

Arguments

n
integer*4
On entry, the number of elements of the vector x.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar multiplier α.
On exit, **alpha** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i - 1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n - i) * |incx|)$.
If **incx** = 0, only the first element is accessed.
On exit, **incx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$.
On exit, if $n \leq 0$, **y** is unchanged. If $n > 0$, **y** is overwritten; each element y_i is replaced by αx_i.

incy
integer*4
On entry, the increment for the array Y.
If **incy** ≥ 0, vector y is stored forward in the array, so that y_i is stored in location $Y(1 + (i - 1) * incy)$
If **incy** < 0, vector y is stored backward in the array, so that y_i is stored in location $Y(1 + (n - i) * |incy|)$.
On exit, **incy** is unchanged.

Description

SVCAL and DVCAL compute the product of a real scalar and a real vector, in single or double precision. CVCAL and ZVCAL compute the product of a complex scalar and a complex vector, in single or double precision. CSVCAL and ZDVCAL compute the product of a real scalar and a complex vector in single or double precision.

These subprograms multiply each element of a vector by a scalar value, returning the result in vector y:

$$y \leftarrow \alpha x$$

If $incy = 0$, the result is unpredictable. If $incx = 0$, each element in y is equal to $\alpha X(1)$.

If α = 0, the computation is a time-consuming way of setting all elements of the vector y equal to zero. Use the _SET routines to perform that operation.

EXAMPLES

1.
```
INTEGER*4 N, INCX, INCY
REAL*4 X(20), Y(40), ALPHA
INCX = 1
INCY = 2
ALPHA = 2.0
N = 20
CALL SVCAL(N,ALPHA,X,INCX,Y,INCY)
```

This Fortran example shows how to scale a vector x by 2.0. Vector y is set equal to the result.

2.
```
INTEGER*4 N, INCX, INCY
COMPLEX*8 X(20), Y(40), ALPHA
INCX = 1
INCY = 2
ALPHA = (5.0, 1.0)
N = 20
CALL CVCAL(N,ALPHA,X,INCX,Y,INCY)
```

This Fortran example shows how to scale a vector x by the complex number (5.0,1.0). Vector y is set equal to the result.

SZAXPY DZAXPY CZAXPY ZZAXPY
Vector Plus the Product of a Scalar and a Vector

Format

{S,D,C,Z}ZAXPY (n, alpha, x, incx, y, incy, z, incz)

Arguments

n
integer*4
On entry, the number of elements of the vectors x and y.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar value to be multiplied with the elements of vector x.
On exit, **alpha** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the n elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
If **incx** > 0, vector x is stored forward in the array, so that x_i is stored in location $X(1 + (i - 1) * incx)$.
If **incx** < 0, vector x is stored backward in the array, so that x_i is stored in location $X(1 + (n - i) * |incx|)$.
If **incx** = 0, only the first element is accessed.
On exit, **incx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$, containing the n elements of the vector y.
On exit, **y** is unchanged.

incy
integer*4
On entry, the increment for the array Y.
If **incy** ≥ 0, vector y is stored forward in the array, so that y_i is stored in location $Y(1 + (i - 1) * incy)$.
If **incy** < 0, vector y is stored backward in the array, so that y_i is stored in location $Y(1 + (n - i) * |incy|)$.
On exit, **incy** is unchanged.

z
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array Z of length at least $(1 + (n - 1) * |incz|)$.
On exit, if $n \leq 0$, then **z** is unchanged. If $n > 0$, **z** is overwritten with the products; each z_i is replaced by $y_i + \alpha x_i$.

incz
integer*4
On entry, the increment for the array Z.
If **incz** ≥ 0, vector z is stored forward in the array, so that z_i is stored in location $Z(1+(i-1)*incz)$.
If **incz** < 0, vector z is stored backward in the array, so that z_i is stored in location $Z(1+(n-i)*|incz|)$.
On exit, **incz** is unchanged.

Description

The _ZAXPY subprograms compute the product of a scalar and a vector, add the result to the elements of another vector, and then store the result in vector z:

$$z \leftarrow \alpha x + y$$

where α is a scalar, and x, y, and z are vectors with n elements.

The scalar α must not share a memory location with any element of the vector z. If $incz = 0$ or if any element of z shares a memory location with an element of x or y, the results are unpredictable.

If $incx = 0$, the computation is a time-consuming way of adding the constant αx_1 to all the elements of y. The following chart shows the resulting operation from the interaction of the **incx** and **incy** arguments:

	incx = 0	**incx** $\neq$ **0**
incy = 0	$z_i = y_1 + \alpha x_1$	$z_i = y_1 + \alpha x_i$
incy $\neq$ **0**	$z_i = y_i + \alpha x_1$	$z_i = y_i + \alpha x_i$

Example

```
INTEGER*4 N, INCX, INCY, INCZ
REAL*4 X(20), Y(20), Z(40), ALPHA
INCX = 1
INCY = 1
INCZ = 2
ALPHA = 2.0
N = 20
CALL SZAXPY(N,ALPHA,X,INCX,Y,INCY,Z,INCZ)
```

This Fortran example shows how all elements of the vector x are multiplied by 2.0 and added to the elements of vector y. Vector z contains the result.

7

Using the Sparse Level 1 BLAS Subprograms

The Sparse Level 1 BLAS subprograms perform vector-vector operations commonly occuring in many computational problems in sparse linear algebra. In contrast to the subprograms in Level 1 BLAS Subprograms and Level I BLAS Extensions, these subprograms operate on sparse vectors. This chapter provides information on the following topics:

- Operations performed by the Sparse Level 1 BLAS subprograms (Section 7.1.1)
- Accuracy (Section 7.1.2)
- Sparse Level 1 vector storage (Section 7.2)
- Naming conventions (Section 7.3)
- Subprogram summary (Section 7.4)
- Calling Sparse Level 1 BLAS subprograms (Section 7.5)
- Argument conventions (Section 7.6)
- Error handling (Section 7.7)
- A look at a Sparse Level 1 BLAS subprogram (Section 7.8)

A description of each Sparse Level 1 BLAS subprogram follows this chapter.

7.1 Sparse Level 1 BLAS Operations

The Sparse Level 1 BLAS subprograms are sparse extensions of the Level 1 BLAS subprograms. While similar in functionality to the Level 1 BLAS subprograms, the sparse subprograms operate on sparse vectors stored in a compressed form.

DXML enhances the functionality of the Sparse Level 1 BLAS outlined in [Dodson, Grimes, and Lewis 1991], by the addition of three subprograms that also operate on sparse vectors.

7.1.1 Types of Operations

The sparse extensions of Level 1 BLAS subprograms that are of interest involve two vectors. The standard approach in sparse vector computation is to expand one vector into its full form and perform the numerical operations between that uncompressed vector and the remaining compressed vector. The Sparse Level 1 BLAS subprograms can be classified into two types:

- A vector in uncompressed form is returned as output.
  In order for these operations to give correct and consistent results on a vector or parallel machine, the values in the index vector, associated with the vector stored in compressed form, must be distinct.
- A scalar or a vector in compressed form is returned as output.

7.1.2 Accuracy

Because of the efficient coding of the subprograms, in some cases, the results obtained might not match the results obtained using the conventional order of evaluation. Whenever this could happen, it is stated in the reference description for that subprogram.

7.2 Sparse Vector Storage

For the Sparse Level 1 BLAS subprograms, a vector is stored in one of two ways:

- In a one-dimensional array in full form
- In two one-dimensional arrays in compressed form

7.2.1 Sparse Vectors

A sparse vector is a vector that has a relatively large number of zeros. In such cases, substantial savings in computation and memory requirements can be achieved by storing and operating on only the nonzero elements. For example, consider the vector x, of length 9, as shown in the following example:

$$\mathrm{X} = \begin{bmatrix} 2.0 \\ 0.0 \\ 0.0 \\ 3.5 \\ 0.0 \\ 9.8 \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix}$$

Only three of the nine elements in this vector are nonzero and by storing just these elements, the memory requirements for storing x can be reduced by a factor of three. This transformation converts the original vector from its full form to a vector in compressed form. Additional storage is, however, required for storing information that enables the original vector to be reconstructed from the vector stored in the compressed form. This implies that in addition to the vector of nonzero elements and the number of nonzero elements, there should be a companion array of indices that map the stored elements into their proper places in the original vector. Thus, the vector x is stored in compressed form as two separate arrays, XC and INDXC:

$$\mathrm{XC} = \begin{bmatrix} 2.0 \\ 3.5 \\ 9.8 \end{bmatrix}$$

$$\mathrm{INDXC} = \begin{bmatrix} 1 \\ 4 \\ 6 \end{bmatrix}$$

where XC is the array of nonzero elements and INDXC is an array of indices that is used to reconstruct the original vector. For example, since the second element in INDXC is 4, it implies that the second element in XC, 3.5, is the fourth element in the original array X. Both XC and INDXC are of length nz = 3, where nz is the number of nonzero elements in X.

Operations on sparse vectors are performed only on the nonzero elements of the vector. As a result, it is the number of nonzero elements that is important, not the length of the original vector x. Moreover, as the elements of the vector stored in compressed form are contiguous, there is no need for an increment parameter; it is always 1.

7.2.2 Storing a Sparse Vector

Suppose X is a sparse one-dimensional array of length n, with nz nonzero elements. As x is a sparse vector, nz is much smaller than n, that is, a large number of elements of the vector x are zero.

Let XC be the vector x stored in compressed form, that is, XC contains only the nonzero elements of x. Let INDXC be the array of indices that map each element of XC into its proper position in the array X. Then X(INDXC(i)) = XC(i). It follows that:

$$\max(\mathrm{INDXC}(i),\ i = 1, nz) \leq n$$

That is, if the original vector has length n, then the values of the elements in array INDXC can be at most n.

A sparse vector, stored in a compressed form, is thus defined by three quantities:

- Number of nonzero elements: nz
- Array of length at least nz, containing the nonzero elements of array X: XC
- Array of length at least nz containing the indices of the nonzero elements in the original uncompressed form: INDXC

7.3 Naming Conventions

Table 7–1 shows the characters used in the names of the Sparse Level 1 BLAS, and their meaning.

Table 7–1 Naming Conventions: Sparse Level 1 BLAS Subprogram

Character Group	Mnemonic	Meaning
First group	S	Single-precision real
	D	Double-precision real
	C	Single-precision complex
	Z	Double-precision complex
Second group	A combination of letters such as DOT or SCTR	Type of computation such as dot product or a vector scatter
Third group	I	Refers to indexed computation used in sparse vectors
	S or Z	Scale or zero
	No mnemonic	-

For example, the name SSCTRS refers to the single precision real subprogram for scaling and then scattering the elements of a sparse vector stored in compressed form.

7.4 Summary of Sparse Level 1 BLAS Subprograms

Table 7–2 summarizes the Sparse Level 1 BLAS subprograms provided by DXML.

Table 7–2 Summary of Sparse Level 1 BLAS Subprograms

Subprogram Name	Operation
SAXPYI	Calculates, in single-precision arithmetic, the product of a real scalar and a real sparse vector in compressed form and adds the result to a real vector in full form.
DAXPYI	Calculates, in double-precision arithmetic, the product of a real scalar and a real sparse vector in compressed form and adds the result to a real vector in full form.
CAXPYI	Calculates, in single-precision arithmetic, the product of a complex scalar and a complex sparse vector in compressed form and adds the result to a complex vector in full form.
ZAXPYI	Calculates, in double-precision arithmetic, the product of a complex scalar and a complex sparse vector in compressed form and adds the result to a complex vector in full form.
SSUMI	Calculates, in single-precision arithmetic, the sum of a real sparse vector stored in compressed form and a real vector stored in full form.
DSUMI	Calculates, in double-precision arithmetic, the sum of a real sparse vector stored in compressed form and a real vector stored in full form.
CSUMI	Calculates, in single-precision arithmetic, the sum of a complex sparse vector stored in compressed form and a complex vector stored in full form.
ZSUMI	Calculates, in double-precision arithmetic, the sum of a complex sparse vector stored in compressed form and a complex vector stored in full form.
SDOTI	Calculates, in single-precision arithmetic, the product of a real vector and a real sparse vector stored in compressed form.
DDOTI	Calculates, in double-precision arithmetic, the product of a real vector and a real sparse vector stored in compressed form.
CDOTUI	Calculates, in single-precision arithmetic, the product of a complex vector and an unconjugated complex sparse vector stored in compressed form.
ZDOTUI	Calculates, in double-precision arithmetic, the product of a complex vector and an unconjugated complex sparse vector stored in compressed form.
CDOTCI	Calculates, in single-precision arithmetic, the product of a complex vector and a conjugated complex sparse vector stored in compressed form.
ZDOTCI	Calculates, in double-precision arithmetic, the product of a complex vector and a conjugated complex sparse vector stored in compressed form.

(continued on next page)

Table 7–2 (Cont.) Summary of Sparse Level 1 BLAS Subprograms

Subprogram Name	Operation
SGTHR	Constructs, in single-precision arithmetic, a real sparse vector in compressed form from the specified elements of a real vector in full form.
DGTHR	Constructs, in double-precision arithmetic, a real sparse vector in compressed form from the specified elements of a real vector in full form.
CGTHR	Constructs, in single-precision arithmetic, a complex sparse vector in compressed form from the specified elements of a complex vector in full form.
ZGTHR	Constructs, in double-precision arithmetic, a complex sparse vector in compressed form from the specified elements of a complex vector in full form.
SGTHRS	Constructs, in single-precision arithmetic, a real sparse vector in compressed form from the specified scaled elements of a real vector in full form.
DGTHRS	Constructs, in double-precision arithmetic, a real sparse vector in compressed form from the specified scaled elements of a real vector in full form.
CGTHRS	Constructs, in single-precision arithmetic, a complex sparse vector in compressed form from the specified scaled elements of a complex vector in full form.
ZGTHRS	Constructs, in double-precision arithmetic, a complex sparse vector in compressed form from the specified scaled elements of a complex vector in full form.
SGTHRZ	Constructs, in single-precision arithmetic, a real sparse vector in compressed form from the specified elements of a real vector in full form and sets the elements to zero.
DGTHRZ	Constructs, in double-precision arithmetic, a real sparse vector in compressed form from the specified elements of a real vector in full form and sets the elements to zero.
CGTHRZ	Constructs, in single-precision arithmetic, a complex sparse vector in compressed form from the specified elements of a complex vector in full form and sets the elements to zero.
ZGTHRZ	Constructs, in double-precision arithmetic, a complex sparse vector in compressed form from the specified elements of a complex vector in full form and sets the elements to zero.
SROTI	Applies, in single-precision arithmetic, a Givens rotation for a real sparse vector stored in compressed form and another vector stored in full form.
DROTI	Applies, in double-precision arithmetic, a Givens rotation for a real sparse vector stored in compressed form and another vector stored in full form.

(continued on next page)

Table 7–2 (Cont.) Summary of Sparse Level 1 BLAS Subprograms

Subprogram Name	Operation
SSCTR	Scatters, in single-precision arithmetic, the components of a sparse real vector in compressed form into the specified components of a real vector in full form.
DSCTR	Scatters, in double-precision arithmetic, the components of a sparse real vector in compressed form into the specified elements a real vector in full form.
CSCTR	Scatters, in single-precision arithmetic, the components of a sparse complex vector in compressed form into the specified elements of a complex vector in full form.
ZSCTR	Scatters, in double-precision arithmetic, the components of a sparse complex vector in compressed form into the specified elements of a complex vector in full form.
SSCTRS	Scales and then scatters, in single-precision arithmetic, the components of a sparse real vector in compressed form into the specified components of a real vector in full form.
DSCTRS	Scales and then scatters, in double-precision arithmetic, the components of a sparse real vector in compressed form into the specified elements a real vector in full form.
CSCTRS	Scales and then scatters, in single-precision arithmetic, the components of a sparse complex vector in compressed form into the specified elements of a complex vector in full form.
ZSCTRS	Scales and then scatters, in double-precision arithmetic, the components of a sparse complex vector in compressed form into the specified elements of a complex vector in full form.

7.5 Calling Subprograms

Some of the subprograms return a scalar. These subprograms are functions and are called as functions by coding a function reference.

In the reference section at the end of this chapter, a reference description for a function includes a **Function Value** section. For all the subprograms that are functions, all of the arguments are input arguments, which are unchanged on exit. The example at the end of each function reference description shows the function call.

Some of the subprograms return a vector. These subprograms are subroutines and are called as subroutines with a CALL statement.

A reference description for a subroutine does not have a **Function Value** section. Each subroutine has an output argument that is overwritten on exit and contains the output vector information. The example at the end of each subroutine reference description shows the subroutine call.

7.6 Argument Conventions

Each Sparse Level 1 BLAS subprogram has arguments that specify the nature and requirements of the subprogram. There are no optional arguments.

The arguments are ordered by category, but not every argument category is needed in each of the subprograms:

- Argument defining the number of nonzero elements
- Argument defining the input scalar
- Arguments describing the input and output vectors

7.6.1 Defining the Number of Nonzero Elements

The Sparse Level 1 BLAS subprograms operate only on the nonzero elements of the sparse vector. Thus in contrast to the Level 1 BLAS subprograms, it is the number of nonzero elements that is input to the subprogram, not the length of the vector. The number of nonzero elements is defined by the argument **nz**.

The values **nz**< 0, **nz** $= 0$ and **nz** > 0 are all allowed. For **nz**≤ 0, the routines return zero function values (if applicable) and make no references to their vector arguments.

7.6.2 Defining the Input Scalar

The input scalar α is always defined by the argument **alpha**.

7.6.3 Describing the Input/Output Vectors

Sparse Level 1 BLAS subprograms operate on two types of vectors: compressed and uncompressed. The elements of the full uncompressed vector y, specified by the argument **y** are stored contiguously, that is, with increment equal to 1. As a result, there is no input parameter for the increment of the vector y as it is always assumed to be 1.

The sparse vector x is stored in compressed form as array X, containing nz elements. The companion array of indices, array INDX, also of length nz, replaces the increment argument of the Level 1 BLAS subprograms.

7.7 Error Handling

The Sparse Level 1 BLAS subprograms assume that input parameters are correct and provide no feedback when problems occur. You must ensure that all input data for these subprograms is correct.

7.8 A Look at a Sparse Level 1 BLAS Subprogram

To understand the differences between a Level 1 BLAS subprogram and its sparse counterpart, consider the routines SAXPY and SAXPYI. They perform essentially the same operation, but SAXPY operates on full vectors and SAXPYI operates on sparse vectors.

Consider two arrays: array Y of length n = 9, stored in full uncompressed form and the sparse array X, also of length n = 9 and also stored in full form.

$$Y = \begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \\ 5.0 \\ 6.0 \\ 7.0 \\ 8.0 \\ 9.0 \end{bmatrix} \qquad X = \begin{bmatrix} 2.0 \\ 0.0 \\ 1.0 \\ 0.0 \\ 0.0 \\ 3.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix}$$

As the array X is sparse, it can be stored in the compressed form as an array XC of length 3 and a companion integer array INDXC, also of length 3.

$$XC = \begin{bmatrix} 2.0 \\ 1.0 \\ 3.0 \end{bmatrix} \qquad INDXC = \begin{bmatrix} 1 \\ 3 \\ 6 \end{bmatrix}$$

The routine SAXPY, from Level 1 BLAS, operates on the arrays X and Y as shown in the following code:

```
REAL*4   X(9), Y(9), ALPHA
INTEGER INCX, INCY, N
ALPHA = 2.0
INCX = 1
INCY = 1
N = 9
CALL SAXPY(N,ALPHA,X,INCX,Y,INCY)
```

The routine SAXPYI, from Sparse Level 1 BLAS, operates on the arrays XC and Y as shown in the following code:

```
REAL*4   XC(3), Y(9), ALPHA
INTEGER INDXC(3), NZ
ALPHA = 2.0
NZ = 3
CALL SAXPYI(NZ,ALPHA,XC,INDXC,Y)
```

With α = 2.0, both calls result in the updated vector y:

$$Y = \begin{bmatrix} 5.0 \\ 2.0 \\ 5.0 \\ 4.0 \\ 5.0 \\ 12.0 \\ 7.0 \\ 8.0 \\ 9.0 \end{bmatrix}$$

In SAXPY, all the elements of the array X and Y are operated on, resulting in 18 arithmetic operations (additions and multiplications). In contrast, SAXPYI operates only on the nonzero elements, resulting in 6 arithmetic operations. Storing both the vectors in uncompressed form requires 18 memory locations for real operands. Storing array X in a compressed form and array Y in an uncompressed form requires 12 memory locations for real operands and 3 for integer operands.

The savings in compute time and memory requirements can be substantial, when the array X is very sparse.

Sparse Level 1 BLAS Subprograms

This section provides descriptions of the Sparse Level 1 BLAS subprograms.

SAXPYI DAXPYI CAXPYI ZAXPYI
Vector Plus the Product of a Scalar and a Sparse Vector

Format

{S,D,C,Z}AXPYI (nz, alpha, x, indx, y)

Arguments

nz
integer*4
On entry, the number of elements in the vector in the compressed form.
On exit, **nz** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar multiplier for the elements of vector x.
On exit, **alpha** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array of the elements of vector x in compressed form.
On exit, **x** is unchanged.

indx
integer*4
On entry, an array containing the indices of the compressed form. The values in the INDX array must be distinct for consistent vector or parallel execution.
On exit, **indx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array of the elements of vector y stored in full form.
On exit, if $nz \leq 0$ or if $\alpha = 0$, **y** is unchanged. If $nz > 0$, the elements in the vector y corresponding to the indices in the INDX array are overwritten.

Description

The _AXPYI subprograms compute the product of a scalar α and a sparse vector x stored in compressed form. The product is then added to a vector y and the result is stored as an updated vector y in full form. Only the elements of vector y whose indices are listed in INDX are updated. For $i = 1, ..., nz$:

$$y(indx(i)) \longleftarrow y(indx(i)) + \alpha * x(i)$$

If $nz \leq 0$ or $\alpha = 0.0$, y is unchanged. SAXPYI and DAXPYI compute the product of a real scalar and a real sparse vector stored in compressed form, and add the product to a real vector in full form. CAXPYI and ZAXPYI compute the product of a complex scalar and a complex sparse vector stored in compressed form, and add the product to a complex vector stored in full form.

Example

```
INTEGER NZ, INDX(10)
REAL*4 Y(40), X(10), ALPHA
NZ = 10
ALPHA = 2.0
CALL SAXPYI(NZ, ALPHA, X, INDX, Y)
```

This Fortran code shows how the nz elements in y, corresponding to the indices in the INDX array, are updated by the addition of a scalar multiple of the corresponding element of the compressed vector, x.

SDOTI DDOTI CDOTUI ZDOTUI CDOTCI ZDOTCI
Inner Product of a Vector and a Sparse Vector

Format

{S,D}DOTI (nz, x, indx, y)

{C,Z}DOT{U,C}I (nz, x, indx, y)

Function Value

dotpr
real*4 | real*8 | complex*8 | complex*16
The inner product of the sparse vector x and the full vector y.

Arguments

nz
integer*4
On entry, the number of elements in the vector in the compressed form.
On exit, **nz** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array of the elements of vector x in compressed form.
On exit, **x** is unchanged.

indx
integer*4
On entry, an array containing the indices of the compressed form.
On exit, **indx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array of the elements of vector y stored in full form.
On exit, **y** is unchanged. Only the elements in the vector y corresponding to the indices in the INDX array are accessed.

Description

These routines compute the vector inner product of a sparse vector x stored in compressed form with a vector y stored in full form. If $nz \leq 0$, **dotpr** is set equal to zero.

SDOTI and DDOTI multiply a real vector by a sparse vector of real values stored in compressed form. CDOTUI and ZDOTUI multiply a complex vector by an unconjugated sparse vector of complex values stored in compressed form. CDOTCI and ZDOTCI multiply a complex vector by a conjugated sparse vector of complex values stored in compressed form.

As shown in (7–1) and (7–2), CDOTUI and ZDOTUI operate on the vector x in the unconjugated form; CDOTCI and ZDOTCI operate on the vector x in conjugated form.

$$\text{Unconjugated form}: \quad dotpr = \sum_{i=1}^{nz} x(i) * y(indx(i)) \tag{7–1}$$

$$\text{Conjugated form :} \quad dotpr = \sum_{i=1}^{nz} \overline{x}(i) * y(indx(i)) \tag{7–2}$$

The order of operations for the evaluation of $dotpr$ may be different fom the sequential order of operations. The results obtained from these two evaluations may not be identical.

Example

```
INTEGER NZ, INDX(15)
COMPLEX*8 Y(50), X(15)
NZ = 10
CINNER = CDOTUI(NZ, X, INDX, Y)
```

This Fortran code produces the inner product of two vectors, x and y. Vector y is stored in full form and vector x is stored in compressed form. The elements of vector x are used in unconjugated form.

SGTHR DGTHR CGTHR ZGTHR
Gathers the Specified Elements of a Vector

Format

{S,D,C,Z}GTHR (nz, y, x, indx)

Arguments

nz
integer*4
On entry, the number of elements to be gathered into the compressed form.
On exit, **nz** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array of the elements of vector y stored in full form.
On exit, **y** is unchanged. Only the elements in the vector y corresponding to the indices in the INDX array are accessed.

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array that receives the specified elements of the vector y.
On exit, if $nz \leq 0$, **x** is unchanged. If $nz > 0$, array X contains the values gathered into compressed form.

indx
integer*4
On entry, an array containing the indices of the values to be gathered into compressed form.
On exit, **indx** is unchanged.

Description

The _GTHR subprograms gather specified elements from a vector in full form, y, and store them as a vector x in compressed form. For $i = 1, ..., nz$:

$$x(i) \longleftarrow y(indx(i))$$

If $nz \leq 0$, x is unchanged.

SGTHR and DGTHR gather the specified elements from a real vector in full form and store them as a real sparse vector in compressed form. CGTHR and ZGTHR gather the specified elements from a complex vector in full form and store them as a complex sparse vector in compressed form.

Example

```
INTEGER NZ, INDX(10)
REAL*4 Y(40), X(10)
NZ = 10
CALL SGTHR(NZ, Y, X, INDX)
```

This Fortran code shows how the nz elements of the vector y, corresponding to the indices in the INDX array, are gathered in a compressed form into the vector x.

SGTHRS DGTHRS CGTHRS ZGTHRS
Gathers and Scales the Specified Elements of a Vector

Format

{S,D,C,Z}GTHRS (nz, alpha, y, x, indx)

Arguments

nz
integer*4
On entry, the number of elements to be gathered into the compressed form.
On exit, **nz** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar multiplier for the elements of vector y.
On exit, **alpha** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array of the elements of vector y stored in full form.
On exit, **y** is unchanged. Only the elements in the vector y corresponding to the indices in the INDX array are accessed.

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array that receives the specified elements of vector y after scaling.
On exit, if $nz \leq 0$, **x** is unchanged. If $nz > 0$, the array X contains the specified elements of vector y after scaling by the scalar, α.

indx
integer*4
On entry, an array containing the indices of the values to be gathered into compressed form.
On exit, **indx** is unchanged.

Description

The _GTHRS subprograms gather specified elements of vector y in full form, multiply the elements by α, and store the result as elements of a sparse vector x in compressed form. For $i = 1, ..., nz$:

$$x(i) \longleftarrow \alpha * y(indx(i))$$

If $nz \leq 0$, x is unchanged.

SGTHRS and DGTHRS gather the elements from a real vector in full storage and scale them into a real vector in compressed storage. CGTHRS and ZGTHRS gather the specified elements from a complex vector in full storage and scale them into a complex vector in compressed storage.

The _GTHRS subprograms are not part of the original set of Sparse BLAS Level 1 subprograms.

Example

```
INTEGER NZ, INDX(10)
REAL*8 Y(40), X(10), ALPHA
NZ = 10
ALPHA = 1.5D0
CALL DGTHRS(NZ, ALPHA, Y, X, INDX)
```

This Fortran code shows how the nz elements of the vector y, corresponding to the indices in the INDX array, are scaled by the scalar **alpha** and gathered in a compressed form into the vector x.

SGTHRZ DGTHRZ CGTHRZ ZGTHRZ
Gathers and Zeros Specified Elements of a Vector

Format

{S,D,C,Z}GTHRZ (nz, y, x, indx)

Arguments

nz
integer*4
On entry, the number of elements to be gathered into the compressed form.
On exit, **nz** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array of the elements of vector y stored in full form.
On exit, if $nz \leq 0$, **y** is unchanged. If $nz > 0$, the gathered elements in Y are set to zero. Only the elements in the vector y corresponding to the indices in the INDX array are overwritten.

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array that receives the specified elements of vector y.
On exit, if $nz \leq 0$ **x** is unchanged. If $nz > 0$, the array X contains the values gathered into compressed form.

indx
integer*4
On entry, an array containing the indices of the values to be gathered into compressed form. The values in INDX must be distinct for consistent vector or parallel execution.
On exit, **indx** is unchanged.

Description

The _GTHRZ subprograms gather the specified elements from a vector y stored in full form into a sparse vector x in compressed form. Those elements in y are then set to zero. For $i = 1, ..., nz$:

$$x(i) \longleftarrow y(indx(i))$$

$$y(indx(i)) \longleftarrow 0.0$$

If $nz \leq 0$, both x and y are unchanged. SGTHRZ and DGTHRZ gather the specified elements of a real vector in full form into a real sparse vector in compressed form. CGTHRZ and ZGTHRZ gather the specified elements of a complex vector in full form into a complex sparse vector in compressed form. In each case, the specified elements of the full vector are set equal to zero.

Example

```
INTEGER NZ, INDX(10)
REAL*4 Y(40), X(10)
NZ = 10
CALL SGTHRZ(NZ, Y, X, INDX)
```

This Fortran code shows how the nz elements of the vector y, corresponding to the indices in the INDX array, are gathered in a compressed form into the vector x. The gathered elements in y are set equal to zero.

SROTI DROTI
Real Givens Plane Rotation Applied to Sparse Vector

Format

{S,D}ROTI (nz, x, indx, y, c, s)

Arguments

nz
integer*4
On entry, the number of elements in the vector in the compressed form.
On exit, **nz** is unchanged.

x
real*4 | real*8
On entry, an array of the elements of vector x in compressed form.
On exit, if $nz \leq 0$, **x** is unchanged. If $nz > 0$, the array X is updated.

indx
integer*4
On entry, an array containing the indices of the compressed form. The values in INDX must be distinct for consistent vector or parallel execution.
On exit, **indx** is unchanged.

y
real*4 | real*8
On entry, an array of the elements of vector y stored in full form.
On exit, if $nz \leq 0$, **y** is unchanged. If $nz > 0$, the elements in the vector y corresponding to the indices in the INDX array are overwritten.

c
real*4 | real*8
On entry, **c** is the first rotation element, which can be interpreted as the cosine of the angle of rotation.
On exit, **c** is unchanged.

s
real*4 | real*8
On entry, **s** is the second rotation element, which can be interpreted as the sine of the angle of rotation.
On exit, **s** is unchanged.

Description

The _ROTI routines apply a real Givens rotation to a sparse vector x stored in compressed form and another vector y stored in full form. For $i = 1, ..., nz$:

$$temp \longleftarrow -s * x(i) + c * y(indx(i))$$

$$x(i) \longleftarrow c * x(i) + s * y(indx(i))$$

$$y(indx(i)) \longleftarrow temp$$

If $nz \leq 0$, x and y are unchanged. Only the elements of y whose indices are listed in INDX are referenced or modified.

The output vectors x and y have nonzero elements in the locations where either input vector x or y had nonzero elements. Because the _ROTI subprograms do not handle this fill-in, the arrays X and INDX must take this into account on input. This means that all nonzero elements of y must be listed in the array INDX, resulting in an INDX array containing the indices of all nonzero elements of both vectors x and y.

Example

```
INTEGER NZ, INDX(10)
REAL*8 Y(40), X(10), C, S
NZ = 10
CALL DROTI(NZ, X, INDX, Y, C, S)
```

This Fortran code shows how to apply a Givens rotation to a sparse vector x, stored in compressed form, and another vector y, stored in full form.

SSCTR DSCTR CSCTR ZSCTR
Scatters the Elements of a Sparse Vector

Format

{S,D,C,Z}SCTR (nz, x, indx, y)

Arguments

nz
integer*4
On entry, the number of elements to be scattered from the compressed form.
On exit, **nz** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array of the elements of vector x to be scattered from compressed form into full form.
On exit, **x** is unchanged.

indx
integer*4
On entry, an array containing the indices of the values to be scattered from the compressed form. The values in the INDX array must be distinct for consistent vector or parallel execution.
On exit, **indx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array that receives the elements of vector x.
On exit, if $nz \leq 0$, **y** is unchanged. If $nz > 0$, the elements in the vector y corresponding to the indices in the INDX array are set to the corresponding elements in vector x.

Description

The _SCTR routines scatter the elements stored in the sparse vector x in compressed form into the specified elements of the vector y in full form. For $i = 1, ..., nz$:

$$y(indx(i)) \longleftarrow x(i)$$

If $nz \leq 0$, y is unchanged. SSCTR and DSCTR scatter the elements of a real sparse vector stored in compressed form into the specified elements of a real vector in full form. CSCTR and ZSCTR scatter the elements of a complex sparse vector stored in compressed form into the specified elements of a complex vector in full form.

Example

```
INTEGER NZ, INDX(10)
REAL*4 Y(40), X(10)
NZ = 10
CALL SSCTR(NZ, X, INDX, Y)
```

This Fortran code scatters the elements of a sparse vector x, stored in compressed form, into the specified elements of the vector y, stored in full form.

SSCTRS DSCTRS CSCTRS ZSCTRS
Scales and Scatters the Elements of a Sparse Vector

Format

{S,D,C,Z}SCTRS (nz, alpha, x, indx, y)

Arguments

nz
integer*4
On entry, the number of elements to be scattered from the compressed form.
On exit, **nz** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar multiplier for the elements of vector x.
On exit, **alpha** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array of the elements of vector x to be scattered from compressed form into full form.
On exit, **x** is unchanged.

indx
integer*4
On entry, an array containing the indices of the values to be scattered from the compressed form. The values in INDX must be distinct for consistent vector or parallel execution.
On exit, **indx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array that receives the scaled elements of the vector x.
On exit, if $nz \leq 0$, **y** is unchanged. If $nz > 0$, the elements in the vector y corresponding to the indices in the INDX array are set to the corresponding scaled entries of x.

Description

The _SCTRS subprograms multiply the elements of a sparse vector x stored in compressed form by a scalar α and then scatter them into the specified elements of the vector y stored in full form. For $i = 1, ..., nz$:

$$y(indx(i)) \longleftarrow \alpha * x(i)$$

If $nz \leq 0$, y is unchanged.

SSCTRS and DSCTRS scatter the elements of a real sparse vector stored in compressed form, after scaling, into the specified elements of a real vector stored in full form. CSCTRS and ZSCTRS scatter the elements of a complex sparse vector stored in compressed form, after scaling, into the specified elements of a complex vector stored in full form.

The _SCTRS subprograms are not part of the original set of Sparse BLAS Level 1 subprograms.

Example

```
INTEGER NZ, INDX(10)
REAL*8 Y(40), X(10), ALPHA
NZ = 10
ALPHA = 2.5D0
CALL DSCTRS(NZ, ALPHA, X, INDX, Y)
```

This Fortran code scales the elements of a sparse vector x, stored in compressed form, by the scalar **alpha**, and then scatters them into the specified elements of the vector y, stored in full form.

SSUMI DSUMI CSUMI ZSUMI
Sum of a Vector and a Sparse Vector

Format

{S,D,C,Z}SUMI (nz, x, indx, y)

Arguments

nz
integer*4
On entry, the number of elements in the vector in the compressed form.
On exit, **nz** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array of the elements of vector x in compressed form.
On exit, **x** is unchanged.

indx
integer*4
On entry, an array containing the indices of the compressed form. The values in the INDX array must be distinct for consistent vector or parallel execution.
On exit, **indx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array of the elements of vector y stored in full form.
On exit, if $nz \leq 0$, **y** is unchanged. If $nz > 0$, the elements in the vector y corresponding to the indices in the INDX array are overwritten.

Description

SSUMI and DSUMI add a sparse vector of real values stored in compressed form to a real vector stored in full form. CSUMI and ZSUMI add a sparse vector of complex values stored in compressed form to a complex vector stored in full form.

Only the elements of vector y whose indices are listed in array INDX are overwritten. For $i = 1, ..., nz$:

$$y(indx(i)) \longleftarrow y(indx(i)) + x(i)$$

If $nz \leq 0$, y is unchanged. The _SUMI subprograms are an efficient implementation of the _AXPYI subprograms when α = 1.0.

The _SUMI subprograms are not part of the original set of Sparse BLAS Level 1 subprograms.

Example

```
INTEGER NZ, INDX(20)
REAL*8 Y(100), X(20)
NZ = 20
CALL DSUMI(NZ, X, INDX, Y)
```

This Fortran code shows how the nz elements in y, corresponding to the indices in the array INDX, are updated by the addition of the corresponding element of the compressed vector, x.

8

Using the Level 2 BLAS Subprograms

The Level 2 BLAS subprograms perform matrix-vector operations commonly occurring in many computational problems in linear algebra. This chapter provides information about the following topics:

- Operations performed by the Level 2 BLAS subprograms (Section 8.1)
- Vector and matrix storage (Section 8.2)
- Subprogram naming conventions (Section 8.3)
- Subprogram summary (Section 8.4)
- Calling Level 2 BLAS subprograms (Section 8.5)
- Arguments used in the subprograms and invalid arguments (Sections 8.6 and 8.6.6)
- Performing rank-one and rank-two updates to band matrices (Section 8.7)
- Error handling (Section 8.8)
- A look at a Level 2 subprogram and its use (Section 8.9)

The reference descriptions of the Level 2 BLAS subprograms are at the end of this chapter.

A key Level 2 BLAS subprogram, {C,D,S,Z}GEMV, has been parallelized for improved peformance on multiprocessor systems. For information about using the parallel library, see Chapter 4.

8.1 Level 2 BLAS Operations

Level 2 BLAS subprograms perform operations that involve only one matrix.

8.1.1 Types of Operations

The subprograms perform three types of basic matrix-vector operations:

- Matrix-vector products

$$y \leftarrow \alpha A x + \beta y$$

$$y \leftarrow \alpha A^T x + \beta y$$

$$y \leftarrow \alpha A^H x + \beta y$$

$$x \leftarrow T x$$

$$x \leftarrow T^T x$$

- Rank-one and rank-two updates

$$A \leftarrow A + \alpha x y^T$$

$$A \leftarrow A + \alpha x y^H$$

$$A \leftarrow A + \alpha x y^T + \alpha y x^T$$

$$A \leftarrow A + \alpha x y^H + \overline{\alpha} y x^H$$

- Triangular system solvers

$$Tx = b$$

$$T^T x = b$$

$$T^H x = b$$

α and β are scalars, x, y, and b are vectors, A is a matrix, and T is an upper- or lower-triangular matrix. For the triangular system solvers, T must also be non-singular; that is, $\det(T)$ is not equal to zero. Where appropriate, these operations are applied to different types of matrices:

- General matrix
- General band matrix
- Symmetric matrix
- Symmetric band matrix
- Hermitian matrix
- Hermitian band matrix
- Triangular matrix
- Triangular band matrix

Sixteen real subprograms perform 67 different operations, and 17 complex subprograms perform 68 different operations in both single-precision and double-precision arithmetic.

8.2 Vector and Matrix Storage

Level 2 BLAS subprograms store a vector in a one-dimensional array. (See Section 6.2.)

The matrix A is stored in one of two ways:

- A is stored in a two-dimensional array.
- A is stored in packed form in a one-dimensional array.

8.2.1 Defining a Matrix in an Array

A matrix is usually stored in a two-dimensional array. When every element of a matrix is stored, the storage scheme is called full matrix storage. If the matrix itself is a special kind of matrix such as a triangular matrix or a band matrix, a large number of storage locations are wasted using full matrix storage, and other storage methods can be used.

If a matrix is complex, each matrix element has the form $a + bi$. For each complex element, two storage locations in succession are needed to store a and b. Storing a complex matrix requires twice the number of storage locations as storing a real matrix of the same precision.

The columns of the matrix are stored one after the other in the array. The array can be much larger than the matrix that is stored in the array.

The storage of a matrix is defined using four arguments in a DXML subprogram argument list:

- Matrix location: Base address of the matrix in the array. It also tells where processing begins.
- The first, or leading, dimension of the array: Space, or increment, between consecutive elements of a row in an array.
- The number of rows m of the matrix.
- The number of columns n of the matrix.

These four quantities together specify which elements of an array are selected to become the matrix.

8.2.1.1 Matrix Location

The location given by the matrix argument in a DXML subroutine argument list is the starting point for selecting matrix elements. For example, consider the array A declared as A(5,7).

$$\mathrm{A} = \begin{bmatrix} 1.0 & 6.0 & 11.0 & 16.0 & 21.0 & 26.0 & 31.0 \\ 2.0 & 7.0 & 12.0 & 17.0 & 22.0 & 27.0 & 32.0 \\ 3.0 & 8.0 & 13.0 & 18.0 & 23.0 & 28.0 & 33.0 \\ 4.0 & 9.0 & 14.0 & 19.0 & 24.0 & 29.0 & 34.0 \\ 5.0 & 10.0 & 15.0 & 20.0 & 25.0 & 30.0 & 35.0 \end{bmatrix} \qquad (8–1)$$

If you specify A(2,3) as the starting point for the selection of matrix elements, then processing begins at the element in row 2 and column 3, which is the element 12.0.

8.2.1.2 First Dimension of the Array

The first (or leading) dimension of an array, which is specified by an argument such as **lda** in the DXML subprogram argument list, is the number of rows in the array from which the matrix elements are being selected. The first dimension of the array is used as an increment to select matrix elements from successive columns of the array.

The first dimension must be greater than or equal to m, the number of rows of the matrix. If the first dimension were less than m, then elements from one column of a matrix would be stored in more than one column of the array. This storage mechanism would lead to access of incorrect elements.

For the array A shown in (8–1), the first dimension is 5. More generally, for an array A declared as A(FL:FU,SL:SU), the first dimension of the array is as follows:

$$\mathrm{FU} - \mathrm{FL} + 1$$

8.2.1.3 Number of Rows and Columns of the Matrix

You specify the number of rows m of the matrix and the number of columns n of the matrix by specifying an integer value for the row and column arguments, such as **m** and **n**.

You can think about the matrix as the number of rows and columns of the array that you want to process. After processing the first element, the subprogram continues until m elements in n columns have been processed.

8.2.1.4 Selecting Matrix Elements from an Array

Again, consider the array A declared as A(5,7), with A(2,3) specified as the location of the matrix, which is also the starting point for the selection of matrix elements.

$$\mathrm{A} = \begin{bmatrix} 1.0 & 6.0 & 11.0 & 16.0 & 21.0 & 26.0 & 31.0 \\ 2.0 & 7.0 & 12.0 & 17.0 & 22.0 & 27.0 & 32.0 \\ 3.0 & 8.0 & 13.0 & 18.0 & 23.0 & 28.0 & 33.0 \\ 4.0 & 9.0 & 14.0 & 19.0 & 24.0 & 29.0 & 34.0 \\ 5.0 & 10.0 & 15.0 & 20.0 & 25.0 & 30.0 & 35.0 \end{bmatrix} \quad (8–2)$$

Processing begins at element 12.0. If the number of rows m to be processed is 3, and the number of columns n to be processed is 4, DXML adds the value of the first dimension of the array, which is 5, to find the starting point in the next column, which is element 17.0. DXML continues this until the number of columns processed is 4. The starting points of the columns are elements 12.0, 17.0, 22.0, and 27.0. Then, to find the matrix elements in each column of A, DXML repeatedly adds the value 1 to the starting point in a column until 3 elements in each column have been processed. The matrix elements selected in this example specify the matrix A shown in (8–3):

$$A = \begin{bmatrix} 12.0 & 17.0 & 22.0 & 27.0 \\ 13.0 & 18.0 & 23.0 & 28.0 \\ 14.0 & 19.0 & 24.0 & 29.0 \end{bmatrix} \quad (8–3)$$

The matrix does not have elements from all the rows and columns of the array. No elements are selected from rows 1 or 5 or from columns 1, 2, or 7. However, the matrix formed is a rectangular block in the array.

8.2.2 Symmetric and Hermitian Matrices

A matrix is symmetric if it is equal to its transpose:

$$A = A^T$$

Symmetric matrix A has the following properties:

- A has the same number of rows as columns; symmetric matrices are square.
- $a_{ij} = a_{ji}$ for all i and j. Each element of A on one side of the diagonal equals its mirror on the other side of the diagonal.

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

$$A = A^H$$

A Hermitian matrix has the same number of rows as columns; Hermitian matrices are square. However, in general, a Hermitian matrix is not symmetric, as shown by looking at a complex Hermitian matrix B of order 3, its transpose B^T and its conjugate transpose B^H: $B = B^H$, but $B \neq B^T$:

$$B = \begin{bmatrix} (2,0) & (8,-9) & (27,26) \\ (8,9) & (12,0) & (1,-7) \\ (27,-26) & (1,7) & (-3,0) \end{bmatrix}$$

$$B^T = \begin{bmatrix} (2,0) & (8,9) & (27,-26) \\ (8,-9) & (12,0) & (1,7) \\ (27,26) & (1,-7) & (-3,0) \end{bmatrix}$$

$$B^H = \begin{bmatrix} (2,0) & (8,-9) & (27,26) \\ (8,9) & (12,0) & (1,-7) \\ (27,-26) & (1,7) & (-3,0) \end{bmatrix}$$

The imaginary part of each of the diagonal elements is 0. This is always true for any Hermitian matrix.

The symmetry properties of symmetric matrices and Hermitian matrices enable storage of only the upper-triangular part of the matrix (the diagonal and above) or the lower-triangular part of the matrix (the diagonal and below).

8.2.3 Storage of Symmetric and Hermitian Matrices

All n by n symmetric or Hermitian matrices are stored in one of two ways:

- In either the upper or lower triangle of a two-dimensional array
- Packed in a one-dimensional array

8.2.3.1 Two-Dimensional Upper- or Lower-Triangular Storage

When the upper-triangular part of the matrix is stored in the upper triangle of the array, the strictly lower-triangular part of the array is not referenced. Conversely, when the lower-triangular part of the matrix is stored in the lower triangle of the array, the strictly upper-triangular part of the array is not referenced.

As an example, consider a 4 by 4 real symmetric matrix A:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Upper-triangular storage in a two-dimensional array A is shown in (8–4):

$$\mathrm{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ * & a_{22} & a_{23} & a_{24} \\ * & * & a_{33} & a_{34} \\ * & * & * & a_{44} \end{bmatrix} \tag{8–4}$$

Lower-triangular storage in a two-dimensional array A is shown in (8–5):

$$\mathrm{A} = \begin{bmatrix} a_{11} & * & * & * \\ a_{21} & a_{22} & * & * \\ a_{31} & a_{32} & a_{33} & * \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \tag{8–5}$$

8.2.3.2 One-Dimensional Packed Storage

The total number of elements in an n by n symmetric or Hermitian matrix is n^2. The total number of elements in the upper or lower triangle is as follows:

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

Therefore, when an n by n symmetric or complex Hermitian matrix is stored in a one-dimensional array, $n(n+1)/2$ memory locations are used. The amount of memory saved is as follows:

$$n^2 - \frac{n(n+1)}{2} = \frac{n(n-1)}{2}$$

Such an arrangement is called packed storage. Either the upper triangle of the matrix can be packed sequentially, column by column, or the lower triangle of the matrix can be packed sequentially, column by column.

As an example, consider a 4 by 4 real symmetric matrix A:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Upper triangle packing for A in a one-dimensional array AP is shown in (8–6):

$$\mathrm{AP} = \begin{bmatrix} a_{11} \\ \cdots \\ a_{12} \\ \\ a_{22} \\ \cdots \\ a_{13} \\ \\ a_{23} \\ \\ a_{33} \\ \cdots \\ a_{14} \\ \\ a_{24} \\ \\ a_{34} \\ \\ a_{44} \end{bmatrix} = \begin{bmatrix} a_{11} \\ \cdots \\ a_{21} \\ \\ a_{22} \\ \cdots \\ a_{31} \\ \\ a_{32} \\ \\ a_{33} \\ \cdots \\ a_{41} \\ \\ a_{42} \\ \\ a_{43} \\ \\ a_{44} \end{bmatrix} \qquad (8\text{–}6)$$

For symmetric matrices, packing the upper triangle by columns is equivalent to packing the lower triangle by rows. For Hermitian matrices, the only difference is that the off-diagonal elements are conjugated.

In this packed storage scheme, the ijth element in the upper triangle of the real symmetric matrix is stored in position k of the array, where:

$$k = i + (j(j-1)/2), \quad \text{for} \quad 1 \le i \le j \quad \text{and} \quad 1 \le j \le n$$

For example, element a_{13} is in position $1 + (3(3-1)/2) = 4$ of the array, and element a_{24} is in position $2 + (4(4-1)/2) = 8$ of the array.

The following Fortran program segment transfers the upper triangle of a symmetric matrix from conventional full matrix storage in a two-dimensional array A to upper-triangle packed storage in a one-dimensional array AP:

```
      K=0
      DO 20 J=1,N
         DO 10 I=1,J
            K=K+1
            AP(K)=A(I,J)
10       CONTINUE
20    CONTINUE
```

Lower triangle packing for A in a one-dimensional array AP is shown in (8–7):

$$\mathrm{AP} = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ a_{41} \\ \cdots \\ a_{22} \\ a_{32} \\ a_{42} \\ \cdots \\ a_{33} \\ a_{43} \\ \cdots \\ a_{44} \end{bmatrix} = \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{14} \\ \cdots \\ a_{22} \\ a_{23} \\ a_{24} \\ \cdots \\ a_{33} \\ a_{34} \\ \cdots \\ a_{44} \end{bmatrix} \qquad (8\text{–}7)$$

For symmetric matrices, packing the lower triangle by columns is equivalent to packing the upper triangle by rows. For Hermitian matrices, the only difference is that the off-diagonal elements are conjugated.

In this packed storage scheme, the ijth element in the lower triangle of the real symmetric matrix is stored in position k of the array where:

$$k = i - (j(j-1)/2) + n(j-1), \text{ for } j \le i \le n \text{ and } 1 \le j \le n$$

For example, element a_{31} is in position $3 - 1(1-1)/2 + 4(1-1) = 3$ of the array, and element a_{43} is in position $4 - 3(3-1)/2 + 4(3-1) = 9$ of the array.

The following Fortran program segment transfers the lower triangle of a symmetric matrix from conventional full matrix storage in a two-dimensional array A to lower-triangle packed storage in a one-dimensional array AP:

```
      K=0
      DO 20 J=1,N
         DO 10 I=J,N
            K=K+1
            AP(K)=A(I,J)
10       CONTINUE
20    CONTINUE
```

8.2.4 Triangular Matrices

A triangular matrix is a square matrix whose nonzero elements are all either in the upper-triangular part of the matrix or in the lower-triangular part of the matrix.

In an n by n upper-triangular matrix, $u_{ij} = 0$ for all $i > j$. In addition, for a unit upper-triangular matrix, $u_{ii} = 1$ for $1 \leq i \leq n$.

The matrix U shown in (8–8) is a 4 by 4 upper-triangular matrix:

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} \tag{8–8}$$

In an n by n lower-triangular matrix, $l_{ij} = 0$ for all $i < j$. In addition, for a unit lower-triangular matrix, $l_{ii} = 1$ for $1 \leq i \leq n$.

The matrix L shown in (8–9) is a 4 by 4 lower-triangular matrix:

$$L = \begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \tag{8–9}$$

8.2.5 Storage of Triangular Matrices

When an n by n upper- or lower-triangular matrix is stored conventionally in a two-dimensional array, the $(n-1)$ by $(n-1)$ strictly lower- or upper-triangular part of the array is not referenced by the subroutine. In the case of a unit upper- or lower-triangular matrix, the main diagonal elements of the array are also not referenced, because these elements are assumed to be unity.

As in the case of symmetric and Hermitian matrices, upper- and lower-triangular matrices can be packed in a one-dimensional array. The upper or lower triangle is packed sequentially, column by column. For packed triangular matrices, the same storage layout is used whether or not the diagonal elements are assumed to have the value 1. That is, space is left for the diagonal elements even if those array elements are not referenced.

8.2.6 General Band Matrices

A general band matrix, or band matrix, is a matrix whose nonzero elements are all near the main diagonal such that:

$$a_{ij} = 0 \quad \text{for} \quad (i-j) > kl \quad \text{or} \quad (j-i) > ku$$

The lower bandwidth is kl, the upper bandwidth is ku, and the total bandwidth is $kt = (kl + ku + 1)$. The matrix is said to have kl subdiagonals and ku superdiagonals.

The m by n matrix B shown in (8–10) is a general band matrix where the lower bandwidth is $kl = (g-1)$ and the upper bandwidth is $ku = (p-1)$:

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & . & b_{1p} & 0 & . & . & 0 \\ b_{21} & b_{22} & b_{23} & & & . & & . & . \\ b_{31} & b_{32} & b_{33} & & & & . & & . \\ . & & & . & & & & . & 0 \\ . & & & & . & & & & b_{n-p+1,n} \\ b_{g1} & & & & & & & & . \\ & & & & & . & & & \\ 0 & . & & & & & & & . \\ . & & . & & & & . & & . \\ . & . & & . & & & & . & . \\ . & . & . & & . & & & & \\ 0 & . & . & . & 0 & b_{m,m-g+1} & . & . & b_{mn} \end{bmatrix} \qquad (8\text{–}10)$$

In matrix B, the number ku is the number $(p-1)$ of diagonals above the main diagonal. The number kl is the number $(g-1)$ of diagonals below the main diagonal. Including the main diagonal, the total bandwidth (or the total number of diagonals) is $(kl + ku + 1)$.

The matrix B shown in (8–11) is a 7 by 8 band matrix with bandwidths $kl = 2$ and $ku = 1$ and total bandwidth of 4:

$$L = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 4 & 5 & 0 & 0 & 0 & 0 & 0 \\ 6 & 7 & 8 & 9 & 0 & 0 & 0 & 0 \\ 0 & 10 & 11 & 12 & 13 & 0 & 0 & 0 \\ 0 & 0 & 14 & 15 & 16 & 17 & 0 & 0 \\ 0 & 0 & 0 & 18 & 19 & 20 & 21 & 0 \\ 0 & 0 & 0 & 0 & 22 & 23 & 24 & 25 \end{bmatrix} \qquad (8\text{–}11)$$

8.2.7 Storage of General Band Matrices

When stored in band storage mode, an m by n band matrix with kl subdiagonals and ku superdiagonals is stored in a two-dimensional $(kl + ku + 1)$ by n array. The matrix is stored columnwise so that the nonzero elements of the jth column of the matrix are stored in the jth column of the Fortran array. Consequently, the zero elements of the matrix are not stored in the array.

The main diagonal of the matrix is stored in row $ku + 1$ of the array. The first superdiagonal is stored in row ku starting at column 2. The first subdiagonal is stored in row $ku + 2$ starting at column 1, and so on. Elements of the array that do not correspond to elements in the band matrix, specifically those in the top left ku by ku triangle and those in the bottom right $(n - m + kl)$ by $(n - m + kl)$ triangle, are not referenced by the subroutine.

For example, consider the 5 by 6 band matrix A shown in (8–12) with 1 subdiagonal and 2 superdiagonals. Here, $kl = 1$, $ku = 2$, $m = 5$, and $n = 7$:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & a_{54} & a_{55} & a_{56} \end{bmatrix} \qquad (8\text{–}12)$$

The band matrix A is stored in array ABD as shown in (8–13). Array ABD is 4 by 6. The main diagonal of A is stored in row 3 of ABD. The first superdiagonal is stored in row 3 starting at column 2. The top left 2 by 2 triangle and the bottom right 2 by 2 triangle is not referenced.

$$\text{ABD} = \begin{bmatrix} * & * & a_{13} & a_{24} & a_{35} & a_{46} \\ * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & * \\ a_{21} & a_{32} & a_{43} & a_{54} & * & * \end{bmatrix} \tag{8–13}$$

In this storage scheme, the ijth element of the band matrix is stored in position $(k,\ j)$ of the array, where $k = (i - j + ku + 1)$.

The following Fortran program segment transfers a band matrix from conventional Fortran full matrix storage in A to band storage in array ABD:

```
      DO 20 J=1,N
         K=KU+1-J
         DO 10 I=MAX(1,J-KU),MIN(M,J+KL)
            ABD(K+I,J)=A(I,J)
10       CONTINUE
20    CONTINUE
```

8.2.8 Real Symmetric Band Matrices and Complex Hermitian Band Matrices

A real symmetric band matrix is a real band matrix that is equal to its transpose.

$$B = B^T$$

A real symmetric band matrix is square. It has all its nonzero elements near the main diagonal.

In an n by n real symmetric band matrix B,

$$b_{ij} = b_{ji} \quad \text{for all } i \text{ and } j$$

and

$$b_{ij} = 0 \quad \text{for} \quad (i - j) > k \quad \text{or} \quad (j - i) > k$$

where k is the lower or upper bandwidth. For example, matrix B, shown in (8–14), is a real symmetric band matrix:

$$B = \begin{bmatrix} 2.0 & 3.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 3.0 & 3.0 & -4.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -4.0 & 4.0 & 5.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 5.0 & 5.0 & -6.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & -6.0 & 6.0 & 7.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 7.0 & 7.0 & -8.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -8.0 & 8.0 \end{bmatrix} \tag{8–14}$$

A complex Hermitian band matrix is a square complex band matrix that is equal to its conjugate transpose. It has all its nonzero elements near the main diagonal, but, in general, it is not symmetric. The imaginary part of each of the diagonal elements is 0.

For example, matrix H shown in (8–15) is a complex Hermitian band matrix:

$$H = \begin{bmatrix} (2,0) & (3,1) & (0,0) & (0,0) & (0,0) \\ (3,-1) & (3,0) & (-4,1) & (0,0) & (0,0) \\ (0,0) & (-4,-1) & (4,0) & (5,-1) & (0,0) \\ (0,0) & (0,0) & (5,1) & (5,0) & (-6,1) \\ (0,0) & (0,0) & (0,0) & (-6,-1) & (6,0) \end{bmatrix} \tag{8–15}$$

8.2.9 Storage of Real Symmetric Band Matrices or Complex Hermitian Band Matrices

When stored in band storage mode, an n by n real symmetric or complex Hermitian band matrix with k subdiagonals and k superdiagonals is stored in a two-dimensional $(k+1)$ by n array. Either the upper-triangular band part or the lower-triangular band part of the matrix can be stored.

When the upper-triangle storage mode is used, the nonzero elements of the upper-triangular part of the jth column of the matrix are stored in the jth column of the array. The main diagonal of the matrix is stored in row $(k+1)$ of the array. The first superdiagonal is stored in row k, starting at column 2, and so on. Elements of the array that do not correspond to elements in the band matrix, specifically those in the top left k by k triangle, are not referenced.

As an example, a 6 by 6 real symmetric band matrix A is shown in (8–16). The matrix A has two superdiagonals and two subdiagonals.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{bmatrix} \qquad (8\text{–}16)$$

The matrix A would be stored in upper-triangle storage mode in array ABD as shown in (8–17):

$$\mathrm{ABD} = \begin{bmatrix} * & * & a_{13} & a_{24} & a_{35} & a_{46} \\ * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \end{bmatrix} \qquad (8\text{–}17)$$

In this storage scheme, the nonzero element in the ijth position of the upper triangular part of the symmetric band matrix is stored in position (m, j) of the array, where:

$$m = (i - j + k + 1), \quad \max(1,\ j-k) \le i \le j \ \text{ and } \ 1 \le j \le n$$

The following Fortran program segment transfers the upper-triangular part of a symmetric band matrix from conventional Fortran full matrix storage in A to band storage in array ABD:

```
      DO 20 J=1,N
         M=K+1-J
         DO 10 I=MAX(1,J-K),J
            ABD(M+I,J)=A(I,J)
10       CONTINUE
20    CONTINUE
```

When the lower-triangle storage mode is used, the nonzero elements of the lower-triangular part of the jth column of the matrix are stored in the jth column of the array. The main diagonal of the matrix is stored in row 1 of the array. The first subdiagonal is stored in row 2 starting at column 1, and so on. Elements of the array that do not correspond to elements in the band matrix, specifically those in the bottom right k by k triangle, are not referenced.

As an example, consider the 7 by 7 real symmetric band matrix A, with two superdiagonals and two subdiagonals shown in (8–18):

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} & a_{67} \\ 0 & 0 & 0 & 0 & a_{75} & a_{76} & a_{77} \end{bmatrix} \quad (8–18)$$

The matrix A would be stored in array ABD in lower-triangle storage mode as shown in (8–19):

$$\mathrm{ABD} = \begin{bmatrix} a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} & a_{77} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & a_{76} & * \\ a_{31} & a_{42} & a_{53} & a_{64} & a_{75} & * & * \end{bmatrix} \quad (8–19)$$

In this storage scheme, the nonzero element in the ijth position of the lower-triangular part of the symmetric band matrix is stored in position (m, j) of the array, where:

$$m = (i - j + 1), \quad j \le i \le \min(n,\ j + k) \quad \text{and} \quad 1 \le j \le n$$

The following Fortran program segment transfers the lower-triangular part of a symmetric band matrix A from conventional Fortran full matrix storage to band storage in array ABD:

```
      DO 20 J=1,N
         M=1-J
         DO 10 I=J,MIN(N,J+K)
            ABD(M+I,J)=A(I,J)
10       CONTINUE
20    CONTINUE
```

For a complex Hermitian band matrix, the imaginary parts of the main diagonal are by definition, 0. Therefore, the imaginary parts of the corresponding array elements need not be set, and are assumed to be 0.

8.2.10 Upper- and Lower-Triangular Band Matrices

A triangular band matrix is a square matrix whose nonzero elements are all near the main diagonal and are in either the upper-triangular part of the matrix or the lower-triangular part of the matrix.

In an n by n upper-triangular band matrix U,

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

and

$$u_{ij} = 0 \quad \text{for} \quad (j - i) > ku$$

where ku is the upper bandwidth.

The matrix U shown in (8–20) is a 4 by 4 upper-triangular band matrix:

$$U = \begin{bmatrix} 1.0 & 2.0 & 0.0 & 0.0 \\ 0.0 & 3.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 5.0 & 6.0 \\ 0.0 & 0.0 & 0.0 & 7.0 \end{bmatrix} \quad (8–20)$$

In an n by n lower-triangular band matrix L,

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

and

$$l_{ij} = 0 \quad \text{for} \quad (i - j) > kl$$

where kl is the lower bandwidth.

The matrix L shown in (8–21) is a 4 by 4 lower-triangular band matrix:

$$L = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 2.0 & 3.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 5.0 & 0.0 \\ 0.0 & 0.0 & 6.0 & 7.0 \end{bmatrix} \tag{8–21}$$

8.2.11 Storage of Upper- and Lower-Triangular Band Matrices

Similar to the case of real symmetric band matrices and complex Hermitian band matrices, upper- and lower-triangular band matrices can also be stored in band storage mode.

Upper-triangle storage mode is used for an upper-triangular band matrix. An n by n upper-triangular band matrix with k superdiagonals is stored in a two-dimensional $(k + 1)$ by n array.

When upper-triangle storage mode is used, the nonzero elements of the upper-triangular part of the jth column of the matrix are stored in the jth column of the array. The main diagonal of the matrix is stored in row $(k + 1)$ of the array. The first superdiagonal is stored in row k starting at column 2; and so on. Elements of the array that do not correspond to elements in the band matrix, specifically those in the top left k by k triangle, are not referenced.

As an example, a 6 by 6 upper-triangular band matrix A is shown in (8–22). The matrix A has two superdiagonals.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\ 0 & a_{22} & a_{23} & a_{24} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} & a_{35} & 0 \\ 0 & 0 & 0 & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & 0 & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & 0 & a_{66} \end{bmatrix} \tag{8–22}$$

The matrix A would be stored in upper-triangle storage mode in array ABD as shown in (8–23):

$$\text{ABD} = \begin{bmatrix} * & * & a_{13} & a_{24} & a_{35} & a_{46} \\ * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \end{bmatrix} \tag{8–23}$$

In this storage scheme, the nonzero element in the ijth position of the upper triangular part of the upper-triangular band matrix is stored in position (m, j) of the array, where:

$$m = (i - j + k + 1), \quad \max(1,\ j - k) \leq i \leq j \quad \text{and} \quad 1 \leq j \leq n$$

The following Fortran program segment transfers the upper-triangular part of an upper-triangular band matrix from conventional Fortran full matrix storage in A to band storage in array ABD:

```
      DO 20 J=1,N
         M=K+1-J
         DO 10 I=MAX(1,J-K),J
            ABD(M+I,J)=A(I,J)
10       CONTINUE
20    CONTINUE
```

Lower-triangle storage mode is used for a lower-triangular band matrix. An n by n lower-triangular band matrix with k subdiagonals is stored in a two-dimensional $(k+1)$ by n array.

When lower-triangle storage mode is used, the nonzero elements of the lower-triangular part of the jth column of the matrix are stored in the jth column of the array. The main diagonal of the matrix is stored in row 1 of the array; the first subdiagonal in row 2 starting at column 1; and so on. Elements of the array that do not correspond to elements in the band matrix, specifically those in the bottom right k by k triangle, are not referenced.

As an example, consider the 7 by 7 lower-triangular band matrix A, with two subdiagonals, shown in (8–24):

$$A = \begin{bmatrix} a_{11} & 0 & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & 0 & 0 & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & 0 & 0 \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} & 0 \\ 0 & 0 & 0 & 0 & a_{75} & a_{76} & a_{77} \end{bmatrix} \qquad \text{(8–24)}$$

The matrix A would be stored in array ABD in lower-triangle storage mode as shown in (8–25):

$$\text{ABD} = \begin{bmatrix} a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} & a_{77} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & a_{76} & * \\ a_{31} & a_{42} & a_{53} & a_{64} & a_{75} & * & * \end{bmatrix} \qquad \text{(8–25)}$$

In this storage scheme, the nonzero element in the ijth position of the lower-triangular part of the lower-triangular band matrix is stored in position (m, j) of the array, where:

$$m = (i - j + 1), \quad j \le i \le \min(n,\ j + k) \quad \text{and} \quad 1 \le j \le n$$

The following Fortran program segment transfers the lower-triangular part of a lower-triangular band matrix A from conventional Fortran full matrix storage to band storage in array ABD:

```
      DO 20 J=1,N
         M=1-J
         DO 10 I=J,MIN(N,J+K)
            ABD(M+I,J)=A(I,J)
10       CONTINUE
20    CONTINUE
```

8.3 Naming Conventions for Level 2 BLAS Subprograms

Each Level 2 BLAS subprogram has a name consisting of four or five characters. The first character of the name denotes the Fortran data type of the matrix. The second and third characters denote the type of matrix operated on by the subprogram. The fourth and fifth characters denote the type of operation.

Table 8–1 shows the characters used in the Level 2 BLAS subprogram names and what the characters mean.

Table 8–1 Naming Conventions: Level 2 BLAS Subprograms

Character	Mnemonic	Meaning
First character	S	Single-precision real data
	D	Double-precision real data
	C	Single-precision complex data
	Z	Double-precision complex data
Second and third characters	GE	General matrix
	GB	General band matrix
	HE	Hermitian matrix
	SY	Symmetric matrix
	HP	Hermitian matrix stored in packed form
	SP	Symmetric matrix stored in packed form
	HB	Hermitian band matrix
	SB	Symmetric band matrix
	TR	Triangular matrix
	TP	Triangular matrix stored in packed form
	TB	Triangular band matrix
Fourth and fifth characters	MV	Matrix-vector product
	R	Rank-one update
	RU	Rank-one unconjugated update
	RC	Rank-one conjugated update
	R2	Rank-two update
	SV	Solution of a system of linear equations

For example, the name SGEMV is the subprogram for performing matrix-vector multiplication, where the matrix is a general matrix with single-precision real elements, and the matrix is stored using full matrix storage.

8.4 Summary of Level 2 BLAS Subprograms

Table 8–2 summarizes the Level 2 BLAS subprograms. For the general rank-one update (_GER) operations, two complex subprograms are provided, CGERC and CGERU. This is the only exception to the one-to-one correspondence between real and complex subprograms.

Subprograms for rank-one and rank-two updates applied to band matrices are not provided because these can be obtained by calls to the rank-one and rank-two full matrix subprogram. See Section 8.7 for information about how to make these calls.

Table 8–2 Summary of Level 2 BLAS Subprograms

Subprogram Name	Operation
SGBMV	Calculates, in single-precision arithmetic, a matrix-vector product for either a real general band matrix or its transpose.
DGBMV	Calculates, in double-precision arithmetic, a matrix-vector product for either a real general band matrix or its transpose.
CGBMV	Calculates, in single-precision arithmetic, a matrix-vector product for either a complex general band matrix, its transpose, or its conjugate transpose.
ZGBMV	Calculates, in double-precision arithmetic, a matrix-vector product for either a complex general band matrix, its transpose, or its conjugate transpose.
SGEMV	Calculates, in single-precision arithmetic, a matrix-vector product for either a real general matrix or its transpose.
DGEMV	Calculates, in double-precision arithmetic, a matrix-vector product for either a real general matrix or its transpose.
CGEMV	Calculates, in single-precision arithmetic, a matrix-vector product for either a complex general matrix, its transpose, or its conjugate transpose.
ZGEMV	Calculates, in double-precision arithmetic, a matrix-vector product for either a complex general matrix, its transpose, or its conjugate transpose.
SGER	Calculates, in single-precision arithmetic, a rank-one update of a real general matrix.
DGER	Calculates, in double-precision arithmetic, a rank-one update of a real general matrix.
CGERC	Calculates, in single-precision arithmetic, a rank-one conjugated update of a complex general matrix.
ZGERC	Calculates, in double-precision arithmetic, a rank-one conjugated update of a complex general matrix.
CGERU	Calculates, in single-precision arithmetic, a rank-one unconjugated update of a complex general matrix.
ZGERU	Calculates, in double-precision arithmetic, a rank-one unconjugated update of a complex general matrix.

(continued on next page)

Table 8–2 (Cont.) Summary of Level 2 BLAS Subprograms

Subprogram Name	Operation
SSBMV	Calculates, in single-precision arithmetic, a matrix-vector product for a real symmetric band matrix.
DSBMV	Calculates, in double-precision arithmetic, a matrix-vector product for a real symmetric band matrix.
CHBMV	Calculates, in single-precision arithmetic, a matrix-vector product for a complex Hermitian band matrix.
ZHBMV	Calculates, in double-precision arithmetic, a matrix-vector product for a complex Hermitian band matrix.
SSPMV	Calculates, in single-precision arithmetic, a matrix-vector product for a real symmetric matrix stored in packed form.
DSPMV	Calculates, in double-precision arithmetic, a matrix-vector product for a real symmetric matrix stored in packed form.
CHPMV	Calculates, in single-precision arithmetic, a matrix-vector product for a complex Hermitian matrix stored in packed form.
ZHPMV	Calculates, in double-precision arithmetic, a matrix-vector product for a complex Hermitian matrix stored in packed form.
SSPR	Calculates, in single-precision arithmetic, a rank-one update of a real symmetric matrix stored in packed form.
DSPR	Calculates, in double-precision arithmetic, a rank-one update of a real symmetric matrix stored in packed form.
CHPR	Calculates, in single-precision arithmetic, a rank-one update of a complex Hermitian matrix stored in packed form.
ZHPR	Calculates, in double-precision arithmetic, a rank-one update of a complex Hermitian matrix stored in packed form.
SSPR2	Calculates, in single-precision arithmetic, a rank-two update of a real symmetric matrix stored in packed form.
DSPR2	Calculates, in double-precision arithmetic, a rank-two update of a real symmetric matrix stored in packed form.
CHPR2	Calculates, in single-precision arithmetic, a rank-two update of a complex Hermitian matrix stored in packed form.
ZHPR2	Calculates, in double-precision arithmetic, a rank-two update of a complex Hermitian matrix stored in packed form.
SSYMV	Calculates, in single-precision arithmetic, a matrix-vector product for a real symmetric matrix.
DSYMV	Calculates, in double-precision arithmetic, a matrix-vector product for a real symmetric matrix.
CHEMV	Calculates, in single-precision arithmetic, a matrix-vector product for a complex Hermitian matrix.
ZHEMV	Calculates, in double-precision arithmetic, a matrix-vector product for a complex Hermitian matrix.

(continued on next page)

Table 8–2 (Cont.) Summary of Level 2 BLAS Subprograms

Subprogram Name	Operation
SSYR	Calculates, in single-precision arithmetic, a rank-one update of a real symmetric matrix.
DSYR	Calculates, in double-precision arithmetic, a rank-one update of a real symmetric matrix.
CHER	Calculates, in single-precision arithmetic, a rank-one update of a complex Hermitian matrix.
ZHER	Calculates, in double-precision arithmetic, a rank-one update of a complex Hermitian matrix.
SSYR2	Calculates, in single-precision arithmetic, a rank-two update of a real symmetric matrix.
DSYR2	Calculates, in double-precision arithmetic, a rank-two update of a real symmetric matrix.
CHER2	Calculates, in single-precision arithmetic, a rank-two update of a complex Hermitian matrix.
ZHER2	Calculates, in double-precision arithmetic, a rank-two update of a complex Hermitian matrix.
STBMV	Calculates, in single-precision arithmetic, a matrix-vector product for either a real triangular band matrix or its transpose.
DTBMV	Calculates, in double-precision arithmetic, a matrix-vector product for either a real triangular band matrix or its transpose.
CTBMV	Calculates, in single-precision arithmetic, a matrix-vector product for a complex triangular band matrix, its transpose, or its conjugate transpose.
ZTBMV	Calculates, in double-precision arithmetic, a matrix-vector product for a complex triangular band matrix, its transpose, or its conjugate transpose.
STBSV	Solves, in single-precision arithmetic, a system of linear equations where the coefficient matrix is a real triangular band matrix.
DTBSV	Solves, in double-precision arithmetic, a system of linear equations where the coefficient matrix is a real triangular band matrix.
CTBSV	Solves, in single-precision arithmetic, a system of linear equations where the coefficient matrix is a complex triangular band matrix.
ZTBSV	Solves, in double-precision arithmetic, a system of linear equations where the coefficient matrix is a complex triangular band matrix.
STPMV	Calculates, in single-precision arithmetic, a matrix-vector product for either a real triangular matrix stored in packed form or its transpose.
DTPMV	Calculates, in double-precision arithmetic, a matrix-vector product for either a real triangular matrix stored in packed form or its transpose.
CTPMV	Calculates, in single-precision arithmetic, a matrix-vector product for a complex triangular matrix stored in packed form, its transpose, or its conjugate transpose.

(continued on next page)

Table 8–2 (Cont.) Summary of Level 2 BLAS Subprograms

Subprogram Name	Operation
ZTPMV	Calculates, in double-precision arithmetic, a matrix-vector product for a complex triangular matrix stored in packed form, its transpose, or its conjugate transpose.
STPSV	Solves, in single-precision arithmetic, a system of linear equations where the coefficient matrix is a real triangular matrix stored in packed form.
DTPSV	Solves, in double-precision arithmetic, a system of linear equations where the coefficient matrix is a real triangular matrix stored in packed form.
CTPSV	Solves, in single-precision arithmetic, a system of linear equations where the coefficient matrix is a complex triangular matrix stored in packed form.
ZTPSV	Solves, in double-precision arithmetic, a system of linear equations where the coefficient matrix is a complex triangular matrix stored in packed form.
STRMV	Calculates, in single-precision arithmetic, a matrix-vector product for either a real triangular matrix or its transpose.
DTRMV	Calculates, in double-precision arithmetic, a matrix-vector product for either a real triangular matrix or its transpose.
CTRMV	Calculates, in single-precision arithmetic, a matrix-vector product for a complex triangular matrix, its transpose, or its conjugate transpose.
ZTRMV	Calculates, in double-precision arithmetic, a matrix-vector product for a complex triangular matrix, its transpose, or its conjugate transpose.
STRSV	Solves, in single-precision arithmetic, a system of linear equations where the coefficient matrix is a real triangular matrix.
DTRSV	Solves, in double-precision arithmetic, a system of linear equations where the coefficient matrix is a real triangular matrix.
CTRSV	Solves, in single-precision arithmetic, a system of linear equations where the coefficient matrix is a complex triangular matrix.
ZTRSV	Solves, in double-precision arithmetic, a system of linear equations where the coefficient matrix is a complex triangular matrix.

8.5 Calling Subprograms

All of the Level 2 subprograms are subroutines, and have the following characteristics:

Return a vector or a matrix
Require a CALL statement from a program
Processing overwrites an output argument with the output vector
No **Function Value** section

8.6 Argument Conventions

The subroutines use a list of arguments to specify the requirements and control the result of the subroutine. All arguments are required. The argument list is in the same order for each subprogram:

- Arguments specifying matrix options
- Arguments defining the size of the matrix
- Arguments specifying the input scalar
- Arguments describing the input matrix
- Arguments describing the input vector or vectors
- Arguments specifying the input scalar associated with the input-output vector
- Arguments describing the input-output vector
- Arguments describing the input-output matrix

Not every type of argument is needed by every subprogram.

8.6.1 Specifying Matrix Options

The arguments that specify matrix options are character arguments:

- **trans**
- **uplo**
- **diag**

In Fortran, a character argument can be longer than its corresponding dummy argument. For example, the value 'T' for the argument **trans** can be passed as 'TRANSPOSE'.

trans

In some subroutines, the argument **trans** is used to select the form of the input matrix to use in an operation. You do not change the form of the input matrix in your application program. DXML selects the proper elements, depending on the value of the **trans** argument.

For example, if A is the input matrix, and you want to use it in the operation, set the **trans** argument to 'N'. If you want to use A^T in the operation, set the **trans** argument to 'T'. The subroutine makes the changes and selects the proper elements from the matrix, so that A^T is used. Table 8–3 shows the meaning of the values for the argument **trans**.

Table 8–3 Values for the Argument TRANS

Value of trans	Meaning
'N' or 'n'	Operate with the matrix
'T' or 't'	Operate with the transpose of the matrix
'C' or 'c'	Operate with the conjugate transpose of the matrix

When the operation is performed on a real matrix, the values 'T' and 't' or 'C' and 'c' all have the same meaning.

uplo

The Hermitian, symmetric, and triangular matrix subroutines (HE, SY, and TR subroutines) use the argument **uplo** to specify either the upper or lower triangle. Because of the structure of these matrices, the subroutines do not refer to all of the matrix values. Table 8–4 shows the meaning of the values for the argument **uplo**.

Table 8–4 Values for the Argument UPLO

Value of uplo	Meaning
'U' or 'u'	Refers to the upper triangle
'L' or 'l'	Refers to the lower triangle

diag

The triangular matrix (TR) subroutines use the argument **diag** to specify whether or not the triangular matrix is unit-triangular. Table 8–5 shows the meaning of the values for the argument **diag**.

Table 8–5 Values for the Argument DIAG

Value of diag	Meaning
'U' or 'u'	Unit-triangular
'N' or 'n'	Not unit-triangular

When **diag** is specified as 'U' or 'u', the diagonal elements are not referenced by the subroutine. These elements are assumed to be unity.

8.6.2 Defining the Size of the Matrix

The following arguments define the size of the input-output matrix:

- For a rectangular matrix, m rows by n columns: arguments **m** and **n**
- For a symmetric, Hermitian, or triangular matrix: argument **n**
- For a rectangular band matrix: arguments **m** and **n** for the rows and columns, **kl** for the subdiagonals, and **ku** for the superdiagonals
- For a symmetric, Hermitian, or triangular band matrix: arguments **n** for the dimensions and **k** for the diagonal

You can call a subroutine with arguments **m** or **n** equal to 0 but the subroutine exits immediately without referencing its other arguments.

8.6.3 Describing the Matrix

The description of the matrix depends on how the matrix is stored. The matrix can be stored in one of the following ways:

- Two-dimensional array
- Packed form of a one-dimensional array

Two-Dimensional Array

When the matrix is stored in a two-dimensional array, the subroutine requires the following arguments in addition to the size arguments:

- Argument **a** specifies the array A in which the matrix is stored
- Argument **lda** specifies the leading dimension of the array A

To store the matrix, the array must contain at least the number of elements as described:

$$(n-1)d+l$$

n is the number of columns of the matrix
d is the leading dimension of the array with $d \geq 1$; and
$l = m$ for the GE subroutines,
$l = n$ for the SY, HE, and TR subroutines,
$l = (kl + ku + 1)$ for the GB subroutines, and
$l = (k + 1)$ for the SB, HB, and TB subroutines.

One-Dimensional Array

When the matrix is stored packed in a one-dimensional array, the matrix is described by only one argument, **ap**, which specifies the one-dimensional array in which the matrix is stored.

To store the packed matrix, the array AP must contain at least $n(n+1)/2$ elements.

8.6.4 Describing the Input Scalars

The input scalars, α and β, are always described by the dummy argument names **alpha** and **beta**.

8.6.5 Describing the Vectors

A vector is described by three arguments:

- The length of the vector: **n**

  When the vector x consists of n elements, the corresponding array X must be of length at least $(1 + (n-1)|incx|)$.

- The location of the vector in the array: **x** for the vector X, **y** for the vector y

  The location is the base address of the vector. If the argument is the name of the array, such as X, the location of the vector is specified at X(1), but the location can be specified at any other element of the array. The array can be much larger than the vector that it contains.

- The spacing increment for selecting the vector elements from the array: **incx** for vector x, argument **incy** for vector y

  The increment can be positive or negative, but, unlike the Level 1 Extensions, it cannot be equal to zero.

If you supply an input scalar **beta** of zero, you do not need to set the array Y. This means that an operation such as $y \leftarrow \alpha Ax$ can be performed without having to set y to zero in the calling program.

8.6.6 Invalid Arguments

The following values of the arguments are invalid:

- Any value of the arguments **trans**, **uplo**, or **diag** that is not defined
- **m** < 0 for GE and GB subroutines
- **n** < 0 for all subroutines
- **kl** < 0 for the GB subroutines
- **ku** < 0 for the GB subroutines
- **k** < 0 for the HB, SB, and TB subroutines
- **lda** < **m** for the GE subroutines
- **lda** < **kl** + **ku** + 1 for the GB subroutines
- **lda** < **n** for the HE, SY, and TR subroutines
- **lda** < **k** + 1 for the HB, SB, and TB subroutines
- **incx** = 0
- **incy** = 0

8.7 Rank-One and Rank-Two Updates to Band Matrices

The BLAS 2 subroutines for full matrix updates can be used to perform rank-one and rank-two updates to band matrices. The following operation is the rank-one update to the band matrix A:

$$A \leftarrow A + xy^T$$

Vectors x and y are such that no fill-in occurs outside the band. In this case, the update affects only a full $(kl+1)$ by $(ku+1)$) rectangle within the band matrix A.

The operation is shown in (8–26) for the case where $m = n = 9$, $kl = 2$, and $ku = 3$. The update begins in row l and column l where $l = 3$ and affects only the 12 elements within A that are in the full 3 by 4 rectangle starting at $a_{ll} = a_{33}$.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & 0 & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & 0 & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} & 0 \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} & a_{69} \\ 0 & 0 & 0 & 0 & a_{75} & a_{76} & a_{77} & a_{78} & a_{79} \\ 0 & 0 & 0 & 0 & 0 & a_{86} & a_{87} & a_{88} & a_{89} \\ 0 & 0 & 0 & 0 & 0 & 0 & a_{97} & a_{98} & a_{99} \end{bmatrix}$$

$$+$$

$$\begin{bmatrix} 0 \\ 0 \\ x_3 \\ x_4 \\ x_5 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & y_3 & y_4 & y_5 & y_6 & 0 & 0 & 0 \end{bmatrix} \qquad (8\text{–}26)$$

For real data, SGER with $\alpha = 1$ provides this operation, as shown in the following code fragment:

```
KM=MIN(KL+1,M-L+1)
KN=MIN(KU+1,N-L+1)
CALL SGER (KM, KN, 1., X, 1, Y, 1, A(KU+1,L), MAX(KM,LDA-1))
```

L denotes the starting row and column for the update. The elements x_l and y_l of the vectors x and y are in elements X(l) and Y(l) of the arrays X and Y.

For the case where A is a symmetric band matrix of n by n with k subdiagonals and k superdiagonals, the operation can be achieved by a call to the subroutine SSYR, referring to either the upper or lower triangle of A. To refer to the upper-triangular part of A, use the following call to SSYR:

```
KN=MIN(K+1,N-L+1)
CALL SSYR ('U', KN, 1., X, 1, A(K+1,L), MAX(1,LDA-1))
```

To refer to the lower-triangular part of A, use the following call to SSYR:

```
KN=MIN(K+1,N-L+1)
CALL SSYR ('L', KN, 1., X, 1, A(1,L), MAX(1,LDA-1))
```

If the data is complex, the same operations can be achieved by calls to the subroutines CGER and CHER.

Rank-two updates for real symmetric band matrices and complex Hermitian band matrices can be achieved by calls to the subroutines SSYR2 and CHER2.

8.8 Error Handling

The BLAS Level 2 subroutines provide a check of the input arguments. If you call a Level 2 subroutine with an invalid value for any of its arguments, DXML reports the message and terminates execution of the program.

The code for BLAS Level 2 subroutines has calls to an input argument error handler, the XERBLA routine. When a subroutine detects an error, it passes the name of the subroutine and the number of the first argument that is in error to the XERBLA routine. DXML directs this information to the device or file defined as *stdout*.

8.9 A Look at a Level 2 BLAS Subroutine

SGEMV computes a matrix-vector product for either a real general matrix or its transpose:

$$y \leftarrow \alpha A x + \beta y$$

or

$$y \leftarrow \alpha A^T x + \beta y$$

where α and β are scalars, x and y are vectors, and A is an m by n matrix.

Let A be the following 6 by 6 matrix:

$$A = \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 & 10.0 & 11.0 & 12.0 \\ 13.0 & 14.0 & 15.0 & 16.0 & 17.0 & 18.0 \\ 19.0 & 20.0 & 21.0 & 22.0 & 23.0 & 24.0 \\ 25.0 & 26.0 & 27.0 & 28.0 & 29.0 & 30.0 \\ 31.0 & 32.0 & 33.0 & 34.0 & 35.0 & 36.0 \end{bmatrix}$$

Let x be the vector:

$$x = [\,2.0 \quad 4.0 \quad 6.0 \quad 8.0 \quad 10.0 \quad 12.0 \quad 14.0 \quad 16.0\,]$$

Let y be the vector:

$$y = [\,1.0 \quad 3.0 \quad 5.0 \quad 7.0 \quad 9.0 \quad 11.0 \quad 13.0 \quad 15.0 \quad 17.0\,]$$

The subroutine SGEMV has the following format:

```
CALL SGEMV(trans,m,n,alpha,lda,x,incx,beta,y,incy)
```

The following code calls SGEMV:

```
REAL     A(6,6), ALPHA, BETA, X(8), Y(9)
INTEGER  LDA,INCY,INCX,M,N
CHARACTER TRANS

CALL SGEMV('n', 2, 3, 1.0, a(2,2), 6, x(1), 2, 0.0, y(1), 1)
```

This code multiplies the submatrix

$$\begin{bmatrix} 8.0 & 9.0 & 10.0 \\ 14.0 & 15.0 & 16.0 \end{bmatrix}$$

by the vector $[\,2.0 \quad 6.0 \quad 10.0\,]$ to give the new vector y:

$$[\,170.0 \quad 278.0 \quad 5.0 \quad 7.0 \quad 9.0 \quad 11.0 \quad 13.0 \quad 15.0 \quad 17.0\,]$$

The vector x and the matrix A are unchanged.

The following code uses the transpose:

```
SGEMV('t', 2, 2, 2.0, a, 6, x(3), 1, 1.0, y(1), 1)
```

This code multiplies the transpose of the submatrix

$$\begin{bmatrix} 1.0 & 2.0 \\ 7.0 & 8.0 \end{bmatrix}$$

by the vector

$$\alpha \begin{bmatrix} 6.0 \\ 8.0 \end{bmatrix}$$

where α is 2.0, and then adds the result to β where β is equal to 1.0 times the first two elements of the vector y to produce the new vector y:

$$[\,125.0 \quad 155.0 \quad 5.0 \quad 7.0 \quad 9.0 \quad 11.0 \quad 13.0 \quad 15.0 \quad 17.0\,]$$

The vector x and the matrix A remain unchanged.

Level 2 BLAS Subprograms

This section provides descriptions of the Level 2 BLAS subroutines for real and complex operations.

SGBMV DGBMV CGBMV ZGBMV
Matrix-Vector Product for a General Band Matrix

Format

{S,D,C,Z}GBMV (trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)

Arguments

trans
character*1
On entry, specifies the operation to be performed:

If **trans** = 'N' or 'n', the operation is $y \leftarrow \alpha Ax + \beta y$.
If **trans** = 'T' or 't', the operation is $y \leftarrow \alpha A^T x + \beta y$.
If **trans** = 'C' or 'c', the operation is $y \leftarrow \alpha A^H x + \beta y$.

On exit, **trans** is unchanged.

m
integer*4
On entry, the number of rows of the matrix A; $m \geq 0$.
On exit, **m** is unchanged.

n
integer*4
On entry, the number of columns of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

kl
integer*4
On entry, the number of sub-diagonals of the matrix A; $kl \geq 0$.
On exit, **kl** is unchanged.

ku
integer*4
On entry, the number of super-diagonals of the matrix A; $ku \geq 0$.
On exit, **ku** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar α.
On exit, **alpha** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with dimensions lda by n. The leading m by n part of the array contains the elements of the matrix A, supplied column by column. The leading diagonal of the matrix is stored in row $(ku + 1)$ of the array, the first super-diagonal is stored in row ku starting at position 2, the first sub-diagonal is stored in row $(ku + 2)$ starting at position 1, and so on. Elements in the array A that do not correspond to elements in the matrix (such as the top left ku by ku triangle) are not referenced.
On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of array A; $lda \geq (kl + ku + 1)$.
On exit, **lda** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array containing the vector x. When **trans** is equal to 'N' or 'n', the length of the array is at least $(1 + (n - 1) * |incx|)$. Otherwise, the length is at least $(1 + (m - 1) * |incx|)$.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

beta
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar β.
On exit, **beta** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array containing the vector y. When **trans** is equal to 'N' or 'n', the length of the array is at least $(1 + (m - 1) * |incy|)$. Otherwise, the length is at least $(1 + (n - 1) * |incy|)$.

If β = 0, **y** need not be set. If β is not equal to zero, the incremented array Y must contain the vector y.
On exit, **y** is overwritten by the updated vector y.

incy
integer*4
On entry, the increment for the elements of Y; $incy$ must not equal zero.
On exit, **incy** is unchanged.

Description

The _GBMV subprograms compute a matrix-vector product for either a general band matrix or its transpose:

$$y \leftarrow \alpha Ax + \beta y$$

$$y \leftarrow \alpha A^T x + \beta y$$

In addition to these operations, the CGBMV and ZGBMV subprograms compute a matrix-vector product for the conjugate transpose:

$$y \leftarrow \alpha A^H x + \beta y$$

α and β are scalars, x and y are vectors, and A is an m by n band matrix.

Example

```
COMPLEX*16 A(5,20), X(20), Y(20), ALPHA, BETA
M = 5
N = 20
KL = 2
KU = 2
ALPHA = (1.0D0, 2.0D0)
LDA = 5
INCX = 1
BETA = (0.0D0, 0.0D0)
INCY = 1
CALL ZGBMV('N',M,N,KL,KU,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
```

This Fortran code multiplies a pentadiagonal matrix A by the vector x to get the vector y. The operation is $y \leftarrow Ax$ where A is stored in banded storage form.

SGEMV DGEMV CGEMV ZGEMV
Matrix-Vector Product for a General Matrix (Serial and Parallel Versions)

Format

{S,D,C,Z}GEMV (trans, m, n, alpha, a, lda, x, incx, beta, y, incy)

Arguments

trans
character*1
On entry, specifies the operation to be performed:

If **trans** = 'N' or 'n', the operation is $y \leftarrow \alpha Ax + \beta y$.
If **trans** = 'T' or 't', the operation is $y \leftarrow \alpha A^T x + \beta y$.
If **trans** = 'C' or 'c', the operation is $y \leftarrow \alpha A^H x + \beta y$.

On exit, **trans** is unchanged.

m
integer*4
On entry, the number of rows of the matrix A; $m \geq 0$.
On exit, **m** is unchanged.

n
integer*4
On entry, the number of columns of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar α.
On exit, **alpha** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with dimensions lda by n. The leading m by n part of the array contains the elements of the matrix A.
On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of array A; $lda \geq \max(1, m)$.
On exit, **lda** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array containing the vector x. When **trans** is equal to 'N' or 'n', the length of the array is at least $(1 + (n-1) * |incx|)$. Otherwise, the length is at least $(1 + (m-1) * |incx|)$.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

beta
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar β.
On exit, **beta** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array containing the vector y. When **trans** is equal to 'N' or 'n', the length of the array is at least $(1 + (m - 1) * |incy|)$. Otherwise, the length is at least $(1 + (n - 1) * |incy|)$.

If β = 0, **y** need not be set. If β is not equal to zero, the incremented array Y must contain the vector y.
On exit, **y** is overwritten by the updated vector y.

incy
integer*4
On entry, the increment for the elements of Y; $incy$ must not equal zero.
On exit, **incy** is unchanged.

Description

The _GEMV subprograms compute a matrix-vector product for either a general matrix or its transpose:

$$y \leftarrow \alpha Ax + \beta y$$

$$y \leftarrow \alpha A^T x + \beta y$$

In addition to these operations, the CGEMV and ZGEMV subprograms compute the matrix-vector product for the conjugate transpose:

$$y \leftarrow \alpha A^H x + \beta y$$

α and β are scalars, x and y are vectors, and A is an m by n matrix.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

Examples

1.
```
REAL*8 A(20,20), X(20), Y(20), ALPHA, BETA
INCX = 1
INCY = 1
LDA = 20
M = 20
N = 20
ALPHA = 1.0D0
BETA = 0.0D0
CALL DGEMV('T',M,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
```

This Fortran code computes the product $y \leftarrow A^T x$.

2.
```
COMPLEX A(20,20), X(20), Y(20), ALPHA, BETA
INCX = 1
INCY = 1
LDA = 20
M = 20
N = 20
ALPHA = (1.0, 1.0)
BETA = (0.0, 0.0)
CALL CGEMV('T',M,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
```

This Fortran code computes the product $y \leftarrow A^T x$.

SGER DGER CGERC ZGERC CGERU ZGERU
Rank-One Update of a General Matrix

Format

{S,D}GER (m, n, alpha, x, incx, y, incy, a, lda)

{C,Z}GER{C,U} (m, n, alpha, x, incx, y, incy, a, lda)

Arguments

m
integer*4
On entry, the number of rows of the matrix A; $m \geq 0$.
On exit, **m** is unchanged.

n
integer*4
On entry, the number of columns of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar α.
On exit, **alpha** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (m - 1) * |incx|)$. Array X contains the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array of length at least $(1 + (n - 1) * |incy|)$. The incremented array Y must contain the vector y.
On exit, **y** is unchanged.

incy
integer*4
On entry, the increment for the elements of Y; $incy$ must not equal zero.
On exit, **incy** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with dimensions lda by n. The leading m by n part of the array contains the elements of the matrix A.
On exit, **a** is overwritten by the updated matrix.

lda
integer*4
On entry, the first dimension of A; $lda \geq \max(1, m)$.
On exit, **lda** is unchanged.

Description

SGER and DGER perform a rank-one update of a real general matrix:

$$A \leftarrow \alpha x y^T + A$$

CGERU and ZGERU perform a rank-one update of an unconjugated complex general matrix:

$$A \leftarrow \alpha x y^T + A$$

CGERC and ZGERC perform a rank-one update of a conjugated complex general matrix:

$$A \leftarrow \alpha x y^H + A$$

α is a scalar, x is an m-element vector, y is an n-element vector, and A is an m by n matrix.

Examples

1.
```
REAL*4 A(10,10), X(10), Y(5), ALPHA
INCX = 1
INCY = 1
LDA = 10
M = 3
N = 4
ALPHA = 2.3
CALL SGER(M,N,ALPHA,X,INCX,Y,INCY,A,LDA)
```

This Fortran code computes the rank-1 update $A \leftarrow \alpha x y^T + A$. Only the upper left submatrix of A, of dimension (3,4) and starting at location A(1,1), is updated.

2.
```
COMPLEX A(10,10), X(10), Y(5), ALPHA
INCX = 1
INCY = 1
LDA = 10
M = 3
N = 4
ALPHA = (2.3, 1.2)
CALL CGERC(M,N,ALPHA,X,INCX,Y,INCY,A,LDA)
```

This Fortran code computes the rank-1 update $A \leftarrow \alpha x y^H + A$. Only the upper left submatrix of A, of dimension (3,4) and starting at location A(1,1), is updated.

SSBMV DSBMV CHBMV ZHBMV
Matrix-Vector Product for a Symmetric or Hermitian Band Matrix

Format

{S,D}SBMV (uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)

{C,Z}HBMV (uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)

Arguments

uplo
character*1
On entry, specifies whether the upper- or lower-triangular part of the array A is referenced:

If **uplo** = 'U' or 'u', the upper-triangular part of A is referenced.
If **uplo** = 'L' or 'l', the lower-triangular part of A is referenced.

On exit, **uplo** is unchanged.

n
integer*4
On entry, the order of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

k
integer*4
On entry, if **uplo** specifies the upper portion of matrix A, **k** represents the number of super-diagonals of the matrix. If **uplo** specifies the lower portion, **k** is the number of subdiagonals; $k \geq 0$.
On exit, **k** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar α.
On exit, **alpha** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with dimensions lda by n.

When **uplo** specifies the upper portion of the matrix, the leading $(k+1)$ by n part of the array must contain the upper-triangular band part of the matrix, supplied column by column. The main diagonal of the matrix is stored in row $(k+1)$ of the array, the first super-diagonal is stored in row k starting at position 2, and so on. The top left k by k triangle of the array A is not referenced.

When **uplo** specifies the lower portion of the matrix, the leading $(k+1)$ by n part of the array must contain the lower-triangular band part of the matrix, supplied column by column. The main diagonal of the matrix is stored in row 1 of the array, the first sub-diagonal is stored in row 2, starting at position 1, and so on. The bottom right k by k triangle of the array A is not referenced.

For CHBMV and ZHBMV routines, the imaginary parts of the diagonal elements are not accessed, need not be set, and are assumed to be zero.
On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of array A; $lda \geq (k+1)$.
On exit, **lda** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1+(n-1)*|incx|)$. Array X contains the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

beta
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar β.
On exit, **beta** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array Y of length at least $(1+(n-1)*|incy|)$.

If β = 0, **y** need not be set. If β is not equal to zero, the incremented array Y must contain the vector y.
On exit, **y** is overwritten by the updated vector y.

incy
integer*4
On entry, the increment for the elements of Y; $incy$ must not equal zero.
On exit, **incy** is unchanged.

Description

SSBMV and DSBMV compute a matrix-vector product for a real symmetric band matrix. CHBMV and ZHBMV compute a matrix-vector product for a complex Hermitian band matrix. Both products are described by the following operation:

$$y \leftarrow \alpha Ax + \beta y$$

α and β are scalars, and x and y are vectors with n elements. In the case of SSBMV and DSBMV, A is a symmetric matrix and in the case of CHBMV and ZHBMV, A is a Hermitian matrix.

Example

```
REAL*8 A(2,10), X(10), Y(10), ALPHA, BETA
N = 10
K = 1
ALPHA = 2.0D0
LDA = 2
INCX = 1
BETA = 1.0D0
INCY = 1
CALL DSBMV('U',N,K,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
```

This Fortran code computes the product $y \leftarrow \alpha Ax + y$ where A is a symmetric tridiagonal matrix, with A stored in upper-triangular form.

```
COMPLEX*8 A(2,10), X(10), Y(10), ALPHA, BETA
N = 10
K = 1
ALPHA = (2.0D0, 2.2D0)
LDA = 2
INCX = 1
BETA = (1.0D0, 0.0D0)
INCY = 1
CALL ZHBMV('U',N,K,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
```

This Fortran code computes the product $y \leftarrow \alpha Ax + y$ where A is a Hermitian tridiagonal matrix, with the upper diagonal of A stored.

SSPMV DSPMV CHPMV ZHPMV
Matrix-Vector Product for a Symmetric or Hermitian Matrix Stored in Packed Form

Format

{S,D}SPMV (uplo, n, alpha, ap, x, incx, beta, y, incy)

{C,Z}HPMV (uplo, n, alpha, ap, x, incx, beta, y, incy)

Arguments

uplo
character*1
On entry, specifies whether the upper- or lower-triangular part of the matrix A is supplied in the packed array AP:

If **uplo** = 'U' or 'u', the upper-triangular part of A is supplied.
If **uplo** = 'L' or 'l', the lower-triangular part of A is supplied.

On exit, **uplo** is unchanged.

n
integer*4
On entry, the order of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar α.
On exit, **alpha** is unchanged.

ap
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array AP of length at least $n(n+1)/2$.

If **uplo** specifies the upper triangular part of the matrix A, the array contains those elements of the matrix, packed sequentially, column by column, so that AP(1) contains a_{11}, AP(2) and AP(3) contain a_{12} and a_{22} respectively, and so on.

If **uplo** specifies the lower triangular part to the matrix A, the array contains those elements of the matrix, also packed sequentially, so that AP(1) contains a_{11}, AP(2) and AP(3) contain a_{21} and a_{31} respectively, and so on.

For CHPMV and ZHPMV routines, the imaginary parts of the diagonal elements are not accessed, need not be set, and are assumed to be zero.
On exit, **ap** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1+(n-1)*|incx|)$. Array X contains the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

beta
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar β.
On exit, **beta** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$.

If β = 0, **y** need not be set. If β is not equal to zero, the incremented array Y must contain the vector y.
On exit, **y** is overwritten by the updated vector y.

incy
integer*4
On entry, the increment for the elements of Y; $incy$ must not equal zero.
On exit, **incy** is unchanged.

Description

SSPMV and DSPMV compute a matrix-vector product for a real symmetric matrix stored in packed form. CHPMV and ZHPMV compute a matrix-vector product for a complex Hermitian matrix stored in packed form. Both products are described by the following operation:

$$y \leftarrow \alpha Ax + \beta y$$

α and β are scalars, and x and y are vectors with n elements. A is an n by n matrix. In the case of SSPMV and DSPMV, matrix A is a symmetric matrix and in the case of CHPMV and ZHPMV, matrix A is a Hermitian matrix.

Example

```
COMPLEX*16 AP(250), X(20), Y(20), ALPHA, BETA
N = 20
ALPHA = (2.3D0, 8.4D0)
INCX = 1
BETA = (4.0D0, 3.3D0)
INCY = 1
CALL ZHPMV('L',N,ALPHA,AP,X,INCX,BETA,Y,INCY)
```

This Fortran code computes the product $y \leftarrow \alpha Ax + \beta y$ where A is a Hermitian matrix with its lower-triangular part stored in packed form in AP.

SSPR DSPR CHPR ZHPR
Rank-One Update of a Symmetric or Hermitian Matrix Stored in Packed Form

Format

{S,D}SPR (uplo, n, alpha, x, incx, ap)

{C,Z}HPR (uplo, n, alpha, x, incx, ap)

Arguments

uplo
character*1
On entry, specifies whether the upper- or lower-triangular part of the matrix A is supplied in the packed array AP:

If **uplo** = 'U' or 'u', the upper-triangular part of A is supplied.
If **uplo** = 'L' or 'l', the lower-triangular part of A is supplied.

On exit, **uplo** is unchanged.

n
integer*4
On entry, the order of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar α.
On exit, **alpha** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$. Array X contains the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

ap
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array AP of length at least $n(n+1)/2$.

If **uplo** specifies the upper triangular part of the matrix A, the array contains those elements of the matrix, packed sequentially, column by column, so that AP(1) contains a_{11}, AP(2) and AP(3) contain a_{12} and a_{22} respectively, and so on.

If **uplo** specifies the lower triangular part to the matrix A, the array contains those elements of the matrix, also packed sequentially, so that AP(1) contains a_{11}, AP(2) and AP(3) contain a_{21} and a_{31} respectively, and so on.

For CHPR and ZHPR routines, the imaginary parts of the diagonal elements are not accessed, need not be set, and are assumed to be zero.

On exit, **ap** is overwritten by the specified part of the updated matrix.

Description

SSPR and DSPR perform the rank-one update of a real symmetric matrix stored in packed form:

$$A \leftarrow \alpha x x^T + A$$

CHPR and ZHPR perform the rank-one update of a complex Hermitian matrix stored in packed form:

$$A \leftarrow \alpha x x^H + A$$

α is a scalar, x is vector with n elements, and A is an n by n matrix in packed form. In the case of SSPR and DSPR, matrix A is a symmetric matrix and in the case of CHPR and ZHPR, matrix A is a Hermitian matrix.

Example

```
REAL*8 AP(500), X(30), Y(30), ALPHA
INCX = 1
ALPHA = 1.0D0
N = 30
CALL DSPR('U',N,ALPHA,X,INCX,AP)
```

This Fortran code computes the rank-1 update $A \leftarrow xx^T + A$ where A is a real symmetric matrix, of order 30, with its upper-triangular part stored in packed form in AP.

SSPR2 DSPR2 CHPR2 ZHPR2
Rank-Two Update of a Symmetric or Hermitian Matrix Stored in Packed Form

Format

{S,D}SPR2 (uplo, n, alpha, x, incx, y, incy, ap)

{C,Z}HPR2 (uplo, n, alpha, x, incx, y, incy, ap)

Arguments

uplo
character*1
On entry, specifies whether the upper- or lower-triangular part of the matrix A is supplied in the packed array AP:

If **uplo** = 'U' or 'u', the upper-triangular part of matrix A is supplied.
If **uplo** = 'L' or 'l', the lower-triangular part of matrix A is supplied.

On exit, **uplo** is unchanged.

n
integer*4
On entry, the order of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar α.
On exit, **alpha** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$. Array X contains the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$. The incremented array Y must contain the vector y.
On exit, **y** is unchanged.

incy
integer*4
On entry, the increment for the elements of Y; $incy$ must not equal zero.
On exit, **incy** is unchanged.

ap
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array AP of length at least $n(n+1)/2$.

If **uplo** specifies the upper triangular part of the matrix A, the array contains those elements of the matrix, packed sequentially, column by column, so that AP(1) contains a_{11}, AP(2) and AP(3) contain a_{12} and a_{22} respectively, and so on.

If **uplo** specifies the lower triangular part to the matrix A, the array contains those elements of the matrix, also packed sequentially, so that AP(1) contains a_{11}, AP(2) and AP(3) contain a_{21} and a_{31} respectively, and so on.

For CHPR2 and ZHPR2 routines, the imaginary parts of the diagonal elements are not accessed, need not be set, and are assumed to be zero.

On exit, **ap** is overwritten by the specified part of the updated matrix.

Description

SSPR2 and DSPR2 perform the rank-two update of a real symmetric matrix stored in packed form:

$$A \leftarrow \alpha x y^T + \alpha y x^T + A$$

CHPR2 and ZHPR2 perform the rank-two update of a complex Hermitian matrix stored in packed form:

$$A \leftarrow \alpha x y^H + \overline{\alpha} y x^H + A$$

α is a scalar, x is vector with n elements, and A is an n by n matrix in packed form. In the case of SSPR2 and DSPR2, matrix A is a symmetric matrix and in the case of CHPR2 and ZHPR2, matrix A is a Hermitian matrix.

Example

```
REAL*4 AP(250), X(20), Y(20), ALPHA
INCX = 1
INCY = 1
ALPHA = 2.0
N = 20
CALL SSPR2('L',N,ALPHA,X,INCX,Y,INCY,AP)
```

This Fortran code computes the rank-2 update of a real symmetric matrix A, given by $A \leftarrow \alpha x y^T + \alpha y x^T + A$. A is a real symmetric matrix, of order 20, with its lower-triangular part stored in packed form in AP.

SSYMV DSYMV CHEMV ZHEMV
Matrix-Vector Product for a Symmetric or Hermitian Matrix

Format

{S,D}SYMV (uplo, n, alpha, a, lda, x, incx, beta, y, incy)

{C,Z}HEMV (uplo, n, alpha, a, lda, x, incx, beta, y, incy)

Arguments

uplo
character*1
On entry, specifies whether the upper- or lower-triangular part of the array A is referenced:

If **uplo** = 'U' or 'u', the upper-triangular part of A is referenced.
If **uplo** = 'L' or 'l', the lower-triangular part of A is referenced.

On exit, **uplo** is unchanged.

n
integer*4
On entry, the order of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar α.
On exit, **alpha** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with dimensions lda by n.

When **uplo** specifies the upper portion of the matrix, the leading n by n part of the array contains the upper-triangular part of the matrix, and the lower-triangular part of array A is not referenced.

When **uplo** specifies the lower portion of the matrix, the leading n by n part of the array contains the lower-triangular part of the matrix, and the upper-triangular part of array A is not referenced.

For CHEMV and ZHEMV routines, the imaginary parts of the diagonal elements are not accessed, need not be set, and are assumed to be zero.
On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of array A; $lda \geq \max(1, n)$.
On exit, **lda** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$. Array X contains the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

beta
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar β.
On exit, **beta** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$.

If β = 0, **y** need not be set. If β is not equal to zero, the incremented array Y must contain the vector y.
On exit, **y** is overwritten by the updated vector y.

incy
integer*4
On entry, the increment for the elements of Y; $incy$ must not equal zero.
On exit, **incy** is unchanged.

Description

SSYMV and DSYMV compute a matrix-vector product for a real symmetric matrix. CHEMV and ZHEMV compute a matrix-vector product for a complex Hermitian matrix. Both products are described by the following operation:

$$y \leftarrow \alpha Ax + \beta y$$

α and β are scalars, x and y are vectors with n elements, and A is an n by n matrix. In the case of SSYMV and DSYMV, matrix A is a symmetric matrix and in the case of CHEMV and ZHEMV, matrix A is a Hermitian matrix.

Examples

1.
```
REAL*8 A(100,40), X(40), Y(40), ALPHA, BETA
N = 40
INCX = 1
INCY = 1
ALPHA = 1.0D0
BETA = 0.0D0
LDA = 100
CALL DSYMV('U',N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
```

This Fortran code computes the product $y \leftarrow Ax$ where A is a symmetric matrix, of order 40, with its upper-triangular part stored.

2.
```
COMPLEX A(100,40), X(40), Y(40), ALPHA, BETA
N = 40
INCX = 1
INCY = 1
ALPHA = (1.0, 0.5)
BETA = (0.0, 0.0)
LDA = 100
CALL CHEMV('U',N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
```

This Fortran code computes the product $y \leftarrow Ax$ where A is a Hermitian matrix, of order 40, with its upper-triangular part stored.

SSYR DSYR CHER ZHER
Rank-One Update of a Symmetric or Hermitian Matrix

Format

{S,D}SYR (uplo, n, alpha, x, incx, a, lda)

{C,Z}HER (uplo, n, alpha, x, incx, a, lda)

Arguments

uplo
character*1
On entry, specifies whether the upper- or lower-triangular part of the array A is referenced:

If **uplo** = 'U' or 'u', the upper-triangular part of A is referenced.
If **uplo** = 'L' or 'l', the lower-triangular part of A is referenced.

On exit, **uplo** is unchanged.

n
integer*4
On entry, the order of the matrix A and the number of elements in vector x; $n \geq 0$.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar α.
On exit, **alpha** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$. Array X contains the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with dimensions lda by n.

When **uplo** specifies the upper portion of the matrix, the leading n by n part of the array contains the upper-triangular part of the matrix, and the lower-triangular part of array A is not referenced.

When **uplo** specifies the lower portion of the matrix, the leading n by n part of the array contains the lower-triangular part of the matrix, and the upper-triangular part of array A is not referenced.

For CHER and ZHER routines, the imaginary parts of the diagonal elements are not accessed, need not be set, and are assumed to be zero.

On exit, **a** is overwritten; the specified part of the array A is overwritten by the part of the updated matrix.

lda
integer*4
On entry, the first dimension of array A; $lda \geq \max(1, n)$.
On exit, **lda** is unchanged.

Description

SSYR and DSYR perform the rank-one update of a real symmetric matrix:

$$A \leftarrow \alpha x x^T + A$$

CHER and ZHER perform the rank-one update of a complex Hermitian matrix:

$$A \leftarrow \alpha x x^H + A$$

α is a scalar, x is vector with n elements, and A is an n by n matrix in packed form. In the case of SSYR and DSYR, matrix A is a symmetric matrix and in the case of CHER and ZHER, matrix A is a Hermitian matrix.

Examples

1.
```
REAL*4 A(50,20), X(20), ALPHA
INCX = 1
LDA = 50
N = 20
ALPHA = 2.0
CALL SSYR('L',N,ALPHA,X,INCX,A,LDA)
```

This Fortran code computes the rank-1 update of the matrix A, given by $A \leftarrow \alpha x x^T + A$. A is a real symmetric matrix with its lower-triangular part stored.

2.
```
COMPLEX*16 A(50,20), X(20), ALPHA
INCX = 1
LDA = 50
N = 20
ALPHA = (2.0D0, 1.0D0)
CALL ZHER('L',N,ALPHA,X,INCX,A,LDA)
```

This Fortran code computes the rank-1 update of the matrix A, given by $A \leftarrow \alpha x x^H + A$. A is a complex Hermitian matrix with its lower-triangular part stored.

SSYR2 DSYR2 CHER2 ZHER2
Rank-Two Update of a Symmetric or Hermitian Matrix

Format

{S,D}SYR2 (uplo, n, alpha, x, incx, y, incy, a, lda)

{C,Z}HER2 (uplo, n, alpha, x, incx, y, incy, a, lda)

Arguments

uplo
character*1
On entry, specifies whether the upper- or lower-triangular part of the array A is referenced:

If **uplo** = 'U' or 'u', the upper-triangular part of A is referenced.
If **uplo** = 'L' or 'l', the lower-triangular part of A is referenced.

On exit, **uplo** is unchanged.

n
integer*4
On entry, the order of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, the scalar α.
On exit, **alpha** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$. Array X contains the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$. The incremented array Y must contain the vector y.
On exit, **y** is unchanged.

incy
integer*4
On entry, the increment for the elements of Y; $incy$ must not equal zero.
On exit, **incy** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with dimensions lda by n.

When **uplo** specifies the upper portion of the matrix, the leading n by n part of the array contains the upper-triangular part of the matrix, and the lower-triangular part of array A is not referenced.

When **uplo** specifies the lower portion of the matrix, the leading n by n part of the array contains the lower-triangular part of the matrix, and the upper-triangular part of array A is not referenced.

For complex routines, the imaginary parts of the diagonal elements need not be set. They are assumed to be 0, and on exit they are set to 0.

On exit, **a** is overwritten; the specified part of the array A is overwritten by the specified part of the updated matrix.

lda
integer*4
On entry, the first dimension of array A; $lda \geq \max(1, n)$.
On exit, **lda** is unchanged.

Description

SSYR2 and DSYR2 perform the rank-two update of a real symmetric matrix:

$$A \leftarrow \alpha x y^T + \alpha y x^T + A$$

CHER2 and ZHER2 perform the rank-two update of a complex Hermitian matrix:

$$A \leftarrow \alpha x y^H + \overline{\alpha} y x^H + A$$

α is a scalar, x and y are vectors with n elements, and A is an n by n matrix. In the case of SSYR2 and DSYR2, matrix A is a symmetric matrix and in the case of CHER2 and ZHER2, matrix A is a Hermitian matrix.

Example

```
REAL*8 A(50,20), X(20), Y(20), ALPHA
INCX = 1
LDA = 50
N = 20
INCY = 1
ALPHA = 1.0D0
CALL DSYR2('U',N,ALPHA,X,INCX,Y,INCY,A,LDA)
```

This Fortran code computes the rank-2 update of a real symmetric matrix A, given by $A \leftarrow xy^T + yx^T + A$. Only the upper-triangular part of A is stored.

STBMV DTBMV CTBMV ZTBMV
Matrix-Vector Product for a Triangular Band Matrix

Format

{S,D,C,Z}TBMV (uplo, trans, diag, n, k, a, lda, x, incx)

Arguments

uplo
character*1
On entry, specifies whether the matrix A is an upper- or lower-triangular matrix:

If **uplo** = 'U' or 'u', A is an upper-triangular matrix.
If **uplo** = 'L' or 'l', A is a lower-triangular matrix.

On exit, **uplo** is unchanged.

trans
character*1
On entry, specifies the operation to be performed:

If **trans** = 'N' or 'n', the operation is $y \leftarrow \alpha Ax + \beta y$.
If **trans** = 'T' or 't', the operation is $y \leftarrow \alpha A^T x + \beta y$.
If **trans** = 'C' or 'c', the operation is $y \leftarrow \alpha A^H x + \beta y$.

On exit, **trans** is unchanged.

diag
character*1
On entry, specifies whether the matrix A is unit-triangular:

If **diag** = 'U' or 'u', A is a unit-triangular matrix.
If **diag** = 'N' or 'n', A is not a unit-triangular matrix.

On exit, **diag** is unchanged.

n
integer*4
On entry, the order of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

k
integer*4
On entry, if **uplo** is equal to 'U' or 'u', the number of super-diagonals k of the matrix A. If **uplo** is equal to 'L' or 'l', the number of sub-diagonals k of the matrix A; $k \geq 0$.
On exit, **k** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with dimensions lda by n.

When **uplo** specifies the upper portion of the matrix, the leading $(k+1)$ by n part of the array must contain the upper-triangular band part of the matrix, supplied column by column. The main diagonal of the matrix is stored in row $(k+1)$ of the array, the first super-diagonal is stored in row k starting at position 2, and so on. The bottom left k by k triangle of the array A is not referenced.

When **uplo** specifies the lower portion of the matrix, the leading $(k+1)$ by n part of the array must contain the lower-triangular band part of the matrix, supplied column by column. The main diagonal of the matrix is stored in row 1 of the array, the first sub-diagonal is stored in row 2, starting at position 1, and so on. The top right k by k triangle of the array A is not referenced.

If **diag** is equal to 'U' or 'u', the elements of the array A corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity. On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of array A; $lda \geq (k+1)$.
On exit, **lda** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1+(n-1)*|incx|)$. Array X contains the vector x.
On exit, **x** is overwritten with the transformed vector x.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

Description

The _TBMV subprograms compute a matrix-vector product for a triangular band matrix or its transpose: $x \leftarrow Ax$ or $x \leftarrow A^T x$.

In addition to these operations, the CTBMV and ZTBMV subprograms compute the matrix-vector product for the conjugate transpose: $x \leftarrow A^H x$.

x is a vector with n elements and A is an n by n band matrix, with $(k+1)$ diagonals. The band matrix is a unit or nonunit, upper- or lower-triangular matrix.

Example

```
REAL*4 A(5,100), X(100)
INCX = 1
LDA = 5
K = 4
N = 100
CALL STBMV('U','N','N',N,K,A,LDA,X,INCX)
```

This Fortran code computes the product $x \leftarrow Ax$ where A is an upper-triangular, nonunit diagonal matrix, with 4 superdiagonals.

STBSV DTBSV CTBSV ZTBSV
Solver of a System of Linear Equations with a Triangular Band Matrix

Format

{S,D,C,Z}TBSV (uplo, trans, diag, n, k, a, lda, x, incx)

Arguments

uplo
character*1
On entry, specifies whether the matrix A is an upper- or lower-triangular matrix:

If **uplo** = 'U' or 'u', A is an upper-triangular matrix.
If **uplo** = 'L' or 'l', A is a lower-triangular matrix.

On exit, **uplo** is unchanged.

trans
character*1
On entry, specifies the system to be solved:

If **trans** = 'N' or 'n', the system is $Ax = b$.
If **trans** = 'T' or 't', the system is $A^T x = b$.
If **trans** = 'C' or 'c', the system is $A^H x = b$.

On exit, **trans** is unchanged.

diag
character*1
On entry, specifies whether the matrix A is unit-triangular:

If **diag** = 'U' or 'u', A is a unit-triangular matrix.
If **diag** = 'N' or 'n', A is not a unit-triangular matrix.

On exit, **diag** is unchanged.

n
integer*4
On entry, the order of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

k
integer*4
On entry, if **uplo** is equal to 'U' or 'u', the number of super-diagonals k of the matrix A. If **uplo** is equal to 'L' or 'l', the number of sub-diagonals k of the matrix A; $k \geq 0$.
On exit, **k** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with dimensions lda by n.

When **uplo** specifies the upper portion of the matrix, the leading $(k + 1)$ by n part of the array contains the upper-triangular band part of the matrix, supplied column by column. The main diagonal of the matrix is stored in row $(k + 1)$ of the array, the first super-diagonal is stored in row k starting at position 2, and so on. The top left k by k triangle of the array A is not referenced.

When **uplo** specifies the lower portion, the leading $(k + 1)$ by n part of the array contains the lower-triangular band part of the matrix, supplied column by column. The main diagonal of the matrix is stored in row 1 of the array, the first sub-diagonal is stored in row 2 starting at position 1, and so on. The top right k by k triangle of the array A is not referenced.

If **diag** is equal to 'U' or 'u', the elements of the array A corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity. On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of array A; $lda \geq (k + 1)$.
On exit, **lda** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$. Array X contains the vector b.
On exit, **x** is overwritten with the solution vector x.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

Description

The _TBSV subprograms solve one of the following systems of linear equations for x: $Ax = b$ or $A^T x = b$. In addition to these operations, the CTBSV and ZTBSV subprograms solve the following system of linear equations for x: $A^H x = b$.

b and x are vectors with n elements and A is an n by n band matrix with $(k + 1)$ diagonals. The matrix is a unit or nonunit, upper- or lower-triangular band matrix.

The _TBSV routines do not perform checks for singularity or near singularity of the triangular matrix. The requirements for such a test depend on the application. If necessary, perform the test in your application program before calling the routine.

Example

```
REAL*8 A(10,100), X(100)
INCX = 1
K = 9
LDA = 10
N = 100
CALL DTBSV('L','T','U',N,K,A,LDA,X,INCX)
```

This Fortran code solves the system $A^T x = b$ where A is a lower-triangular matrix, with a unit diagonal and 9 subdiagonals. The right hand side b is originally contained in the vector x.

STPMV DTPMV CTPMV ZTPMV
Matrix-Vector Product for a Triangular Matrix in Packed Form

Format

{S,D,C,Z}TPMV (uplo, trans, diag, n, ap, x, incx)

Arguments

uplo
character*1
On entry, specifies whether the matrix A is an upper- or lower-triangular matrix:

If **uplo** = 'U' or 'u', A is an upper-triangular matrix.
If **uplo** = 'L' or 'l', A is a lower-triangular matrix.

On exit, **uplo** is unchanged.

trans
character*1
On entry, specifies the operation to be performed:

If **trans** = 'N' or 'n', the operation is $x \leftarrow Ax$.
If **trans** = 'T' or 't', the operation is $x \leftarrow A^T x$.
If **trans** = 'C' or 'c', the operation is $x \leftarrow A^H x$.

On exit, **trans** is unchanged.

diag
character*1
On entry, specifies whether the matrix A is unit-triangular:

If **diag** = 'U' or 'u', A is a unit-triangular matrix.
If **diag** = 'N' or 'n', A is not a unit-triangular matrix.

On exit, **diag** is unchanged.

n
integer*4
On entry, the order of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

ap
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array AP of length at least $\frac{n(n+1)}{2}$.

If **uplo** specifies the upper triangular part of the matrix A, the array contains those elements of the matrix, packed sequentially, column by column, so that AP(1) contains a_{11}, AP(2) and AP(3) contain a_{12} and a_{22} respectively, and so on.

If **uplo** specifies the lower triangular part to the matrix A, so that AP(1) contains a_{11}, AP(2) and AP(3) contain a_{21} and a_{31} respectively, and so on.

If **diag** is equal to 'U' or 'u', the diagonal elements of A are not referenced, but are assumed to be unity.

On exit, **ap** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$. Array X contains the vector x.
On exit, **x** is overwritten with the transformed vector x.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

Description

The _TPMV subprograms compute a matrix-vector product for a triangular matrix stored in packed form or its transpose: $x \leftarrow Ax$ or $x \leftarrow A^T x$. In addition to these operations, the CTPMV and ZTPMV subprograms compute a matrix-vector product for the conjugate transpose: $x \leftarrow A^H x$.

x is a vector with n elements and A is an n by n, unit or nonunit, upper- or lower-triangular matrix, supplied in packed form.

Example

```
REAL*4 AP(250), X(20)
INCX = 1
N = 20
CALL STPMV('U','N','N',N,AP,X,INCX)
```

This Fortran code computes the product $x \leftarrow Ax$ where A is an upper-triangular matrix of order 20, with nonunit diagonal, stored in packed form.

STPSV DTPSV CTPSV ZTPSV
Solve a System of Linear Equations with a Triangular Matrix in Packed Form

Format

{S,D,C,Z}TPSV (uplo, trans, diag, n, ap, x, incx)

Arguments

uplo
character*1
On entry, specifies whether the matrix A is an upper- or lower-triangular matrix:

If **uplo** = 'U' or 'u', A is an upper-triangular matrix.
If **uplo** = 'L' or 'l', A is a lower-triangular matrix.

On exit, **uplo** is unchanged.

trans
character*1
On entry, specifies the system to be solved:

If **trans** = 'N' or 'n', the system is $Ax = b$.
If **trans** = 'T' or 't', the system is $A^T x = b$.
If **trans** = 'C' or 'c', the system is $A^H x = b$.

On exit, **trans** is unchanged.

diag
character*1
On entry, specifies whether the matrix A is unit-triangular:

If **diag** = 'U' or 'u', A is a unit-triangular matrix.
If **diag** = 'N' or 'n', A is not a unit-triangular matrix.

On exit, **diag** is unchanged.

n
integer*4
On entry, the order of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

ap
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array AP of length at least $n(n+1)/2$.

If **uplo** specifies the upper triangular part of the matrix A, the array contains those elements of the matrix, packed sequentially, column by column, so that AP(1) contains a_{11}, AP(2) and AP(3) contain a_{12} and a_{22} respectively, and so on.

If **uplo** specifies the lower triangular part to the matrix A, so that AP(1) contains a_{11}, AP(2) and AP(3) contain a_{21} and a_{31} respectively, and so on.

If **diag** is equal to 'U' or 'u', the diagonal elements of A are not referenced, but are assumed to be unity.
On exit, **ap** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$. Array X contains the vector b.
On exit, **x** is overwritten with the solution vector x.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

Description

The _TPSV subprograms solve one of the following systems of linear equations for x: $Ax = b$ or $A^T x = b$. In addition to these operations, the CTPSV and ZTPSV subprograms solve the following system of equations: $A^H x = b$.

b and x are vectors with n elements and A is an n by n, unit or nonunit, upper- or lower-triangular matrix, supplied in packed form.

The _TPSV routines do not perform checks for singularity or near singularity of the triangular matrix. The requirements for such a test depend on the application. If necessary, perform the test in your application program before calling this routine.

Example

```
REAL*8 A(500), X(30)
INCX = 1
N = 30
CALL DTPSV('L','T','N',N,AP,X,INCX)
```

This Fortran code solves the system $A^T x = b$ where A is a lower-triangular matrix of order 30, with nonunit diagonal, stored in packed form. The right hand side b is originally contained in the vector x.

STRMV DTRMV CTRMV ZTRMV
Matrix-Vector Product for a Triangular Matrix

Format

{S,D,C,Z}TRMV (uplo, trans, diag, n, a, lda, x, incx)

Arguments

uplo
character*1
On entry, specifies whether the matrix A is an upper- or lower-triangular matrix:

If **uplo** = 'U' or 'u', A is an upper-triangular matrix.
If **uplo** = 'L' or 'l', A is a lower-triangular matrix.

On exit, **uplo** is unchanged.

trans
character*1
On entry, specifies the operation to be performed:

If **trans** = 'N' or 'n', the operation is $x \leftarrow Ax$.
If **trans** = 'T' or 't', the operation is $x \leftarrow A^T x$.
If **trans** = 'C' or 'c', the operation is $x \leftarrow A^H x$.

On exit, **trans** is unchanged.

diag
character*1
On entry, specifies whether the matrix A is unit-triangular:

If **diag** = 'U' or 'u', A is a unit-triangular matrix.
If **diag** = 'N' or 'n', A is not a unit-triangular matrix.

On exit, **diag** is unchanged.

n
integer*4
On entry, the order of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with dimensions lda by n.

When **uplo** specifies the upper portion of the matrix, the leading n by n part of the array contains the upper-triangular part of the matrix, and the lower-triangular part of array A is not referenced.

When **uplo** specifies the lower portion of the matrix, the leading n by n part of the array contains the lower-triangular part of the matrix, and the upper-triangular part of array A is not referenced.

If **diag** is equal to 'U' or 'u', the diagonal elements of A are also not referenced, but are assumed to be unity.
On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of array A; $lda \geq \max(1, n)$.
On exit, **lda** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$. Array X contains the vector x.
On exit, **x** is overwritten with the transformed vector x.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

Description

The _TRMV subprograms compute a matrix-vector product for a triangular matrix or its transpose: $x \leftarrow Ax$ or $x \leftarrow A^T x$. In addition to these operations, the CTRMV and ZTRMV subprograms compute a matrix-vector product for conjugate transpose: $x \leftarrow A^H x$.

x is a vector with n elements, and A is an n by n, unit or nonunit, upper- or lower-triangular matrix.

Example

```
REAL*4 A(50,20), X(20)
INCX = 1
N = 20
LDA = 50
CALL STRMV('U','N','N',N,A,LDA,X,INCX)
```

This Fortran code computes the product $x \leftarrow Ax$ where A is an upper-triangular matrix, of order 20, with a nonunit diagonal.

STRSV DTRSV CTRSV ZTRSV
Solver of a System of Linear Equations with a Triangular Matrix

Format

{S,D,C,Z}TRSV (uplo, trans, diag, n, a, lda, x, incx)

Arguments

uplo
character*1
On entry, specifies whether the matrix A is an upper- or lower-triangular matrix:

If **uplo** = 'U' or 'u', A is an upper-triangular matrix.
If **uplo** = 'L' or 'l', A is a lower-triangular matrix.

On exit, **uplo** is unchanged.

trans
character*1
On entry, specifies the system to be solved:

If **trans** = 'N' or 'n', the system is $Ax = b$.
If **trans** = 'T' or 't', the system is $A^T x = b$.
If **trans** = 'C' or 'c', the system is $A^H x = b$.

On exit, **trans** is unchanged.

diag
character*1
On entry, specifies whether the matrix A is unit-triangular:

If **diag** = 'U' or 'u', A is a unit-triangular matrix.
If **diag** = 'N' or 'n', A is not a unit-triangular matrix.

On exit, **diag** is unchanged.

n
integer*4
On entry, the order of the matrix A; $n \geq 0$.
On exit, **n** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with dimensions lda by n.

When **uplo** specifies the upper portion of the matrix, the leading n by n part of the array contains the upper-triangular part of the matrix, and the lower-triangular part of array A is not referenced.

When **uplo** specifies the lower portion of the matrix, the leading n by n part of the array contains the lower-triangular part of the matrix, and the upper-triangular part of array A is not referenced.

If **diag** is equal to 'U' or 'u', the diagonal elements of A are also not referenced, but are assumed to be unity.
On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of array A; $lda \geq \max(1, n)$.
On exit, **lda** is unchanged.

x
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$. Array X contains the vector b.
On exit, **x** is overwritten with the solution vector x.

incx
integer*4
On entry, the increment for the elements of X; $incx$ must not equal zero.
On exit, **incx** is unchanged.

Description

The _TRSV subprograms solve one of the following systems of linear equations for x: $Ax = b$ or $A^T x = b$. In addition to these operations, the CTRSV and ZTRSV subprograms solve the following systems of linear equation: $A^H x = b$.

b and x are vectors with n elements and A is an n by n, unit or nonunit, upper- or lower-triangular matrix.

The _TRSV routines do not perform checks for singularity or near singularity of the triangular matrix. The requirements for such a test depend on the application. If necessary, perform the test in your application program before calling this routine.

Example

```
REAL*8 A(100,40), X(40)
INCX = 1
N = 40
LDA = 100
CALL DTRSV('L','N','U',N,A,LDA,X,INCX)
```

This Fortran code solves the system $Ax = b$ where A is a lower-triangular matrix of order 40, with a unit diagonal. The right hand side b is originally stored in the vector x.

9

Using the Level 3 BLAS Subprograms

The Level 3 BLAS subprograms perform matrix-matrix operations commonly occurring in many computational problems in linear algebra. This chapter provides information about the following topics:

- Operations performed by the Level 3 BLAS subprograms (Section 9.1)
- Level 3 vector and matrix storage (Section 9.1.2)
- Subprogram naming conventions (Section 9.1.3)
- Subprogram summary (Section 9.2)
- Calling Level 3 BLAS subprograms (Section 9.3)
- Arguments used in the subprograms and invalid arguments (Sections 9.4 and 9.4.5)
- Error handling (Section 9.5)
- A look at some Level 3 subprograms and their use (Section 9.6)

The reference descriptions of the Level 3 BLAS subprograms are at the end of this chapter.

A key Level 3 BLAS subprogram, {C,D,S,Z}GEMM, has been parallelized for improved peformance on multiprocessor systems. For information about using the parallel library, see Chapter 4.

9.1 Level 3 BLAS Operations

The BLAS Level 3 subprograms perform operations that involve two or three matrices. The operations do not involve vectors.

9.1.1 Types of Operations

The subprograms perform seven types of basic matrix-matrix operations:

- Matrix addition operations

  $$C \leftarrow \alpha \mathrm{op}(A) + \beta \mathrm{op}(B))$$

  where $\mathrm{op}(X) = X, X^T, \overline{X}, or X^H$
- Matrix multiply-and-add operations

  $$C \leftarrow \alpha \mathrm{op}(A) \mathrm{op}(B) + \beta C$$

  where $\mathrm{op}(X) = X, X^T, \overline{X}, or X^H$
- Matrix subtraction operations

  $$C \leftarrow \alpha \mathrm{op}(A) - \beta \mathrm{op}(B)$$

  where $\mathrm{op}(X) = X, X^T, \overline{X}, or X^H$

- Miscellaneous matrix operations

$$C \leftarrow \alpha \mathrm{op}(A)$$

  where $\mathrm{op}(X) = X, X^T, \overline{X}, or X^H$

- Rank-k and rank-2k updates of a symmetric matrix

$$C \leftarrow \alpha A A^T + \beta C$$

$$C \leftarrow \alpha A^T A + \beta C$$

$$C \leftarrow \alpha A B^T + \alpha B A^T + \beta C$$

$$C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$$

- Matrix-triangular matrix multiply operations

$$B \leftarrow TB$$

$$B \leftarrow T^T B$$

$$B \leftarrow BT$$

$$B \leftarrow BT^T$$

- Solution of triangular systems of equations

$$TX = \alpha B$$

$$T^T X = \alpha B$$

$$XT = \alpha B$$

$$XT^T = \alpha B$$

where α and β are scalars, A, B and C are rectangular matrices (sometimes symmetric or Hermitian), and T is an upper- or lower-triangular matrix.

Where appropriate, these operations are applied to different types of matrices. For each of the Level 3 subprograms, the matrices involved in the operations have one of the following characteristics:

- All matrices are general rectangular.
- Only one of the matrices is symmetric.
- Only one of the matrices is Hermitian.
- Only one of the matrices is triangular.

9.1.2 Matrix Storage

For the Level 3 BLAS subroutines, all matrices are stored in a two-dimensional array.

There is no provision for packed storage of symmetric, Hermitian, or triangular matrices. Only the upper triangle or the lower triangle is stored. Since the imaginary parts of the diagonal elements of a Hermitian matrix are zero, you do not have to set the imaginary parts of the corresponding Fortran array. They are assumed to be zero.

9.1.3 Naming Conventions

Each Level 3 BLAS subroutine has a name consisting of five or six characters. The first character of the name denotes the Fortran data type of the matrix. The second and third characters denote the type of matrices involved in the operation. The fourth, fifth, and sixth characters denote types of operations.

Table 9–1 shows the characters used in the Level 3 BLAS subroutine names and what the characters mean.

Table 9–1 Naming Conventions: Level 3 BLAS Subprograms

Character	Mnemonic	Meaning
First character	S	Single-precision real data
	D	Double-precision real data
	C	Single-precision complex data
	Z	Double-precision complex data
Second and third characters	GE	General matrices
	HE	One Hermitian matrix
	SY	One symmetric matrix
	TR	One triangular matrix
Fourth, fifth, and sixth characters	MM	Matrix-matrix product
	MA	Matrix addition
	MS	Matrix subtraction
	MT	Matrix transposition
	RK	Rank-k update
	R2K	Rank-2k update
	SM	Solution of a system of linear equations

For example, the name SGEMM is the subroutine for performing matrix-matrix multiplication (and addition if desired), where the matrices are general matrices with single-precision real elements.

9.2 Summary of Level 3 BLAS Subprograms

Table 9–2 summarizes the Level 3 BLAS subroutines provided by DXML. The rank-k updates of general matrices are provided by the _GEMM subroutines.

Table 9–2 Summary of Level 3 BLAS Subprograms

Routine Name	Operation
SGEMA	Calculates, in single-precision arithmetic, the sum of two real general matrices or their transposes.

(continued on next page)

Table 9–2 (Cont.) Summary of Level 3 BLAS Subprograms

Routine Name	Operation
DGEMA	Calculates, in double-precision arithmetic, the sum of two real general matrices or their transposes.
CGEMA	Calculates, in single-precision arithmetic, the sum of two complex general matrices, their transposes, their conjugates, or their conjugate transposes.
ZGEMA	Calculates, in double-precision arithmetic, the sum of two complex general matrices, their transposes, their conjugates, or their conjugate transposes.
SGEMM	Calculates, in single-precision arithmetic, a matrix-matrix product and addition for real general matrices or their transposes.
DGEMM	Calculates, in double-precision arithmetic, a matrix-matrix product and addition for real general matrices or their transposes.
CGEMM	Calculates, in single-precision arithmetic, a matrix-matrix product and addition for complex general matrices, their transposes, their conjugates, or their conjugate transposes.
ZGEMM	Calculates, in double-precision arithmetic, a matrix-matrix product and addition for complex general matrices, their transposes, their conjugates, or their conjugate transposes.
SGEMS	Calculates, in single-precision arithmetic, the difference of two real general matrices or their transposes.
DGEMS	Calculates, in double-precision arithmetic, the difference of two real general matrices or their transposes.
CGEMS	Calculates, in single-precision arithmetic, the difference of two complex general matrices, their transposes, their conjugates, or their conjugate transposes.
ZGEMS	Calculates, in double-precision arithmetic, the difference of two complex general matrices, their transposes, their conjugates, or their conjugate transposes.
SGEMT	Copies a single-precision, real general matrix or its transpose.
DGEMT	Copies a double-precision, real general matrix or its transpose.
CGEMT	Copies a single-precision, complex general matrix, its transpose, its conjugate, or its conjugate transpose.
ZGEMT	Copies a double-precision, complex general matrix, its transpose, its conjugate, or its conjugate transpose.
SSYMM	Calculates, in single-precision arithmetic, a matrix-matrix product and addition where a matrix multiplier is a real symmetric matrix.
DSYMM	Calculates, in double-precision arithmetic, a matrix-matrix product and addition where a matrix multiplier is a real symmetric matrix.
CSYMM	Calculates, in single-precision arithmetic, a matrix-matrix product and addition where a matrix multiplier is a complex symmetric matrix.

(continued on next page)

Table 9–2 (Cont.) Summary of Level 3 BLAS Subprograms

Routine Name	Operation
ZSYMM	Calculates, in double-precision arithmetic, a matrix-matrix product and addition where a matrix multiplier is a complex symmetric matrix.
CHEMM	Calculates, in single-precision arithmetic, a matrix-matrix product and addition where a matrix multiplier is a complex Hermitian matrix.
ZHEMM	Calculates, in double-precision arithmetic, a matrix-matrix product and addition where a matrix multiplier is a complex Hermitian matrix.
SSYRK	Calculates, in single-precision arithmetic, the rank-k update of a real symmetric matrix.
DSYRK	Calculates, in double-precision arithmetic, the rank-k update of a real symmetric matrix.
CSYRK	Calculates, in single-precision arithmetic, the rank-k update of a complex symmetric matrix.
ZSYRK	Calculates, in double-precision arithmetic, the rank-k update of a complex symmetric matrix.
CHERK	Calculates, in single-precision arithmetic, the rank-k update of a complex Hermitian matrix.
ZHERK	Calculates, in double-precision arithmetic, the rank-k update of a complex Hermitian matrix.
SSYR2K	Calculates, in single-precision arithmetic, the rank-2k update of a real symmetric matrix.
DSYR2K	Calculates, in double-precision arithmetic, the rank-2k update of a real symmetric matrix.
CSYR2K	Calculates, in single-precision arithmetic, the rank-2k update of a complex symmetric matrix.
ZSYR2K	Calculates, in double-precision arithmetic, the rank-2k update of a complex symmetric matrix.
CHER2K	Calculates, in single-precision arithmetic, the rank-2k update of a complex Hermitian matrix.
ZHER2K	Calculates, in double-precision arithmetic, the rank-2k update of a complex Hermitian matrix.
STRMM	Calculates, in single-precision arithmetic, a matrix-matrix product for a real triangular matrix or its transpose.
DTRMM	Calculates, in double-precision arithmetic, a matrix-matrix product for a real triangular matrix or its transpose.
CTRMM	Calculates, in single-precision arithmetic, a matrix-matrix product for a complex triangular matrix, its transpose, or its conjugate transpose.
ZTRMM	Calculates, in double-precision arithmetic, a matrix-matrix product for a complex triangular matrix, its transpose, or its conjugate transpose.

(continued on next page)

Table 9–2 (Cont.) Summary of Level 3 BLAS Subprograms

Routine Name	Operation
STRSM	Solves, in single-precision arithmetic, a triangular system of equations where the coefficient matrix is a real triangular matrix.
DTRSM	Solves, in double-precision arithmetic, a triangular system of equations where the coefficient matrix is a real triangular matrix.
CTRSM	Solves, in single-precision arithmetic, a triangular system of equations where the coefficient matrix is a complex triangular matrix.
ZTRSM	Solves, in double-precision arithmetic, a triangular system of equations where the coefficient matrix is a complex triangular matrix.

9.3 Calling the Subprograms

Each of the BLAS Level 3 subprograms returns a matrix. All the subprograms are subroutines. They are called as subroutines with a CALL statement. The example at the end of each subroutine reference description shows the subroutine call.

9.4 Argument Conventions

Each Level 3 BLAS subroutine has arguments that specify the nature and requirements of the subroutine. There are no optional arguments.

The arguments are ordered, but not every argument category is needed in each of the subroutines.

- Arguments specifying matrix options
- Arguments defining the size of the matrices
- Argument specifying the input scalar
- Arguments describing the input matrices
- Argument specifying the input scalar associated with the input-output matrix
- Arguments describing the input-output matrix

9.4.1 Specifying Matrix Options

The arguments that specify matrix options are character arguments:

- **side**
- **trans**
- **transa**
- **transb**
- **uplo**
- **diag**

In Fortran, a character argument can be longer than its corresponding dummy argument. For example, the value 'T' for **trans** can be passed as 'TRANSPOSE'.

side

The argument **side** is used in some subroutines to specify whether the matrix multiplier is on the left or the right. Table 9–3 shows the meaning of the values for the argument **side**.

Table 9–3 Values for the Argument SIDE

Value of side	Meaning
'L' or 'l'	Multiply the general matrix by the symmetric or triangular matrix on the left
'R' or 'r'	Multiply the general matrix by the symmetric or triangular matrix on the right

trans, transa, transb

Arguments **transa** and **transb** define the form of the input matrices to use in an operation. You do not change the form of an input matrix in your application program. DXML selects the proper elements, depending on the value of the **transa** and **transb** arguments.

For example, if A is the input matrix, and you want to use A in the operation, set the **transa** argument to 'N'. If you want to use A^T in the operation, set the **trans** argument to 'T'. The subroutine makes the changes and selects the proper elements from the matrix, so that A^T is used. Table 9–4 shows the meaning of the values for the arguments **transa** and **transb**.

Table 9–4 Values for the Arguments transa and transb

Value of transa and transb	Meaning for transa	Meaning for transb
'N' or 'n'	Operate with the matrix A.	Operate with the matrix B.
'T' or 't'	Operate with the transpose of matrix A.	Operate with the transpose of matrix B.
'R' or 'r'	Operate with the conjugate of matrix A.	Operate with the conjugate of matrix B.
'C' or 'c'	Operate with the conjugate transpose of matrix A.	Operate with the conjugate transpose of matrix B.

When an operation is performed with real matrices, the values 'T', 't', 'C', and 'c' have the same meaning, and the values 'N', 'n', 'R', and 'r' have the same meaning.

uplo

The Hermitian, symmetric, and triangular matrix subroutines (HE, SY, and TR subroutines) use the argument **uplo** to specify either the upper or lower triangle. Because of the structure of these matrices, the subroutines do not refer to all of the matrix values. Table 9–5 shows the meaning of the values for the argument **uplo**.

Table 9–5 Values for the Argument uplo

Value of uplo	Meaning
'U' or 'u'	Refers to the upper triangle
'L' or 'l'	Refers to the lower triangle

diag

The triangular matrix (TR) subroutines use the argument **diag** to specify whether or not the triangular matrix is unit-triangular. Table 9–6 shows the meaning of the values for the argument **diag**.

Table 9–6 Values for the Argument diag

Value of diag	Meaning
'U' or 'u'	Unit-triangular
'N' or 'n'	Not unit-triangular

When **diag** is specified as 'U' or 'u', the diagonal elements are not referenced by the subroutine. These elements are assumed to be unity.

9.4.2 Defining the Size of the Matrices

The sizes of the matrices are defined by the arguments **m**, **n**, and **k**. These arguments specify the number of rows or columns, m, n, or k of particular matrices.

You can call a subroutine with arguments **m** or **n** equal to 0 but the subroutine exits immediately without referencing its other arguments. For the _GEMM, _SYRK, and _HERK subroutines, if **k** = 0, the operations are reduced to $C \leftarrow \beta C$.

9.4.3 Describing the Matrices

In addition to their size, the description of each matrix is given by two arguments:

- Arguments that specify the array that stores the matrix: **a**, **b**, and **c** specify the arrays A, B, and C that store the matrices A, B, and C.
- Arguments that specify the leading dimension of each array: **lda**, **ldb**, and **ldc** specify the leading dimension of the arrays A, B, and C.

9.4.4 Specifying the Input Scalar

The input scalars α and β are always specified by the dummy argument names **alpha** and **beta**.

The input scalar associated with the input-output matrix is normally β and is specified by the dummy argument name **beta**.

If you supply an input scalar **beta** of zero, you do not need to set the array C. This means that an operation such as $C \leftarrow \alpha AB$ can be performed without having to set C to zero in the calling program.

9.4.5 Invalid Arguments

The following values of Level 3 subroutine arguments are invalid:

- Any value of the arguments **side**, **trans**, **transa**, **transb**, **uplo**, or **diag** that is not specified for the routine
- **m** < 0
- **n** < 0
- **k** < 0
- **lda** < the number of rows in the matrix A
- **ldb** < the number of rows in the matrix B
- **ldc** < the number of rows in the matrix C

9.5 Error Handling

The BLAS Level 3 subroutines do provide a check of the input arguments. If you call a Level 3 subroutine with an invalid value for any of its arguments, DXML reports the message and terminates execution of the program.

The code for BLAS Level 3 subroutines has calls to an input argument error handler, the XERBLA routine. When a subroutine detects an error, it passes the name of the subroutine and the number of the first argument that is in error to the XERBLA routine. DXML directs this information to the device or file defined as *stdout*.

9.6 A Look at a Level 3 BLAS Subroutine

The _TRMM subroutines compute a matrix-matrix product for either a triangular matrix, its transpose, or its conjugate transpose:

$$B \leftarrow \alpha AB \qquad B \leftarrow \alpha A^T B \qquad B \leftarrow \alpha A^H B$$

$$B \leftarrow \alpha BA \qquad B \leftarrow \alpha BA^T \qquad B \leftarrow \alpha BA^H$$

The triangular matrix multiply subroutines DTRMM (REAL*8 matrices) and ZTRMM (COMPLEX*16 matrices) have the following call format:

```
CALL DTRMM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
CALL ZTRMM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
```

The matrix B is an arbitrary rectangular matrix of size m by n, embedded in a possibly larger ldb by q matrix. The matrix A is a triangular matrix embedded in a larger matrix of size lda by p. The triangular part of A is n by n or m by m, depending on whether A multiplies B on the left side or the right by setting the argument **side**.

The lower or upper triangle of A is accessed by setting the argument **uplo**. The other triangle of A is ignored. The diagonal of A can be used, or it can be assumed to be unity by setting the argument **diag**. Either A, A^T, or A^H can be used by setting the argument **transa**. The scalar α has the same data type as the matrices A and B.

For the following real examples, assume A is a 7 by 7 matrix and B is a 6 by 5 matrix: $lda = 7$, $p = 7$, $ldb = 6$, and $q = 5$:

$$A = \begin{bmatrix} -10.00 & -7.00 & -6.00 & 3.00 & 6.00 & -9.00 & 8.00 \\ -4.00 & 5.00 & -6.00 & 2.00 & 3.00 & 5.00 & 1.00 \\ -1.00 & -5.00 & -4.00 & -5.00 & -8.00 & -9.00 & 8.00 \\ -5.00 & 1.00 & -7.00 & -1.00 & 3.00 & 1.00 & -8.00 \\ 6.00 & -2.00 & -6.00 & -4.00 & 9.00 & 2.00 & 3.00 \\ 10.00 & 0.00 & -7.00 & -4.00 & -1.00 & -10.00 & 3.00 \\ 1.00 & 10.00 & 6.00 & 2.00 & -4.00 & -3.00 & 2.00 \end{bmatrix}$$

For matrix B, only the leading 4 by 3 rectangle of B will be used: $m = 4$ and $n = 3$:

$$B = \begin{bmatrix} 4.00 & 9.00 & 2.00 & 5.00 & 0.00 \\ -8.00 & 3.00 & 9.00 & 3.00 & 4.00 \\ 4.00 & -8.00 & 0.00 & 4.00 & -9.00 \\ 7.00 & 4.00 & 8.00 & -2.00 & 0.00 \\ 2.00 & 10.00 & -7.00 & 4.00 & 5.00 \\ 2.00 & 4.00 & -7.00 & 5.00 & -3.00 \end{bmatrix}$$

Example 1: REAL*8 Matrices

$$B \leftarrow \alpha AB \text{ with } \alpha = 1.0$$

We will use the upper 4 by 4 triangle of A and the original diagonal. The matrix A essentially becomes:

$$\begin{bmatrix} -10.00 & -7.00 & -6.00 & 3.00 \\ 0.00 & 5.00 & -6.00 & 2.00 \\ 0.00 & 0.00 & -4.00 & -5.00 \\ 0.00 & 0.00 & 0.00 & -1.00 \end{bmatrix}$$

The call is shown in the following code:

```
SIDE = 'L'
UPLO = 'U'
TRANSA = 'N'
DIAG = 'N'
CALL DTRMM(SIDE, UPLO, TRANSA, DIAG, 4, 3, 1.0D0, A, LDA, B, LDB)
```

The product matrix B is as follows:

$$\begin{bmatrix} 13.00 & -51.00 & -59.00 \\ -50.00 & 71.00 & 61.00 \\ -51.00 & 12.00 & -40.00 \\ -7.00 & -4.00 & -8.00 \end{bmatrix}$$

Example 2: REAL*8 Matrices

$$B \leftarrow \alpha A^T B \text{ with } \alpha = -2.0$$

We will use the lower 4 by 4 triangle of A, the original diagonal, and take the transpose. On entry, the matrix A looks like the following:

$$\begin{bmatrix} -10.00 & 0.00 & 0.00 & 0.00 \\ -4.00 & 5.00 & 0.00 & 0.00 \\ -1.00 & -5.00 & -4.00 & 0.00 \\ -5.00 & 1.00 & -7.00 & -1.00 \end{bmatrix}$$

But after the transpose, A becomes:

$$\begin{bmatrix} -10.00 & -4.00 & -1.00 & -5.00 \\ 0.00 & 5.00 & -5.00 & 1.00 \\ 0.00 & 0.00 & -4.00 & -7.00 \\ 0.00 & 0.00 & 0.00 & -1.00 \end{bmatrix}$$

The call is shown in the following code:

```
SIDE = 'L'
UPLO = 'L'
TRANSA = 'T'
DIAG = 'N'
CALL DTRMM(SIDE, UPLO, TRANSA, DIAG, 4, 3, -2.0D0, A, LDA, B, LDB)
```

The product matrix B is as follows:

$$\begin{bmatrix} 94.00 & 228.00 & 192.00 \\ 106.00 & -118.00 & -106.00 \\ 130.00 & -8.00 & 112.00 \\ 14.00 & 8.00 & 16.00 \end{bmatrix}$$

Example 3: REAL*8 Matrices

$$B \leftarrow \alpha BA \text{ with } \alpha = 3.0$$

Matrix A multiplies matrix B on the right, so that A must be 3 by 3. We take the lower triangle of A, and we also assume A has unit diagonal. On entry, A is treated as the following:

$$\begin{bmatrix} 1.00 & 0.00 & 0.00 \\ -4.00 & 1.00 & 0.00 \\ -1.00 & -5.00 & 1.00 \end{bmatrix}$$

The call is shown in the following code:

```
SIDE = 'R'
UPLO = 'L'
TRANSA = 'N'
DIAG = 'U'
CALL DTRMM(SIDE, UPLO, TRANSA, DIAG, 4, 3, 3.0D0, A, LDA, B, LDB)
```

The product matrix B is as follows:

$$\begin{bmatrix} -102.00 & -3.00 & 6.00 \\ -87.00 & -126.00 & 27.00 \\ 108.00 & -24.00 & 0.00 \\ -51.00 & -108.00 & 24.00 \end{bmatrix}$$

Example 4: COMPLEX*16 Matrices

$$B \leftarrow \alpha B A^H \quad \text{with } \alpha = (1.0, -2.0).$$

On entry, A is the following 3 by 3 complex matrix:

$$\begin{bmatrix} (-7.0,-6.0) & (2.0,6.0) & (-4.0,9.0) \\ (-5.0,4.0) & (-6.0,-4.0) & (-10.0,-6.0) \\ (6.0,8.0) & (-3.0,8.0) & (1.0,-8.0) \end{bmatrix}$$

B is the following 4 by 3 complex matrix:

$$\begin{bmatrix} (8.0,-2.0) & (8.0,-9.0) & (10.0,-8.0) \\ (-3.0,8.0) & (3.0,-5.0) & (3.0,4.0) \\ (-2.0,-6.0) & (-6.0,-2.0) & (6.0,2.0) \\ (0.0,10.0) & (10.0,0.0) & (-6.0,-5.0) \end{bmatrix}$$

We will use the upper triangle of A and the original diagonal. The triangular A effectively looks like the following:

$$\begin{bmatrix} (-7.0,-6.0) & (2.0,6.0) & (-4.0,9.0) \\ (0.0,0.0) & (-6.0,-4.0) & (-10.0,-6.0) \\ (0.0,0.0) & (0.0,0.0) & (1.0,-8.0) \end{bmatrix}$$

After the conjugate transpose, A becomes the following:

$$\begin{bmatrix} (-7.0,6.0) & (0.0,0.0) & (0.0,0.0) \\ (2.0,-6.0) & (-6.0,-4.0) & (0.0,0.0) \\ (-4.0,-9.0) & (-10.0,6.0) & (1.0,8.0) \end{bmatrix}$$

The call is shown in the following code:

```
SIDE = 'R'
UPLO = 'U'
TRANSA = 'C'
DIAG = 'N'
ALPHA = (1.0D0, -2.0D0)
CALL ZTRMM(SIDE, UPLO, TRANSA, DIAG, 4, 3, ALPHA, A, LDA, B, LDB)
```

The product matrix B is as follows:

$$\begin{bmatrix} (-318.0,326.0) & (388.0,354.0) & (218.0,-76.0) \\ (-317.0,-91.0) & (-12.0,124.0) & (27.0,86.0) \\ (20.0,-40.0) & (-20.0,60.0) & (90.0,70.0) \\ (-173.0,66.0) & (138.0,-6.0) & (-72.0,-121.0) \end{bmatrix}$$

Level 3 BLAS Subroutines

This section provides descriptions of the Level 3 BLAS subroutines for real and complex operations.

For real operations, $\overline{A} = A$ and $A^H = A^T$.

SGEMA DGEMA CGEMA ZGEMA
Matrix-Matrix Addition

Format

{S,D,C,Z}GEMA (transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc)

Arguments

transa
character*1
On entry, specifies the form of $\mathrm{op}(A)$ as follows:

If **transa** = 'N' or 'n', $\mathrm{op}(A) = A$
If **transa** = 'T' or 't', $\mathrm{op}(A) = A^T$
If **transa** = 'R' or 'r', $\mathrm{op}(A) = \overline{A}$
If **transa** = 'C' or 'c', $\mathrm{op}(A) = A^H$

On exit, **transa** is unchanged.

transb
character*1
On entry, specifies the form of $\mathrm{op}(B)$ as follows:

If **transb** = 'N' or 'n', $\mathrm{op}(B) = B$
If **transb** = 'T' or 't', $\mathrm{op}(B) = B^T$
If **transb** = 'R' or 'r', $\mathrm{op}(B) = \overline{B}$
If **transb** = 'C' or 'c', $\mathrm{op}(B) = B^H$

On exit, **transb** is unchanged.

m
integer*4
On entry, the number of rows in the matrices $\mathrm{op}(A)$, $\mathrm{op}(B)$, and C; $m \geq 0$.
On exit, **m** is unchanged.

n
integer*4
On entry, the number of columns in the matrices $\mathrm{op}(A)$, $\mathrm{op}(B)$, and C; $n \geq 0$.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar α.
On exit, **alpha** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array A with dimensions lda by k.
For $\mathrm{op}(A) = A$ or $\overline{A}$, $k = n$ and the leading m by n part of array A contains the matrix A.
For $\mathrm{op}(A) = A^T$ or A^H, $k = m$ and the leading n by m part of array A contains the matrix A.
On exit, **a** is unchanged.

lda
integer*4
On entry, specifies the first dimension of array A.
For $\mathrm{op}(A) = A$ or $\overline{A}$, $lda \geq \max(1, m)$.
For $\mathrm{op}(A) = A^T$ or A^H, $lda \geq \max(1, n)$.
On exit, **lda** is unchanged.

beta
real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar β.
On exit, **beta** is unchanged.

b
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with dimensions ldb by k.
For $\mathrm{op}(B) = B$ or $\overline{B}$, $k = n$ and the leading m by n part of array B contains the matrix B.
For $\mathrm{op}(B) = B^T$ or B^H, $k = m$ and the leading n by m part of array B contains the matrix B.

ldb
integer*4
On entry, specifies the first dimension of array B.
For $\mathrm{op}(B) = B$ or $\overline{B}$, $ldb \geq \max(1, m)$.
For $\mathrm{op}(B) = B^T$ or B^H, $ldb \geq \max(1, n)$.
On exit, **ldb** is unchanged.

c
real*4 | real*8 | complex*8 | complex*16
On entry, an array with the dimension ldc by n.
On exit, the leading m by n part of array C is overwritten by the matrix $\alpha * \mathrm{op}(A) + \beta * \mathrm{op}(B)$.

ldc
integer*4
On entry, specifies the first dimension of array C; $ldc \geq \max(1, m)$.
On exit, **ldc** is unchanged.

Description

The _GEMA routines perform the following operations:

$$C \leftarrow \alpha * op(A) + \beta * op(B)$$

where $\mathrm{op}(X) = X, X^T, \overline{X}$, or X^H, α and β are scalars, and A, B, and C are matrices. $\mathrm{op}(A)$, $\mathrm{op}(B)$, and C are m by n matrices.

These subroutines can also perform the following operation when lda = ldc, and **transa** = 'N' or 'n', that is, when $\mathrm{op}(A) = A$:

$$A \leftarrow \alpha * A + \beta * \mathrm{op}(B)$$

where $\mathrm{op}(X) = X, X^T, \overline{X}$, or X^H, α and β are scalars, and A and B are m by n matrices.

SGEMM DGEMM CGEMM ZGEMM
Matrix-Matrix Product and Addition (Serial and Parallel Versions)

Format

{S,D,C,Z}GEMM (transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

Arguments

transa
character*1
On entry, specifies the form of $\mathrm{op}(A)$ used in the matrix multiplication:

If **transa** = 'N' or 'n', $\mathrm{op}(A) = A$
If **transa** = 'T' or 't', $\mathrm{op}(A) = A^T$
If **transa** = 'R' or 'r', $\mathrm{op}(A) = \overline{A}$
If **transa** = 'C' or 'c', $\mathrm{op}(A) = A^H$

On exit, **transa** is unchanged.

transb
character*1
On entry, specifies the form of $\mathrm{op}(B)$ used in the matrix multiplication:

If **transb** = 'N' or 'n', $\mathrm{op}(B) = B$
If **transb** = 'T' or 't', $\mathrm{op}(B) = B^T$
If **transb** = 'R' or 'r', $\mathrm{op}(B) = \overline{B}$
If **transb** = 'C' or 'c', $\mathrm{op}(B) = B^H$

m
integer*4
On entry, the number of rows of the matrix $\mathrm{op}(A)$ and of the matrix C; $m \geq 0$
On exit, **m** is unchanged.

n
integer*4
On entry, the number of columns of the matrix $\mathrm{op}(B)$ and of the matrix C; $n \geq 0$
On exit, **n** is unchanged.

k
integer*4
On entry, the number of columns of the matrix $\mathrm{op}(A)$ and the number of rows of the matrix $\mathrm{op}(B)$; $k \geq 0$
On exit, **k** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar α.
On exit, **alpha** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array A with dimensions lda by ka.
For $\mathrm{op}(A) = A$ or $\overline{A}$, $ka \geq k$ and the leading m by k portion of the array A contains the matrix A.

For $\mathrm{op}(A) = A^T$ or A^H, $ka \geq m$ and the leading k by m part of the array A contains the matrix A.
On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of array A.
For $\mathrm{op}(A) = A$ or $\overline{A}$, $lda \geq \max(1, m)$.
For $\mathrm{op}(A) = A^T$ or A^H, $lda \geq \max(1, k)$.
On exit, **lda** is unchanged.

b
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array B with dimensions ldb by kb.
For $\mathrm{op}(B)$ = B or $\overline{B}$, $kb \geq n$ and the leading k by n portion of the array contains the matrix B.
For $\mathrm{op}(B)$ = $(B)^T$ or B^H, $kb \geq k$ and the leading n by k part of the array contains the matrix B.
On exit, **b** is unchanged.

ldb
integer*4
On entry, the first dimension of array B.
For $\mathrm{op}(B)$ = B or $\overline{B}$, $ldb \geq \max(1, k)$.
For $\mathrm{op}(B)$ = B^T or B^H, $ldb \geq \max(1, n)$.
On exit, **ldb** is unchanged.

beta
real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar β.
On exit, **beta** is unchanged.

c
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with the dimension ldc by at least n.
On exit, the leading m by n part of array C is overwritten by the matrix $\alpha\mathrm{op}(A)\mathrm{op}(B) + \beta C$.

ldc
integer*4
On entry, the first dimension of array C; $ldc \geq \max(1, m)$
On exit, **ldc** is unchanged.

Description

The _GEMM routines perform the following operations:

$$C \leftarrow \alpha\mathrm{op}(A)\mathrm{op}(B) + \beta C$$

where $\mathrm{op}(X) = X, X^T, \overline{X}$, or X^H, α and β are scalars, and A, B, and C are matrices. $\mathrm{op}(A)$ is an m by k matrix, $\mathrm{op}(B)$ is a k by n matrix, and C is an m by n matrix.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

Example

```
REAL*4 A(20,40), B(20,30), C(40,30), ALPHA, BETA
M = 10
N = 20
K = 15
LDA = 20
LDB = 20
LDC = 40
ALPHA = 2.0
BETA = 2.0
CALL SGEMM ('T','N',M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
```

This Fortran code computes the product $C \leftarrow \alpha A^T B + \beta C$ where A is a real general matrix. A is a 15 by 10 real general matrix embedded in array A. B is a 15 by 20 real general matrix embedded in array B. C is a 10 by 20 real general matrix embedded in array C.

SGEMS DGEMS CGEMS ZGEMS
Matrix-Matrix Subtraction

Format

{S,D,C,Z}GEMS (transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc)

Arguments

transa
character*1
On entry, specifies the form of $op(A)$ as follows:

If **transa** = 'N' or 'n', $\mathrm{op}(A) = A$
If **transa** = 'T' or 't', $\mathrm{op}(A) = A^T$
If **transa** = 'R' or 'r', $\mathrm{op}(A) = \overline{A}$
If **transa** = 'C' or 'c', $\mathrm{op}(A) = A^H$

On exit, **transa** is unchanged.

transb
character*1
On entry, specifies the form of $\mathrm{op}(B)$ as follows:

If **transb** = 'N' or 'n', $\mathrm{op}(B) = B$
If **transb** = 'T' or 't', $\mathrm{op}(B) = B^T$
If **transb** = 'R' or 'r', $\mathrm{op}(B) = \overline{B}$
If **transb** = 'C' or 'c', $\mathrm{op}(B) = B^H$

On exit, **transb** is unchanged.

m
integer*4
On entry, the number of rows in the matrices $\mathrm{op}(A)$, $\mathrm{op}(B)$, and C; $m \geq 0$.
On exit, **m** is unchanged.

n
integer*4
On entry, the number of columns in the matrices $\mathrm{op}(A)$, $\mathrm{op}(B)$, and C; $m \geq 0$.
On exit, **n** is unchanged.

alpha
Input real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar α.
On exit, **alpha** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array A with dimensions lda by k.
For $\mathrm{op}(A) = A$ or $\overline{A}$, $k = n$ and the leading m by n part of array A contains the matrix A.
For $\mathrm{op}(A) = A^T$ or A^H, $k = m$ and the leading n by m part of array A contains the matrix A.
On exit, **a** is unchanged.

lda
integer*4
On entry, specifies the first dimension of array A.
For $\mathrm{op}(A) = A$ or $\overline{A}$, $lda \geq \max(1, m)$.
For $\mathrm{op}(A) = A^T$ or A^H, $lda \geq \max(1, n)$.
On exit, **lda** is unchanged.

beta
real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar β.
On exit, **beta** is unchanged.

b
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array A with dimensions ldb by k.
For $\mathrm{op}(B) = B$ or $\overline{B}$, $k = n$ and the leading m by n part of array B contains the matrix B.
For $\mathrm{op}(B) = B^T$ or B^H, $k = m$ and the leading n by m part of array B contains the matrix B.

ldb
integer*4
On entry, specifies the first dimension of array B.
For $\mathrm{op}(B) = B$ or $\overline{B}$, $ldb \geq \max(1, m)$.
For $\mathrm{op}(B) = B^T$ or B^H, $ldb \geq \max(1, n)$.
On exit, **ldb** is unchanged.

c
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with the dimension ldc by at least n.
On exit, the leading m by n part of array C is overwritten by the matrix $\alpha * \mathrm{op}(A) - \beta * \mathrm{op}(B)$.

ldc
integer*4
On entry, specifies the first dimension of array C; $ldc \geq \max(1, m)$.
On exit, **ldc** is unchanged.

Description

The _GEMS routines perform one of the following matrix-matrix operations:

$$C \leftarrow \alpha * op(A) - \beta * op(B)$$

where $\mathrm{op}(X) = X, X^T, \overline{X}$, or $\overline{X}^T$, α and β are scalars, and A, B, and C are matrices. $\mathrm{op}(A)$, $\mathrm{op}(B)$, and C are m by n matrices.

These subroutines can also perform the following operation when lda = ldc, and **transa** = 'N' or 'n', that is, when $\mathrm{op}(A) = A$:

$$A \leftarrow \alpha * A - \beta * \mathrm{op}B$$

where $\mathrm{op}(X) = X, X^T, \overline{X}$, or X^H, α and β are scalars, and A and B are m by n matrices.

SGEMT DGEMT CGEMT ZGEMT
Matrix-Matrix Copy

Format

{S,D,C,Z}GEMT (trans, m, n, alpha, a, lda, b, ldb)

Arguments

trans
character*1
On entry, specifies the form of op(A) as follows:

When **trans** = 'N' or 'n', $\mathrm{op}(A) = A$
When **trans** = 'T' or 't', $\mathrm{op}(A) = A^T$
When **trans** = 'R' or 'r', $\mathrm{op}(A) = \overline{A}$
When **trans** = 'C' or 'c', $\mathrm{op}(A) = A^H$

On exit, **trans** is unchanged.

m
integer*4
On entry, the number of rows in the matrices $\mathrm{op}(A)$ and B; $m \geq 0$.
On exit, **m** is unchanged.

n
integer*4
On entry, the number of columns in the matrices $\mathrm{op}(A)$, and B; $n \geq 0$.
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar α.
On exit, **alpha** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array A with dimensions lda by k.
For $\mathrm{op}(A) = A$ or $\overline{A}$, $k = n$ and the leading m by n part of array A contains the matrix A.
For $\mathrm{op}(A) = A^T$ or A^H, $k = m$ and the leading n by m part of array A contains the matrix A.
On exit, **a** is unchanged.

lda
integer*4
On entry, specifies the first dimension of array A.
For $\mathrm{op}(A) = A$ or $\overline{A}$, $lda \geq \max(1, m)$.
For $\mathrm{op}(A) = A^T$ or A^H, $lda \geq \max(1, n)$.
On exit, **lda** is unchanged.

b
real*4 | real*8 | complex*8 | complex*16
On entry, an array with dimensions *ldb* by *n*.
On exit, the leading *m* by *n* part of the array B is overwritten by the matrix $\alpha * op(A)$.

ldb
integer*4
On entry, specifies the first dimension of array B; $ldb \geq \max(1, m)$.
On exit, **ldb** is unchanged.

Description

The _GEMT routines perform the following operation:

$$B \leftarrow \alpha * op(A)$$

$\mathrm{op}(X) = X, X^T, \overline{X},$ or $\overline{X}^T$, α is a scalar, and A and B are matrices. $\mathrm{op}(A)$ and B are m by n matrices.

These subroutines can also perform matrix scaling when *lda* = *ldb*, and **trans** = 'N', 'n', 'R', or 'r':

$$A \leftarrow \alpha * op(A)$$

where $\mathrm{op}(X) = X$ or $\overline{X}$, α is a scalar, and A and $\mathrm{op}(A)$ are m by n matrices.

An in place matrix transpose or conjugate transpose may be performed when *lda* = *ldb*, **trans** = 'T', 't', 'C', or 'c', and $m = n$:

$$A \leftarrow \alpha * op(A)$$

where $\mathrm{op}(X) = X^T$ or X^H, α is a scalar, and A and $\mathrm{op}(A)$ are m by n matrices.

SSYMM DSYMM CSYMM ZSYMM CHEMM ZHEMM
Matrix-Matrix Product and Addition for a Symmetric or Hermitian Matrix

Format

{S,D,C,Z}SYMM (side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)

{C,Z}HEMM (side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)

Arguments

side
character*1
On entry, specifies whether the symmetric matrix A multiplies B on the left side or the right side:

If **side** = 'L' or 'l', the operation is $C \leftarrow \alpha AB + \beta C$.
If **side** = 'R' or 'r', the operation is $C \leftarrow \alpha BA + \beta C$.

On exit, **side** is unchanged.

uplo
character*1
On entry, specifies whether the upper- or lower-triangular part of the symmetric matrix A is referenced:

If **uplo** = 'U' or 'u', the upper-triangular part of A is referenced.
If **uplo** = 'L' or 'l', the lower-triangular part of A is referenced.

On exit, **uplo** is unchanged.

m
integer*4
On entry, the number of rows of the matrix C; $m \geq 0$
On exit, **m** is unchanged.

n
integer*4
On entry, the number of columns of the matrix C; $n \geq 0$
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar α.
On exit, **alpha** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array A with dimensions lda by ka.
If the multiplication is on the left side, $ka \geq m$ and the leading m by m part of the array contains the matrix A.
If the multiplication is on the right side, $ka \geq n$ and the leading n by n part of the array A must contain the matrix A.
In either case, when the leading part of the array is specified as the upper part, the upper triangular part of array A contains the upper-triangular part of the matrix A, and the lower-triangular part of matrix A is not referenced. When

the lower part is specified, the lower triangular part of the array A contains the lower triangular part of the matrix A, and the upper-triangular part of A is not referenced.

In complex Hermitian matrices, the imaginary parts of the diagonal elements need not be initialized. They are assumed to be zero.

On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of array A. When multiplication is on the left, $lda \geq \max(1, m)$. When multiplication is on the right, $lda \geq \max(1, n)$.
On exit, **lda** is unchanged.

b
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array B of dimensions ldb by at least n. The leading m by n part of the array B must contain the matrix B.
On exit, **b** is unchanged.

ldb
integer*4
On entry, the first dimension of B; $ldb \geq \max(1, m)$
On exit, **ldb** is unchanged.

beta
real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar β.
On exit, **beta** is unchanged.

c
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array with the dimension ldc by at least n.
On exit, **c** is overwritten; the array C is overwritten by the m by n updated matrix.

ldc
integer*4
On entry, the first dimension of array C; $ldc \geq \max(1, n)$
On exit, **ldc** is unchanged.

Description

These routines compute a matrix-matrix product and addition for a real or complex symmetric matrix or a complex Hermitian matrix:

$$C \leftarrow \alpha AB + \beta C$$

$$C \leftarrow \alpha BA + \beta C$$

α and β are scalars, A is the symmetric or Hermitian matrix, and B and C are m by n matrices.

Example

```
REAL*4 A(20,20), B(30,40), C(30,50), ALPHA, BETA
M = 10
N = 20
LDA = 20
LDB = 30
LDC = 30
ALPHA = 2.0
BETA = 3.0
CALL SSYMM ('L','U',M,N,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
```

This Fortran code computes the product of a symmetric matrix and a rectangular matrix. The operation is $C \leftarrow \alpha AB + \beta C$ where A is a 10 by 10 real symmetric matrix embedded in array A, B is a 10 by 20 real matrix embedded in array B, and C is a 10 by 20 real matrix embedded in array C. The leading 10 by 10 upper-triangular part of the array A contains the upper-triangular part of the matrix A. The lower-triangular part of A is not referenced.

```
COMPLEX*16 A(30,40), B(15,20), C(19,13), ALPHA, BETA
M = 12
N = 7
LDA = 30
LDB = 15
LDC = 19
ALPHA = (2.0D0, 0.0D0)
BETA = (0.0D0, -2.0D0)
CALL ZHEMM ('R','L',M,N,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
```

This Fortran code computes the product of a Hermitian matrix and a rectangular matrix. The operation is $C \leftarrow \alpha BA + \beta C$ where A is a 7 by 7 complex Hermitian matrix embedded in array A, B is a 12 by 7 complex matrix embedded in array B, and C is a 12 by 7 complex matrix embedded in array C. The leading 7 by 7 lower-triangular part of the array A contains the lower-triangular part of the matrix A. The upper-triangular part of A is not referenced.

SSYRK DSYRK CSYRK ZSYRK
Rank-k Update of a Symmetric Matrix

Format

{S,D,C,Z}SYRK (uplo, trans, n, k, alpha, a, lda, beta, c, ldc)

Arguments

uplo
character*1
On entry, specifies whether the upper- or lower-triangular part of the symmetric matrix C is to be referenced:

If **uplo** = 'U' or 'u', the upper-triangular part of C is to be referenced.
If **uplo** = 'L' or 'l', the lower-triangular part of C is to be referenced.

On exit, **uplo** is unchanged.

trans
character*1
On entry, specifies the operation to be performed:

If **trans** = 'N' or 'n', $C \leftarrow \alpha AA^T + \beta C$
If **trans** = 'T' or 't', $C \leftarrow \alpha A^T A + \beta C$

On exit, **trans** is unchanged.

n
integer*4
On entry, specifies the order of the matrix C; $n \geq 0$
On exit, **n** is unchanged.

k
integer*4
On entry, the number of columns of the matrix A when **trans** = 'N' or 'n', or the number of rows of the matrix A when **trans** = 'T' or 't'; $k \geq 0$.
On exit, **k** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar α.
On exit, **alpha** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array A with dimensions lda by ka.
For **trans** = 'N' or 'n', $ka \geq k$ and the leading n by k portion of the array A contains the matrix A.
For **trans** = 'T' or 't', $ka \geq n$ and the leading k by n part of the array A contains the matrix A.
On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of array A.
For **trans** = 'N' or 'n', $lda \geq \max(1, n)$.

For **trans** = 'T', 't', 'C' or 'c', $lda \geq \max(1, k)$.
On exit, **lda** is unchanged.

beta
real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar β.
On exit, **beta** is unchanged.

c
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array C of dimensions ldc by at least n. If **uplo** specifies the upper part, the leading n by n upper-triangular part of the array C must contain the upper-triangular part of the symmetric matrix C, and the strictly lower-triangular part of C is not referenced.

If **uplo** specifies the lower part, the leading n by n lower-triangular part of the array C must contain the lower-triangular part of the symmetric matrix C, and the strictly upper-triangular part of C is not referenced.
On exit, **c** is overwritten; the triangular part of the array C is overwritten by the triangular part of the updated matrix.

ldc
integer*4
On entry, the first dimension of array C; $ldc \geq \max(1, n)$
On exit, **ldc** is unchanged.

Description

The _SYRK routines perform the rank-k update of a symmetric matrix:

$$C \leftarrow \alpha A A^T + \beta C$$

$$C \leftarrow \alpha A^T A + \beta C$$

α and β are scalars, C is an n by n symmetric matrix. In the first case, A is an n by k matrix, and in the second case, A is a k by n matrix.

Example

```
REAL*4 A(40,20), C(20,20), ALPHA, BETA
LDA = 40
LDC = 20
N = 10
K = 15
ALPHA = 1.0
BETA = 2.0
CALL SSYRK ('U','N',N,K,ALPHA,A,LDA,BETA,C,LDC)
```

This Fortran code computes the rank-k update of the real symmetric matrix C: $C \leftarrow \alpha A A^T + \beta C$. C is a 10 by 10 matrix, and A is a 10 by 15 matrix. Only the upper-triangular part of C is referenced. The leading 10 by 15 part of array A contains the matrix A. The leading 10 by 10 upper-triangular part of array C contains the upper-triangular matrix C.

CHERK, ZHERK
Rank-k Update of a Complex Hermitian Matrix

Format

{C,Z}HERK (uplo, trans, n, k, alpha, a, lda, beta, c, ldc)

Arguments

uplo
character*1
On entry, specifies whether the upper- or lower-triangular part of the Hermitian matrix C is to be referenced:

If **uplo** = 'U' or 'u', the upper-triangular part of C is to be referenced.
If **uplo** = 'L' or 'l', the lower-triangular part of C is to be referenced.

On exit, **uplo** is unchanged.

trans
character*1
On entry, specifies the operation to be performed:

If **trans** = 'N' or 'n', $C \leftarrow \alpha AA^H + \beta C$
If **trans** = 'C' or 'c', $C \leftarrow \alpha A^H A + \beta C$

On exit, **trans** is unchanged.

n
integer*4
On entry, the order of the matrix C; $n \geq 0$
On exit, **n** is unchanged.

k
integer*4
On entry, the number of columns of the matrix A when **trans** = 'N' or 'n', or the number of rows of the matrix A when **trans** = 'C' or 'c'; $k \geq 0$.
On exit, **k** is unchanged.

alpha
real*4 | real*8
On entry, specifies the scalar α.
On exit, **alpha** is unchanged.

a
complex*8 | complex*16
On entry, a two-dimensional array A with dimensions lda by ka.
For **trans** = 'N' or 'n', $ka \geq k$ and the leading n by k portion of the array A contains the matrix A.
For **trans** = 'T', 't', 'C', or 'c', $ka \geq n$ and the leading k by n part of the array A contains the matrix A.
On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of array A.
For **trans** = 'N' or 'n' $lda \geq \max(1, n)$.

For **trans** = 'C' or 'c', $lda \geq \max(1, k)$.
On exit, **lda** is unchanged.

beta
real*4 | real*8
On entry, the scalar β.
On exit, **beta** is unchanged.

c
complex*8 | complex*16
On entry, a two-dimensional array C of dimensions ldc by at least n.

If **uplo** specifies the upper part, the leading n by n upper-triangular part of the array C must contain the upper-triangular part of the Hermitian matrix C, and the strictly lower-triangular part of C is not referenced.

If **uplo** specifies the lower part, the leading n by n lower-triangular part of the array C must contain the lower-triangular part of the Hermitian matrix C, and the strictly upper-triangular part of C is not referenced.

The imaginary parts of the diagonal elements need not be initialized. They are assumed to be 0, and on exit, they are set to 0.
On exit, **c** is overwritten; the triangular part of the array C is overwritten by the triangular part of the updated matrix.

ldc
integer*4
On entry, the first dimension of array C; $ldc \geq \max(1, n)$
On exit, **ldc** is unchanged.

Description

CHERK and ZHERK perform the rank-k update of a complex Hermitian matrix:

$$C \leftarrow \alpha A A^H + \beta C$$

$$C \leftarrow \alpha A^H A + \beta C$$

α and β are real scalars, C is an n by n Hermitian matrix, and A is an n by k matrix in the first case and a k by n matrix in the second case.

Example

```
COMPLEX*8 A(40,20), C(20,20)
REAL*4 ALPHA, BETA
LDA = 40
LDC = 20
N = 10
K = 15
ALPHA = 1.0
BETA = 2.0
CALL CHERK ('U','N',N,K,ALPHA,A,LDA,BETA,C,LDC)
```

This Fortran code computes the rank-k update of the complex Hermitian matrix C: $C \leftarrow \alpha A A^H + \beta C$. C is a 10 by 10 matrix, and A is a 10 by 15 matrix. Only the upper-triangular part of C is referenced. The leading 10 by 15 part of array A contains the matrix A. The leading 10 by 10 upper-triangular part of array C contains the upper-triangular matrix C.

SSYR2K DSYR2K CSYR2K ZSYR2K
Rank-2k Update of a Symmetric Matrix

Format

{S,D,C,Z}SYR2K (uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

Arguments

uplo
character*1
On entry, specifies whether the upper- or lower-triangular part of the symmetric matrix C is to be referenced:

If **uplo** = 'U' or 'u', the upper-triangular part of C is to be referenced.
If **uplo** = 'L' or 'l', the lower-triangular part of C is to be referenced.

On exit, **uplo** is unchanged.

trans
character*1
On entry, specifies the operation to be performed:

If **trans** = 'N' or 'n', $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$
If **trans** = 'T' or 't', $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$

On exit, **trans** is unchanged.

n
integer*4
On entry, the order n of the matrix C; $n \geq 0$
On exit, **n** is unchanged.

k
integer*4
On entry, the number of columns of the matrices A and B when **trans** = 'N' or 'n', or the number of rows of the matrix A and B when **trans** = 'T' or 't': $k \geq 0$.
On exit, **k** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar α.
On exit, **alpha** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array A with dimensions lda by ka.
For **trans** = 'N' or 'n', $ka \geq k$ and the leading n by k portion of the array A contains the matrix A.
For **trans** = 'T' or 't', $ka \geq n$ and the leading k by n part of the array A contains the matrix A.
On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of array A.
For **trans** = 'N' or 'n' $lda \geq \max(1, n)$.
For **trans** = 'T' or 't', $lda \geq \max(1, k)$.
On exit, **lda** is unchanged.

b
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array B with dimensions ldb by kb.
For **trans** = 'N' or 'n', $kb \geq k$ and the leading n by k portion of the array B contains the matrix B.
For **trans** = 'T' or 't', $kb \geq n$ and the leading k by n part of the array B contains the matrix B.
On exit, **b** is unchanged.

ldb
integer*4
On entry, the first dimension of array B.
For **trans** = 'N' or 'n', $ldb \geq \max(1, n)$.
For **trans** = 'T' or 't', $ldb \geq \max(1, k)$.
On exit, **ldb** is unchanged.

beta
real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar β.
On exit, **beta** is unchanged.

c
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array C of dimensions ldc by at least n.

If **uplo** specifies the upper part, the leading n by n upper-triangular part of the array C must contain the upper-triangular part of the symmetric matrix C, and the strictly lower-triangular part of C is not referenced.

If **uplo** specifies the lower part, the leading n by n lower-triangular part of the array C must contain the lower-triangular part of the symmetric matrix C, and the strictly upper-triangular part of C is not referenced.
On exit, **c** is overwritten; the triangular part of the array C is overwritten by the triangular part of the updated matrix.

ldc
integer*4
On entry, the first dimension of array C; $ldc \geq \max(1, n)$
On exit, **ldc** is unchanged.

Description

The _SYR2K routines perform the rank-2k update of a symmetric matrix:

$$C \leftarrow \alpha A B^T + \alpha B A^T + \beta C$$

$$C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$$

α and β are scalars, C is an n by n symmetric matrix, and A and B are n by k matrices in the first case and k by n matrices in the second case.

Example

```
REAL*4 A(40,10), B(40,10), C(20,20), ALPHA, BETA
LDA = 40
LDB = 30
LDC = 20
N = 18
K = 10
ALPHA = 1.0
BETA = 2.0
CALL SSYR2K ('U','N',N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
```

This Fortran code computes the rank-2k update of the real symmetric matrix C: $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$. Only the upper-triangular part of C is referenced. The leading 18 by 10 part of array A contains the matrix A. The leading 18 by 10 part of array B contains the matrix B. The leading 18 by 18 upper-triangular part of array C contains the upper-triangular matrix C.

CHER2K, ZHER2K
Rank-2k Update of a Complex Hermitian Matrix

Format

{C,Z}HER2K (uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

Arguments

uplo
character*1
On entry, specifies whether the upper- or lower-triangular part of the Hermitian matrix C is to be referenced:

If **uplo** = 'U' or 'u', the upper-triangular part of C is to be referenced.
If **uplo** = 'L' or 'l', the lower-triangular part of C is to be referenced.

On exit, **uplo** is unchanged.

trans
character*1
On entry, specifies the operation to be performed:

If **trans** = 'N' or 'n', $C \leftarrow \alpha AB^H + \overline{\alpha}BA^H + \beta C$
If **trans** = 'C' or 'c', $C \leftarrow \alpha A^H B + \overline{\alpha}B^H A + \beta C$

On exit, **trans** is unchanged.

n
integer*4
On entry, the order of the matrix C; $n \geq 0$
On exit, **n** is unchanged.

k
integer*4
On entry, the number of columns of the matrices A and B when **trans** = 'N' or 'n', or the number of rows of the matrices A and B when **trans** = 'C' or 'c'; $k \geq 0$.
On exit, **k** is unchanged.

alpha
complex*8 | complex*16
On entry, specifies the scalar α.
On exit, **alpha** is unchanged.

a
complex*8 | complex*16
On entry, a two-dimensional array A with dimensions lda by ka.
For **trans** = 'N' or 'n', $ka \geq k$ and the leading n by k portion of the array A contains the matrix A.
For **trans** = 'C' or 'c', $ka \geq n$ and the leading k by n part of the array A contains the matrix A.
On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of array A.
For **trans** = 'N' or 'n', $lda \geq \max(1, n)$.
For **trans** = 'C' or 'c', $lda \geq \max(1, k)$.
On exit, **lda** is unchanged.

b
complex*8 | complex*16
On entry, a two-dimensional array B with dimensions ldb by kb.
For **trans** = 'N' or 'n', $kb \geq k$ and the leading n by k portion of the array B contains the matrix B.
For **trans** = 'C' or 'c', $kb \geq n$ and the leading k by n part of the array B contains the matrix B.
On exit, **b** is unchanged.

ldb
integer*4
For **trans** = 'N' or 'n' $ldb \geq \max(1, n)$.
For **trans** = 'C' or 'c', $ldb \geq \max(1, k)$.
On exit, **ldb** is unchanged.

beta
real*4 | real*8
On entry, specifies the scalar β.
On exit, **beta** is unchanged.

c
complex*8 | complex*16
On entry, a two-dimensional array C of dimensions ldc by at least n.

If **uplo** specifies the upper part, the leading n by n upper-triangular part of the array C must contain the upper-triangular part of the Hermitian matrix C, and the strictly lower-triangular part of C is not referenced.

If **uplo** specifies the lower part, the leading n by n lower-triangular part of the array C must contain the lower-triangular part of the Hermitian matrix C, and the strictly upper-triangular part of C is not referenced.

The imaginary parts of the diagonal elements need not be initialized. They are assumed to be 0, and on exit, they are set to 0.

On exit, **c** is overwritten; the triangular part of the array C is overwritten by the triangular part of the updated matrix.

ldc
integer*4
On entry, the first dimension of array C; $ldc \geq \max(1, n)$
On exit, **ldc** is unchanged.

Description

CHER2K and ZHER2K perform the rank-2k update of a complex Hermitian matrix:

$$C \leftarrow \alpha AB^H + \overline{\alpha}BA^H + \beta C$$

$$C \leftarrow \alpha A^H B + \overline{\alpha}B^H A + \beta C$$

where α is a complex scalar, β is a real scalar, and C is an n by n Hermitian matrix. In the first case, A and B are n by k matrices, and in the second case, they are k by n matrices.

Example

```
COMPLEX*8 A(40,10), B(40,10), C(20,20), ALPHA
REAL*4 BETA
LDA = 40
LDB = 30
LDC = 20
N = 18
K = 10
ALPHA = (1.0, 1.0)
BETA = 2.0
CALL CHER2K ('U','N',N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
```

This Fortran code computes the rank-2k update of the complex Hermitian matrix C: $C \leftarrow \alpha AB^H + \overline{\alpha}BA^H + \beta C$. Only the upper-triangular part of C is referenced. The leading 18 by 10 part of array A contains the matrix A. The leading 18 by 10 part of array B contains the matrix B. The leading 18 by 18 upper-triangular part of array C contains the upper-triangular matrix C.

STRMM DTRMM CTRMM ZTRMM
Matrix-Matrix Product for Triangular Matrix

Format

{S,D,C,Z}TRMM (side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)

Arguments

side
character*1
On entry, specifies whether $\mathrm{op}(A)$ multiplies B on the left or right in the operation:

If **side** = 'L' or 'l', the operation is $B \leftarrow \alpha * \mathrm{op}(A)B$.
If **side** = 'R' or 'r', the operation is $B \leftarrow \alpha * B\mathrm{op}(A)$.

On exit, **side** is unchanged.

uplo
character*1
On entry, specifies whether the matrix A is an upper- or lower-triangular matrix:

If **uplo** = 'U' or 'u', the matrix A is an upper-triangular matrix.
If **uplo** = 'L' or 'l', the matrix A is a lower-triangular matrix.

On exit, **uplo** is unchanged.

transa
character*1
On entry, specifies the form of $\mathrm{op}(A)$ used in the matrix multiplication:

If **transa** = 'N' or 'n', $\mathrm{op}(A) = A$.
If **transa** = 'T' or 't', $\mathrm{op}(A) = A^T$.
If **transa** = 'C' or 'c', $\mathrm{op}(A) = A^H$.

On exit, **transa** is unchanged.

diag
character*1
On entry, specifies whether the matrix A is unit-triangular:

If **diag** = 'U' or 'u', A is a unit-triangular matrix.
If **diag** = 'N' or 'n', A is not a unit-triangular matrix.

On exit, **diag** is unchanged.

m
integer*4
On entry, the number of rows of the matrix B; $m \geq 0$
On exit, **m** is unchanged.

n
integer*4
On entry, the number of columns of the matrix B; $n \geq 0$
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar α.
On exit, **alpha** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array A with dimensions lda by k.
If the multiplication is on the left side, $k \geq m$ and the leading m by m part of the array contains the matrix A.
If the multiplication is on the right side, $k \geq n$ and the leading n by n part of the array A must contain the matrix A.
In either case, when the leading part of the array is specified as the upper part, the upper triangular part of array A contains the upper-triangular part of the matrix A, and the lower-triangular part of matrix A is not referenced. When the lower part is specified, the lower triangular part of the array A contains the lower triangular part of the matrix A, and the upper-triangular part of A is not referenced.

If matrix A is unit-triangular, its diagonal elements are assumed to be unity and are not referenced.
On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of A. When multiplication is on the left, $lda \geq \max(1, m)$. When multiplication is on the right, $lda \geq \max(1, n)$.
On exit, **lda** is unchanged.

b
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array B of dimensions ldb by at least n. The leading m by n part of the array B must contain the matrix B.
On exit, **b** is overwritten by the m by n updated matrix.

ldb
integer*4
On entry, the first dimension of B; $ldb \geq \max(1, m)$
On exit, **ldb** is unchanged.

Description

STRMM and DTRMM compute a matrix-matrix product for a real triangular matrix or its transpose. CTRMM and ZTRMM compute a matrix-matrix product for a complex triangular matrix, its transpose, or its conjugate transpose.

$$B \leftarrow \alpha \mathrm{op}(A) B$$

$$B \leftarrow \alpha B (\mathrm{op}(A))$$

where $\mathrm{op}(A) = A, A^T,$ or A^H

α is a scalar, B is an m by n matrix, and A is a unit or non-unit, upper- or lower-triangular matrix.

Example

```
REAL*8 A(25,40), B(30,35), ALPHA
M = 15
N = 18
LDA = 25
LDB = 30
ALPHA = -1.0D0
CALL DTRMM ('R','L','T','U',M,N,ALPHA,A,LDA,B,LDB)
```

This Fortran code solves the system $B \leftarrow \alpha BA^T$ where A is a lower-triangular real matrix with a unit diagonal. A is an 18 by 18 real triangular matrix embedded in array A, and B is a 15 by 18 real rectangular matrix embedded in array B. The leading 18 by 18 lower-triangular part of the array A must contain the lower-triangular matrix A. The upper-triangular part of A and the diagonal are not referenced.

```
COMPLEX*16 A(25,40), B(30,35), ALPHA
M = 15
N = 18
LDA = 25
LDB = 30
ALPHA = (-1.0D0, 2.0D0)
CALL ZTRMM ('R','L','T','U',M,N,ALPHA,A,LDA,B,LDB)
```

This Fortran code solves the system $B \leftarrow \alpha BA^T$ where A is a lower-triangular complex matrix with a unit diagonal. A is an 18 by 18 complex triangular matrix embedded in array A, and B is a 15 by 18 complex rectangular matrix embedded in array B. The leading 18 by 18 lower-triangular part of the array A must contain the lower-triangular matrix A. The upper-triangular part of A and the diagonal are not referenced.

STRSM DTRSM CTRSM ZTRSM
Solve a Triangular System of Equations with a Triangular Coefficient Matrix

Format

{S,D,C,Z}TRSM (side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)

Arguments

side
character*1
On entry, specifies whether $\mathrm{op}(A)$ is on the left side or the right side of X in the system of equations:

If **side** = 'L' or 'l', the system is $\mathrm{op}(A)X = \alpha B$.
If **side** = 'R' or 'r', the system is $X\mathrm{op}(A) = \alpha B$.

On exit, **side** is unchanged.

uplo
character*1
On entry, specifies whether the matrix A is an upper- or lower-triangular matrix:

If **uplo** = 'U' or 'u', the matrix A is an upper-triangular matrix.
If **uplo** = 'L' or 'l', the matrix A is a lower-triangular matrix.

On exit, **uplo** is unchanged.

transa
character*1
On entry, specifies the form of $\mathrm{op}(A)$ used in the system of equations:

If **transa** = 'N' or 'n', $\mathrm{op}(A) = A$.
If **transa** = 'T' or 't', $\mathrm{op}(A) = A^T$.
If **transa** = 'C' or 'c', $\mathrm{op}(A) = A^H$.

On exit, **transa** is unchanged.

diag
character*1
On entry, specifies whether the matrix A is unit-triangular:

If **diag** = 'U' or 'u', A is a unit-triangular matrix.
If **diag** = 'N' or 'n', A is not a unit-triangular matrix.

On exit, **diag** is unchanged.

m
integer*4
On entry, the number of rows m of the matrix B; $m \geq 0$
On exit, **m** is unchanged.

n
integer*4
On entry, the number of columns n of the matrix B; $n \geq 0$
On exit, **n** is unchanged.

alpha
real*4 | real*8 | complex*8 | complex*16
On entry, specifies the scalar α.
On exit, **alpha** is unchanged.

a
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array A with dimensions lda by k.
If the multiplication is on the left side, $k \geq m$ and the leading m by m part of the array contains the matrix A.
If the multiplication is on the right side, $k \geq n$ and the leading n by n part of the array A must contain the matrix A.
In either case, when the leading part of the array is specified as the upper part, the upper triangular part of array A contains the upper-triangular part of the matrix A, and the lower-triangular part of matrix A is not referenced. When the lower part is specified, the lower triangular part of the array A contains the lower triangular part of the matrix A, and the upper-triangular part of A is not referenced.

If matrix A is unit-triangular, its diagonal elements are assumed to be unity and are not referenced.
On exit, **a** is unchanged.

lda
integer*4
On entry, the first dimension of A. When multiplication is on the left, $lda \geq \max(1, m)$. When multiplication is on the right, $lda \geq \max(1, n)$.
On exit, **lda** is unchanged.

b
real*4 | real*8 | complex*8 | complex*16
On entry, a two-dimensional array B of dimensions ldb by at least n. The leading m by n part of the array B must contain the right-hand-side matrix B.
On exit, **b** is overwritten by the m by n solution matrix X.

ldb
integer*4
On entry, the first dimension of B; $ldb \geq \max(1, m)$
On exit, **ldb** is unchanged.

Description

The _TRSM routines solve a triangular system of equations where the coefficient matrix A is a triangular matrix:

$$\mathrm{op}(A)X = \alpha B$$

$$X\mathrm{op}(A) = \alpha B$$

$\mathrm{op}(A) = A, A^T$, or A^H, α is a scalar, X and B are m by n matrices, and A is a unit or non-unit, upper- or lower-triangular matrix.

Example

```
REAL*8 A(100,40), B(40,20), ALPHA
M = 16
N = 18
LDA = 100
LDB = 40
ALPHA = 2.0D0
CALL DTRSM ('L','U','N','U',M,N,ALPHA,A,LDA,B,LDB)
```

This Fortran code solves the system $AX = \alpha B$ where A is an upper-triangular real matrix with a unit diagonal. X and B are 16 by 18 matrices. The leading 16 by 16 upper-triangular part of the array A must contain the upper-triangular matrix A. The leading 16 by 18 part of the array B must contain the matrix B. The lower-triangular part of A and the diagonal are not referenced. The leading 16 by 18 part of B is overwritten by the solution matrix X.

10

Using LAPACK Subprograms

LAPACK is a collection of Fortran 77 routines written to solve a wide array of problems in applied linear algebra. These routines provide DXML users with state of the art tools for linear equation solutions, eigenvalue problems, and linear least squares problems.

The routines are intended by their developers (at major government labs and research universities) to replace and expand the functionality of the famous LINPACK and EISPACK routines.

LAPACK provides enhancements in speed, primarily by utilizing blocked algorithms and the highly optimized DXML BLAS Level 3 and other BLAS routines. The collection also provides better accuracy and robustness than the LINPACK and EISPACK packages. Additionally, two LAPACK computational routines, {C,D,S,Z}GETRF and {C,D,S,Z}POTRF, have been parallelized for improved performance on multiprocessor systems. See Chapter 4 for information about using the DXML parallel library. The first public release of LAPACK, Version 1.0, was on February 29, 1992. LAPACK Version 2.0 was released on September 30, 1994 and is part of DXML.

This chapter provides information about the following topics:

- Overview of LAPACK (Section 10.1)
- Naming conventions and mnemonics (Section 10.2)
- A summary of LAPACK driver routines (Section 10.3)
- An example of how LAPACK is used (Section 10.4)
- How to experiment with performance parameters (Section 10.5)

To use LAPACK, you must purchase the LAPACK documentation, published in book form, by the Society for Industrial and Applied Math (SIAM) in 1995:

LAPACK Users' Guide, 2nd Edition, by E. Anderson et al,
SIAM
3600 University City Science Center
Philadelphia PA 19104-2688
ISBN 0-89871-345-5
Tel: 1-800-447-SIAM
FAX: 1-215-386-7999
email: service@siam.org

Information on ordering SIAM books, including the *LAPACK Users' Guide*, is also available through the World Wide Web at "http://www.siam.org."

You can display the html version of the *LAPACK Users' Guide* on the Internet through a mosaic interface by using the url:

`http://www.netlib.org/lapack/lug/lapack_lug.html`

A quick reference card for all the driver routines is included with SIAM's *LAPACK Users' Guide*.

The LAPACK project is currently centered at the University of Tennessee. Information on software releases, corrections to the user guide, and other information can be obtained by sending the following one-line message to `netlib@ornl.gov`:

```
send release_notes from lapack
```

The DXML release notes indicate the version of LAPACK included in the DXML product.

10.1 Overview

The computational tasks carried out by the LAPACK routines play an essential role in solving problems arising in virtually every area of scientific computation, simulation, or mathematical modeling. Optimal performance of the LAPACK routines is assured by the inclusion of high performance BLAS (particularly BLAS Level 3) as part of the DXML library, and the automatic choice of suitable blocking parameters.

The major capabilities provided by LAPACK include:

- Solution of linear systems of equations, that is, solving:

  $$Ax = b$$

  where A is a square matrix, and x and b are vectors.
- Solution of eigenvalue/eigenvector problems, that is, solving:

  $$Ax = \lambda * x$$

  or

  $$Ax = \lambda * B * x$$

  for either λ and/or x. Routines for more general eigenproblems and matrix factorizations involving eigenproblems are also provided.
- Solution of overdetermined systems by means of modern least squares methods including singular value decomposition. These problems involve linear systems of equations where A typically has many more rows than columns (so there are many more equations than unknowns).

Most of the capabilities in LAPACK are provided for several storage formats (full matrix, banded, packed symmetric, and so on). Consult SIAM's LAPACK Users' Guide for a complete description of LAPACK capabilities, including algorithm descriptions and further references.

10.2 Naming Conventions

LAPACK routine names have single letter prefixes indicating the precision (data type) of the input and/or output data:

Mnemonic	Meaning
S	real*4, single-precision
D	real*8, double-precision
C	complex*8, single-precision
Z	complex*16, double-precision

The LAPACK driver and computational (top level) routines always have names of the form:

```
_MMFF
_MMFFX
```

The underscore character (_) is one of the prefixes {S,D,C,Z}. MM is a two-letter code indicating the matrix type, that is, its storage and/or mathematical property. FF is a code of two or three letters indicating the type of mathematical task being performed. The letter *X* as the last letter on a routine name, indicates an **expert driver** routine, that is, a more sophisticated version of an existing routine which either uses or computes additional information about the problem, for example, condition numbers or error estimates.

Table 10–1 lists the mnemonics and their meaning used for the mm code. alphabetical order - this change needs to be done in next VMS release also.)

Table 10–1 Naming Conventions: Mnemonics for MM

Mnemonic	Meaning
GB	General band matrix
GE	General matrix
GG	General matrices, generalized problems (i.e. a pair of general matrices)
GT	General tridiagonal
HB	(Complex) Hermitian band
HE	Hermitian indefinite (C, Z prefixes only)
HP	Hermitian indefinite, packed storage (C,Z prefixes only)
PB	Positive definite, either symmetric or Hermitian, banded storage
PO	Positive definite, either symmetric or Hermitian
PP	Positive definite, either symmetric or Hermitian, packed storage
PT	Positive definite, either symmetric or Hermitian, tridiagonal
SB	(Real) symmetric band
SP	Symmetric indefinite, packed storage (S, D prefixes only)
ST	Symmetric tridiagonal
SY	Symmetric indefinite (S, D prefixes), or complex symmetric (C, Z prefixes)

Table 10–2 lists the mnemonics for the driver routines and their meaning.

Table 10–2 Naming Conventions: Mnemonics for FF

Mnemonic	Meaning
ES	Eigenvalues and Schur decomposition
EV	Eigenvalues and/or vectors
GLM	Generalized linear regression model
GS	Generalized eigenvalues, Schur form, and/or Schur vectors
GV	Generalized eigenvalues, and/or generalized eigenvectors
LS	Least squares solution, orthogonal factorization (general matrix only)
LSS	Least squares solution, singular value decomposition (general matrix only)
LSE	Least squares solution, Eigenvalues
SV	Linear system solutions
SVD	Singular value decomposition

Subprograms that provide linear system solutions use SV in the `ff` portion of their names. Thus, the simple driver routines for this task all have names of the form:

`{S,D,C,Z}mmSV.`

For example, to solve a general linear system with complex input data, you need to call CGESV.

To find the eigenvalues and (optionally) eigenvectors of a general complex Hermitian matrix stored in single precision, you need to call CHEEV.

The generalized eigenvalue routines involve problems of the form:

$$A * x = \lambda * B * x$$

The mnemonics for the mathematical task are **GV** (generalized eigenvalue/vector), or **GS** (generalized Schur factorization).

10.3 Summary of LAPACK Driver Subroutines

Table 10–3 lists simple driver routines for eigenvalue and singular value problems, linear equation solvers and linear least square problems.

Table 10–3 Simple Driver Routines

Routine	Function
Eigenvalue and Singular Value Problems	
SSYEV DSYEV CHEEV ZHEEV	Computes all eigenvalues and eigenvectors of a symmetric/Hermitian matrix.
SSPEV DSPEV CHPEV ZHPEV	Computes all eigenvalues and eigenvectors of a symmetric/Hermitian matrix in packed storage.

(continued on next page)

Table 10–3 (Cont.) Simple Driver Routines

Routine	Function
Eigenvalue and Singular Value Problems	
SSBEV DSBEV CHBEV ZHBEV	Computes all eigenvalues and eigenvectors of a symmeric/Hermitian band matrix.
SSTEV DSTEV	Computes all eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
SGEES DGEES CGEES ZGEES	Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.
SGEEV DGEEV CGEEV ZGEEV	Computes the eigenvalues and left and right eigenvectors of a general matrix.
SGESVD DGESVD CGESVD ZGESVD	Computes the singular value decomposition (SVD) of a general rectangular matrix.
SSYGV DSYGV CHEGV ZHEGV	Computes all eigenvalues and the eigenvectors of a generalized symmetric /Hermitian-definite generalized eigenproblem, $Ax = \lambda Bx$, or $BAx = \lambda x$.
SSPGV DSPGV CHPGV ZHPGV	Computes all eigenvalues and eigenvectors of a generalized symmetric /Hermitian-definite generalized eigenproblem, $Ax = \lambda Bx$, or $BAx = \lambda x$, where A and B are in packed storage.
SSBGV CHBGV	Computes all eigenvalues and eigenvectors of a generalized symmetric /Hermitian-definite and banded eigenproblem, $Ax = \lambda Bx$, or $BAx = \lambda x$.
SGEGS DGEGS CGEGV ZGEGV	Computes the generalized eigenvalues, Schur form, and left and/or right Schur vectors for a pair of nonsymmetric matrices.
SGGSVD DGGSVD CGGSVD ZGGSVD	Computes the generalized singular value decomposition.

(continued on next page)

Table 10–3 (Cont.) Simple Driver Routines

Routine	Function
Linear Equation Problems	
SGESV DGESV CGESV ZGESV	Solves a general system of linear equations AX=B.
SGBSV DGBSV CGBSV ZGBSV	Solves a general banded system of linear equations AX=B.
SGTSV DGTSV CGTSV ZGTSV	Solves a general tridiagonal system of linear equations AX=B.
SPOSV DPOSV CPOSV ZPOSV	Solves a symmetric/Hermitian positive definite system of linear equations AX=B.
SPPSV DPPSV CPPSV ZPPSV	Solves a symmetric/Hermitian positive definite system of linear equations AX=B, where A is held in packed storage.
SPBSV DPBSV CPBSV ZPBSV	Solves a symmetric/Hermitian positive definite banded system of linear equations AX=B.
SPTSV DPTSV CPTSV ZPTSV	Solves a symmetric/Hermitian positive definite tridiagonal system of linear equations AX=B.
SSYSV DSYSV CSYSV ZSYSV CHESV ZHESV	Solves a real/complex/complex symmetric/symmetric/Hermitian indefinite system of linear equations AX=B.
SSPSV DSPSV CSPSV ZSPSV CHPSV ZHPSV	Solves a real/complex/complex symmetric/symmetric/Hermitian indefinite system of linear equations AX=B, where A is held in packed storage.

(continued on next page)

Table 10–3 (Cont.) Simple Driver Routines

Routine	Function
Linear Least Squares Problems	
SGELS DGELS CGELS ZGELS	Computes the least squares solution to an over-determined system of linear equations, AX=B or A**H X=B, or the minimum norm solution of an under-determined system, where A is a general rectangular matrix of full rank, using a QR or LQ factorization of A.
SGELSS DGELSS CGELSS ZGELSS	Computes the minimum norm least squares solution to an over-determined or under-determined system of linear equations AX=B, using the singular value decomposition of A.
SGGGLM DGGGLM CGGGLM ZGGGLM	Solves the GLM (Generalized Linear Regression Model) using the GQR (Generalized QR) factorization.
SGGLSE DGGLSE CGGLSE ZGGLSE	Solves the LSE (Constrained Linear Least Squares Problem) using the GRQ (Generalized RQ) factorization.

Table 10–4 lists the following expert driver routines: linear equation, least square, and eigenvalue.

Table 10–4 Expert Driver Routines

Routine	Function
Linear Equation Problems	
SGESVX DGESVX CGESVX ZGESVX	Solves a general system of linear equations AX=B, A**T X=B or A**H X=B, and provides an estimate of the condition number and error bounds on the solution.
SGBSVX DGBSVX CGBSVX ZGBSVX	Solves a general banded system of linear equations AX=B, A**T X=B or A**H X=B, and provides an estimate of the condition number and the error bounds on the solution.
SGTSVX DGTSVX CGTSVX ZGTSVX	Solves a general tridiagonal system of linear equations AX=B, A**T X=B or A**H X=B, and provides an estimate of the condition number and the error bounds on the solution.
SPOSVX DPOSVX CPOSVX ZGOSVX	Solves a symmetric/Hermitian positive definite system of linear equations AX=B, and provides an estimate of the condition number and error bounds on the solution.
SPPSVX DPPSVX CPPSVX ZPPSVX	Solves a symmetric/Hermitian positive definite system of linear equations AX=B, where A is held in packed storage, and provides an estimate of the condition number and error bounds on the solution.
SPBSVX DPBSVX CPBSVX ZPBSVX	Solves a symmetric/Hermitian positive definite banded system of linear equations AX=B, and provides an estimate of the condition number and error bounds on the solution.

(continued on next page)

Table 10–4 (Cont.) Expert Driver Routines

Routine	Function
Linear Equation Problems	
SPTSVX DPTSVX CPTSVX ZPTSVX	Solves a symmetric/Hermitian positive tridiagonal system of linear equations AX=B, and provides an estimate of the condition number and error bounds on the solution.
SSYSVX DSYSVX CSYSVX ZSYSVX CHESVX ZHESVX	Solves a real/complex/complex symmetric/symmetric/Hermitian indefinite system of linear equations AX=B, and provides an estimate of the condition number and error bounds on the solution.
SSPSVX DSPSVX CSPSVX ZSPSVX CHPSVX ZHPSVX	Solves a real/complex/complex symmetric/symmetric/Hermitian indefinite system of linear equations AX=B, where A is held in packed storage, and provides an estimate of the condition number and error bounds on the solution.
Least Squares Problems	
SGELSX DGELSX CGELSX ZGELSX	Computes the minimum norm least squares solution to an over-determined or under-determined system of linear equations AX=B, using a complete orthogonal factorization of A.
Eigenvalue Problems	
SSYEVX DSYEVX CHEEVX ZHEEVX	Computes selected eigenvalues and eigenvectors of a symmetric/Hermitian matrix.
SSPEVX DSPEVX CHPEVX ZHPEVX	Computes selected eigenvalues and eigenvectors of a symmetric/Hermitian matrix in packed storage.
SSBEVX DSBEVX CHBEVX ZHBEVX	Computes selected eigenvalues and eigenvectors of a symmetric/Hermitian band matrix.
SSTEVX DSTEVX	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
SGEESX DGEESX CGEESX ZGEESX	Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization so that selected eigenvalues are at the top left of the Schur form, and computes reciprocal condition numbers for the average of the selected eigenvalues, and for the associated right invariant subspace.
SGEEVX DGEEVX CGEEVX ZGEEVX	Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary balancing of the matrix, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

The remaining routines are either computational or auxiliary. The computational routines solve a lower level of problem than the driver routines and the auxiliary routines solve an even lower level problem.

10.4 Example of LAPACK Use and Design

One of the most common uses for LAPACK is solving linear systems. If you are using the old LINPACK routines to solve $Ax = b$, you first call the subroutine `dgefa` to factor A, and then call `dgesl` to solve the system using the factored A.

The corresponding LAPACK call is as follows:

```
CALL DGESV(N,NRHS,A,LDA,IPIV,B,LDB,INFO)
```

The routine DGESV calls DGETRF to factor and overwrite A, and DGETRS to solve (overwriting B) using the LU factors computed by DGETRF.

Blocked BLAS Level 3 algorithms come into the picture because DGETRF calls the LAPACK routine ILAENV to obtain an optimal blocksize, and then DGETRF uses blocked algorithms (for example, DTRSM) to complete its task. All of this is invisible to normal top-level use because only the above call to the driver routine DGESV need be made.

10.5 Performance Tuning

In the public release of LAPACK, the routine ILAENV provides default values for blocksizes, crossover points, and other performance-tuning parameters for use with specified routines. These values are generally sufficient for routine use of LAPACK, and are invisible to the top-level user of the package. Certain lower-level LAPACK routines call ILAENV to obtain the value of a parameters of interest.

In this DXML release, LAPACK includes the routine XLAENV which enables experimentation with blocksizes, crossover points, and other performance-tuning parameters. Use of XLAENV is of interest to expert users familiar with LAPACK source code.

Thus, you can either use the default values or experiment with the parameters to tune the performance. The descriptions of ILAENV and XLAENV specify how to switch between these modes, and how to set custom parameter values. The following sample code segment sets a custom blocksize (in this case, 32) for DGESV:

```
CALL XLAENV(100,1)
IBLK=32
CALL XLAENV(1,IBLK)
...
CALL DGESV( ... )
CALL XLAENV(100,0)
```

The final call to XLAENV reverts subsequent code back to the normal mode of using the hard-coded parameters in ILAENV. The other calls are equally straightforward, and are explained in the header and comments for the ILAENV routine, and the entire source for the XLAENV routine, as displayed in Examples 10–1 and 10–2.

Example 10–1 ILAENV

```
      INTEGER          FUNCTION ILAENV( ISPEC, NAME, OPTS, N1, N2, N3,
     $                 N4 )
*
*  -- LAPACK auxiliary routine --
*  Digital version. Modified to allow expert user modifications of
*  blocksize and other parameters using the XLAENV routine,
*  via a common block shared by ILAENV and XLAENV.
*
*  Purpose
*  =======
*
*  ILAENV returns problem-dependent parameters for the local
*  environment.  See ISPEC for a description of the parameters.
*
*  In this (Digital) version, the problem-dependent parameters are either:
*   (a.) contained in the integer array IPARMS in the common block CLAENV
*        and the value with index ISPEC is copied to ILAENV.
*   (b.) hard coded in the code below. (normal use).
*   Common block initializaton forces IPARMS(100)=0 and thus
*   option (b.) is the default.
*
*   Option a.) is used if IPARMS(100)=1. (set by calling XLAENV)
*   This option is provided for parameter-tuning and testing purposes.
*   In this case values in IPARMS must be set to the desired values
*   by calling XLAENV.
*
*   Option b.) is used if IPARMS(100)=0:
*  This version provides a set of parameters which should give good,
*  but not optimal, performance on many of the currently available
*  computers.
*
*  Arguments
*  =========
*
*  ISPEC   (input) INTEGER
*          Specifies the parameter to be returned as the value of
*          ILAENV.
*          = 1: the optimal blocksize; if this value is 1, an unblocked
*               algorithm will give the best performance.
*          = 2: the minimum block size for which the block routine
*               should be used; if the usable block size is less than
*               this value, an unblocked routine should be used.
*          = 3: the crossover point (in a block routine, for N less
*               than this value, an unblocked routine should be used)
*          = 4: the number of shifts, used in the nonsymmetric
*               eigenvalue routines
*          = 5: the minimum column dimension for blocking to be used;
*               rectangular blocks must have dimension at least k by m,
*               where k is given by ILAENV(2,...) and m by ILAENV(5,...)
*          = 6: the crossover point for the SVD (when reducing an m by n
*               matrix to bidiagonal form, if max(m,n)/min(m,n) exceeds
*               this value, a QR factorization is used first to reduce
*               the matrix to a triangular form.)
*          = 7: the number of processors
```

(continued on next page)

Example 10–1 (Cont.) ILAENV

```
*          = 8: the crossover point for the multishift QR and QZ methods
*               for nonsymmetric eigenvalue problems.
*
*  NAME    (input) CHARACTER*(*)
*          The name of the calling subroutine, in either upper case or
*          lower case.
*
*  OPTS    (input) CHARACTER*(*)
*          The character options to the subroutine NAME, concatenated
*          into a single character string.  For example, UPLO = 'U',
*          TRANS = 'T', and DIAG = 'N' for a triangular routine would
*          be specified as OPTS = 'UTN'.
*
*  N1      (input) INTEGER
*  N2      (input) INTEGER
*  N3      (input) INTEGER
*  N4      (input) INTEGER
*          Problem dimensions for the subroutine NAME; these may not all
*          be required.
*
* (ILAENV) (output) INTEGER
*          >= 0: the value of the parameter specified by ISPEC
*          < 0:  if ILAENV = -k, the k-th argument had an illegal value.
*
*  Further Details
*  ===============
*
*  The following conventions have been used when calling ILAENV from the
*  LAPACK routines:
*  1)  OPTS is a concatenation of all of the character options to
*      subroutine NAME, in the same order that they appear in the
*      argument list for NAME, even if they are not used in determining
*      the value of the parameter specified by ISPEC.
*  2)  The problem dimensions N1, N2, N3, N4 are specified in the order
*      that they appear in the argument list for NAME.  N1 is used
*      first, N2 second, and so on, and unused problem dimensions are
*      passed a value of -1.
*  3)  The parameter value returned by ILAENV is checked for validity in
*      the calling subroutine.  For example, ILAENV is used to retrieve
*      the optimal blocksize for STRTRI as follows:
*
*      NB = ILAENV( 1, 'STRTRI', UPLO // DIAG, N, -1, -1, -1 )
*      IF( NB.LE.1 ) NB = MAX( 1, N )
*
*  =====================================================================
*     ..
*     .. Arrays in Common ..
      INTEGER            IPARMS( 100 ) /100*0/
*     ..
*     .. Common blocks ..
      COMMON             / CLAENV / IPARMS
```

Example 10–2 XLAENV

```
      SUBROUTINE XLAENV( ISPEC, NVALUE )
*
*  -- LAPACK auxiliary routine --
*  Digital version (shared common block with ILAENV)
*     .. Scalar Arguments ..
      INTEGER            ISPEC, NVALUE
*     ..
*
*  Purpose
*  =======
*
*  XLAENV sets certain machine- and problem-dependent quantities
*  which will later be retrieved by ILAENV.
*
*  Arguments
*  =========
*
*  ISPEC   (input) INTEGER
*          Specifies the parameter to be set in the COMMON array IPARMS.
*          = 1: the optimal blocksize; if this value is 1, an unblocked
*               algorithm will give the best performance.
*          = 2: the minimum block size for which the block routine
*               should be used; if the usable block size is less than
*               this value, an unblocked routine should be used.
*          = 3: the crossover point (in a block routine, for N less
*               than this value, an unblocked routine should be used)
*          = 4: the number of shifts, used in the nonsymmetric
*               eigenvalue routines
*          = 5: the minimum column dimension for blocking to be used;
*               rectangular blocks must have dimension at least k by m,
*               where k is given by ILAENV(2,...) and m by ILAENV(5,...)
*          = 6: the crossover point for the SVD (when reducing an m by n
*               matrix to bidiagonal form, if max(m,n)/min(m,n) exceeds
*               this value, a QR factorization is used first to reduce
*               the matrix to a triangular form)
*          = 7: the number of processors
*          = 8: another crossover point, for the multishift QR and QZ
*               methods for nonsymmetric eigenvalue problems.
*          = 100: with NVALUE=1, subsequent calls to ILAENV will fetch
*                 a requested value directly from the common block (rather
*                 than use the hard-coded values in ILAENV). With NVALUE=0,
*                 subsequent calls to ILAENV will use the hard-coded values.
*
*  NVALUE  (input) INTEGER
*          The value of the parameter specified by ISPEC.
*
*     .. Arrays in Common ..
      INTEGER            IPARMS( 100 )
*     ..
*     .. Common blocks ..
      COMMON             / CLAENV / IPARMS
*     ..
*     .. Save statement ..
      SAVE               / CLAENV /
```

(continued on next page)

Example 10–2 (Cont.) XLAENV

```
*     ..
*     .. Executable Statements ..
*
      IF( ISPEC.GE.1 .AND. ISPEC.LE.100 ) THEN
         IPARMS( ISPEC ) = NVALUE
      END IF
*
      RETURN
*
*     End of XLAENV
*
      END
```

10.6 Equivalence Between LAPACK and LINPACK/EISPACK Routines

The LAPACK equivalence utility provides the names and parameter lists of LAPACK routines that are equivalent to the LINPACK and EISPACK routines you specify. The utility command is as follows:

```
/usr/share/equivalence_lapack routine_name [routine_name...]
```

where you replace routine_name with the LINPACK and/or EISPACK routine names.

For example,

```
/usr/share/equivalence_lapack dgesl imtql1
```

returns:

```
DGESL:
      SUBROUTINE SGETRS( TRANS, N, NRHS, A, LDA, IPIV, B, LDB, INFO )
      SUBROUTINE DGETRS( TRANS, N, NRHS, A, LDA, IPIV, B, LDB, INFO )
IMTQL1:
      SUBROUTINE SSTEQR( COMPZ, N, D, E, Z, LDZ, WORK, INFO )
      SUBROUTINE DSTEQR( COMPZ, N, D, E, Z, LDZ, WORK, INFO )
```

The LINPACK or EISPACK routine names are to the left of the colons. The equivalent LAPACK routines and calling sequences are to the right of the colons.

This utility helps you to convert LINPACK and EISPACK routine calls to equivalent LAPACK routine calls. The utility has limitations in that the argument lists of the LAPACK routines are generally different from those of the corresponding LINPACK and EISPACK routines, and the workspace requirements are often different as well.

The LAPACK equivalence utility is installed at the following location:

```
/usr/opt/XMDLOAnnn/dxml/equivalence_lapack.c  (source code)
/usr/opt/XMDLOAnnn/dxml/equivalence_lapack    (executable)
```

where *nnn* refers to the version number for the release.

11

Using the Signal Processing Subprograms

DXML provides functions that perform the following signal processing operations:

- Fast Fourier transforms (FFT), described in Section 11.1
- Cosine (DCT) and Sine (DST) transforms, described Section 11.2.1
- Convolutions and correlations, described in Section 11.3
- Digital filters, described in Section 11.4

This chapter provides information about the following topics:

- Mathematical definitions of FFT (Section 11.1.1)
- Storing the Fourier coefficients (Section 11.1.2)
- Fourier transform functions (Section 11.1.3)
- Mathematical definitions of DCT and DST (Section 11.2)
- Cosine and Sine transform functions (Section 11.2.2.4)
- Mathematical description of convolution and correlation (Section 11.3.1)
- Convolution and correlation functions (Section 11.3.2)
- Mathematical description of a digital filter (Section 11.4.1)
- Controlling the digital filter (Sections 11.4.2 and 11.4.3)
- Filtering routines (Section 11.4.4)
- Error handling (Section 11.5)

The descriptions of each signal processing routine and Fortran code examples are at the end of this chapter. For information about using DXML routines with non-Fortran programming languages, see Section 3.4 and Section 3.4.2. Additional examples are included online in the `/usr/examples/dxml` directory. See `*.c` and `*.cxx`. If you need more comprehensive explanations of signal processing operations, consult the references provided in Appendix A.

Key Fast Fourier subprograms have been parallelized for improved peformance on multiprocessor systems. For a list of these subprograms and information about using the parallel library, see Chapter 4.

11.1 Fourier Transform

A finite or discrete Fourier transform decomposes a collection of data into component sine and cosine representation. A continuous Fourier transform of a function decomposes the function into a generalized sum of sinusoids of different frequencies. A continuous Fourier transform is represented graphically by a diagram that shows the amplitude and frequency of each of the sinusoids.

11.1.1 Mathematical Definition of FFT

The forward Fourier transform is a mathematical operation that converts numbers typically in the time domain to numbers typically in the frequency domain. The inverse Fourier transform performs the reverse operation, converting numbers in the frequency domain to numbers in the time domain.

This section reviews the mathematical definition of the various Fourier transforms.

11.1.1.1 One-Dimensional Continuous Fourier Transform

The analytical expression for the one-dimensional forward Fourier transform for continuous functions is commonly given as:

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{-i2\pi ft} dt \qquad (11\text{–}1)$$

where $H(f)$, a function in the frequency domain, is the Fourier transform of $h(t)$; $h(t)$, a function in the time domain, is the waveform to be decomposed into a sum of sinusoids; and $i = \sqrt{-1}$.

The one-dimensional inverse operation is given as:

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{i2\pi tf} df \qquad (11\text{–}2)$$

Variations on the definitions given in Equations (11–1) and (11–2) do exist. Sometimes a weighting function of $1/2\pi$ is found in front of the integral sign, and 2π is removed from the exponential term. See the references given in Appendix A for information on the various definitions of the continuous forward and inverse Fourier transforms.

11.1.1.2 One-Dimensional Discrete Fourier Transform

A digital computer cannot perform the integration indicated by the mathematical expressions for the continuous Fourier transform. Since a digital computer can only deal with discrete data points, the integration can only be approximated.

The Fourier transform functions must use a method known as the discrete Fourier transform (DFT) to approximate the continuous Fourier transform at discrete frequencies. The discrete Fourier transform does not process a continuous function. Instead, it processes discrete points or samples that give only an approximation of the continuous function. The continuous function might not be known analytically.

The simplest interpretation of the one-dimensional discrete Fourier transform results from interpreting a finite sequence as one period of a periodic sequence.

The mathematical expression for the one-dimensional discrete Fourier transform is given as:

$$H(k) = \sum_{m=0}^{n-1} h(m) e^{-i2\pi km/n} \qquad (11\text{–}3)$$

where m and k are indices, $k = 0, 1, 2, \ldots, n-1$; $H(k)$ and $h(m)$ represent discrete functions of uniformly spaced data in the time and frequency domains respectively; n is the data length, and $i = \sqrt{-1}$.

The one-dimensional inverse operation is given as:

$$h(m) = \frac{1}{n} \sum_{k=0}^{n-1} H(k) e^{i2\pi mk/n} \tag{11–4}$$

where $m = 0, 1, 2, \ldots, n-1$

11.1.1.3 Two-Dimensional Discrete Fourier Transform

The simplest interpretation of the two-dimensional discrete Fourier transform results from interpreting a two-dimensional sequence as one period of a doubly periodic sequence.

Thus, with $H(j,k)$ denoting the discrete Fourier transform of $h(m_1, m_2)$, the mathematical expression for the discrete Fourier transform in two dimensions is given as:

$$H(j,k) = \sum_{m_1=0}^{n_1-1} \sum_{m_2=0}^{n_2-1} h(m_1, m_2) e^{(-i2\pi jm_1)/n_1} e^{(-i2\pi km_2)/n_2} \tag{11–5}$$

where:

$j = 0, 1, 2, \ldots, n_1 - 1$
$k = 0, 1, 2, \ldots, n_2 - 1$
$i = \sqrt{-1}$

The inverse transform operation in two dimensions is given as:

$$h(m_1, m_2) = \frac{1}{n_1 n_2} \sum_{j=0}^{n_1-1} \sum_{k=0}^{n_2-1} H(j,k) e^{(i2\pi jm_1)/n_1} e^{(i2\pi km_2)/n_2} \tag{11–6}$$

The two-dimensional discrete Fourier transform given by Equation (11–5) can be rewritten as:

$$H(j,k) = \sum_{m_1=0}^{n_1-1} \left\{ \sum_{m_2=0}^{n_2-1} h(m_1, m_2) e^{(-i2\pi km_2)/n_2} \right\} e^{(-i2\pi jm_1)/n_1} \tag{11–7}$$

The quantity in braces, which we now call $G(m_1, k)$, is a two-dimensional sequence which allows $H(j,k)$ to be rewritten as:

$$G(m_1, k) = \sum_{m_2=0}^{n_2-1} h(m_1, m_2) e^{(-i2\pi km_2)/n_2} \tag{11–8}$$

$$H(j,k) = \sum_{m_1=0}^{n_1-1} G(m_1, k) e^{(-i2\pi m_1 j)/n_1} \tag{11–9}$$

Each row of G is the one-dimensional discrete Fourier transform of the corresponding row of h. Each column of H is the one-dimensional discrete Fourier transform of the corresponding column of G.

11.1.1.4 Three-Dimensional Discrete Fourier Transform

For three dimensions, the definition of the forward transform can be written as:

$$H(j,k,l) =$$

$$\sum_{m_1=0}^{n_1-1}\sum_{m_2=0}^{n_2-1}\sum_{m_3=0}^{n_3-1} h(m_1,m_2,m_3)e^{(-i2\pi jm_1)/n_1}e^{(-i2\pi km_2)/n_2}e^{(-i2\pi lm_3)/n_3} \qquad (11\text{–}10)$$

where:

$j = 0, 1, 2, \ldots, n_1 - 1$
$k = 0, 1, 2, \ldots, n_2 - 1$
$l = 0, 1, 2, \ldots, n_3 - 1$
$i = \sqrt{-1}$

The three-dimensional inverse operation can be written as:

$$h(m_1,m_2,m_3) =$$

$$\frac{1}{n_1n_2n_3}\sum_{j=0}^{n_1-1}\sum_{k=0}^{n_2-1}\sum_{l=0}^{n_3-1} H(j,k,l)e^{(i2\pi jm_1)/n_1}e^{(i2\pi km_2)/n_2}e^{(i2\pi lm_3)/n_3} \qquad (11\text{–}11)$$

where:

$m_1 = 0, 1, 2, \ldots, n_1 - 1$
$m_2 = 0, 1, 2, \ldots, n_2 - 1$
$m_3 = 0, 1, 2, \ldots, n_3 - 1$

11.1.1.5 Size of Fourier Transform

Table 11–1 shows the restrictions on the size of FFT.

Table 11–1 FFT Size

Complex FFT	1D	$n > 0$
	2D	$n_x > 0$
		$n_y > 0$
	3D	$n_x > 0$
		$n_y > 0$
		$n_z > 0$
Real FFT	1D	$n > 0$, n is even
	2D	$n_x > 0$, n is even
		n_x is even
		$n_y > 0$
	3D	$n_x > 0$, n is even
		n_x is even
		$n_y > 0$
		$n_z > 0$

11.1.2 Data Storage

The output of Fourier transforms can be stored in several ways, depending on the format of the data, and its symmetry. This section describes the efficiencies of the data storage method.

11.1.2.1 Storing the Fourier Coefficients of a 1D-FFT

When the Fourier transform of a real data sequence is performed, the transformed data is complex, and the identity shown in Equation (11–12) results from symmetry considerations:

$$H(n-k) = H^*(k) \tag{11–12}$$

where $H^*(k)$ is the complex conjugate of $H(k)$, and $k = 0, 1, 2, \ldots, \frac{n}{2}$.

Note that $H(0) = H^*(n) = H^*(0)$ and $H(\frac{n}{2}) = H^*(\frac{n}{2})$. Therefore, $H(0)$ and $H(\frac{n}{2})$ are real. So, to specify the Fourier transform of a real sequence, only $(\frac{n}{2} - 1)$ complex values and 2 real values are needed. The storage of the Fourier coefficient takes advantage of this.

When the Fourier transform of a complex data sequence is performed, the transformed data does not usually exhibit symmetry properties. The elements of the resulting output array are usually unique. As a result, all of the output data needs to be stored. DXML stores all the output data, and the length of the output array is the same as the length of the input array. In the following let X be the Fourier transform of x.

Storing the 1D-FFT in Real Data Format (R,R)

$$\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} \longleftrightarrow \begin{pmatrix} X_r(0) \\ X_r(1) \\ \vdots \\ X_r(\frac{n}{2}) \\ X_i(\frac{n}{2}-1) \\ X_i(\frac{n}{2}-2) \\ \vdots \\ X_i(1) \end{pmatrix}$$

In each type of transform, the resulting array has the size described in Table 11–2.

Table 11–2 Size of Output Array for SFFT and DFFT

Direction	Input Format	Output Format	Input Values		Output Values	
			Complex	Real	Complex	Real
F	R	C	0	n	$\frac{n}{2}+1$	0
B	C	R	$\frac{n}{2}+1$	0	0	n
F	R	R	0	n	$\frac{n}{2}-1$	2
B	R	R	$\frac{n}{2}-1$	2	0	n

Storing the 1D-FFT in Complex Data Format (R,C)

$$\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} \longleftrightarrow \begin{pmatrix} X_r(0) \\ X_i(0) \\ \vdots \\ X_r(\frac{n}{2}) \\ X_i(\frac{n}{2}) \end{pmatrix}$$

In each type of transform, the resulting array has the size described in Table 11–3.

Table 11–3 Size of Output Array from CFFT and ZFFT

Direction	Input Format	Output Format	Input Values		Output Values	
			Complex	Real	Complex	Real
F/B	R	R	n	0	n	0
F/B	C	C	n	0	n	0

Storing the 1D-FFT in Complex Data Format (C,C)

$$\begin{pmatrix} (x_r(0), x_i(0)) \\ \vdots \\ (x_r(N-1), x_i(N-1)) \end{pmatrix} \longleftrightarrow \begin{pmatrix} (X_r(0), X_i(0)) \\ \vdots \\ (X_r(N-1), X_i(N-1)) \end{pmatrix}$$

Storing the Transform of a Complex Sequence in Real Data Format (C,R)

$$\begin{pmatrix} x_r(0) \\ \vdots \\ x_r(N-1) \end{pmatrix} \begin{pmatrix} x_i(0) \\ \vdots \\ x_i(N-1) \end{pmatrix} \longleftrightarrow \begin{pmatrix} X_r(0) \\ \vdots \\ X_r(N-1) \end{pmatrix} \begin{pmatrix} X_i(0) \\ \vdots \\ X_i(N-1) \end{pmatrix}$$

11.1.2.2 Storing the Fourier Coefficients of 2D-FFT

When the 2D FFT of a real data sequence is performed, the transformed data is complex with the following symmetry:

$$H(i,j) = H^*(n_x - i, n_y - j) \tag{11–13}$$

for:

$0 \leq \text{i} \leq \text{n}_\text{x}\text{-}1$
$0 \leq \text{j} \leq \text{n}_\text{y}\text{-}1$

where $H_i(i,j) = 0$ for $(0,0), (\frac{n_x}{2}, 0)(0, \frac{n_y}{2})$, and$(\frac{n_x}{2}, \frac{n_y}{2})$. The storage of FFT takes advantage of this.

When the Fourier transform of a complex data sequence is performed, the transformed data does not usually exhibit symmetry properties. As a result, all of the output data needs to be stored. DXML stores all the output data, and the length of the output array is the same as the length of the input array.

Storing a Real Sequence and its Transform in Real Data Format

The following cases show how the value of X is stored in a location in array A. The index of array A starts at zero. When n_y is odd, cases 2 and 4 do not apply.

1. $X_r(0,0) \rightarrow A(0,0)$
2. $X_r(0, \frac{n_y}{2}) \rightarrow A(0, \frac{n_y}{2})$
3. $X_r(\frac{n_x}{2}, 0) \rightarrow A(\frac{n_x}{2}, 0)$
4. $X_r(\frac{n_x}{2}, \frac{n_y}{2}) \rightarrow A(\frac{n_x}{2}, \frac{n_y}{2})$
5. $X_r(i,j) \rightarrow A(i,j), X_i(i,j) \rightarrow A(i, n_y - j)$ for $0 < j < \frac{n_y}{2}, i = 0, \frac{n_x}{2}$
6. $X_r(i,j) \rightarrow A(i,j), X_i(i,j) \rightarrow A(n_x - i, j)$ for $0 \leq j \leq n_y - 1, 1 \leq i \leq \frac{n_x}{2} - 1$

The following is an example of $n_x = 8$ and $n_y = 4$:

$$\begin{pmatrix} x(0,0) & x(0,1) & x(0,2) & x(0,3) \\ x(1,0) & x(1,1) & x(1,2) & x(1,3) \\ x(2,0) & x(2,1) & x(2,2) & x(2,3) \\ x(3,0) & x(3,1) & x(3,2) & x(3,3) \\ x(4,0) & x(4,1) & x(4,2) & x(4,3) \\ x(5,0) & x(5,1) & x(5,2) & x(5,3) \\ x(6,0) & x(6,1) & x(6,2) & x(6,3) \\ x(7,0) & x(7,1) & x(7,2) & x(7,3) \end{pmatrix} \longleftrightarrow \begin{pmatrix} X_r(0,0) & X_r(0,1) & X_r(0,2) & X_i(0,1) \\ X_r(1,0) & X_r(1,1) & X_r(1,2) & X_r(1,3) \\ X_r(2,0) & X_r(2,1) & X_r(2,2) & X_r(2,3) \\ X_r(3,0) & X_r(3,1) & X_r(3,2) & X_r(3,3) \\ X_r(4,0) & X_r(4,1) & X_r(4,2) & X_i(4,1) \\ X_i(3,0) & X_i(3,1) & X_i(3,2) & X_i(3,3) \\ X_i(2,0) & X_i(2,1) & X_i(2,2) & X_i(2,3) \\ X_i(1,0) & X_i(1,1) & X_i(1,2) & X_i(1,3) \end{pmatrix}$$

Storing a Real Sequence and its Transform in Complex Data Format

$$X_r(i,j) \rightarrow A(2i,j) \qquad 0 \leq i \leq \frac{n_x}{2},\ 0 \leq j \leq n_y - 1$$

$$X_i(i,j) \rightarrow A(2i+1,j) \qquad 0 \leq i \leq \frac{n_x}{2},\ 0 \leq j \leq n_y - 1$$

$$\begin{pmatrix} x(0,0) & x(0,1) & x(0,2) & x(0,3) \\ x(1,0) & x(1,1) & x(1,2) & x(1,3) \\ x(2,0) & x(2,1) & x(2,2) & x(2,3) \\ x(3,0) & x(3,1) & x(3,2) & x(3,3) \\ x(4,0) & x(4,1) & x(4,2) & x(4,3) \\ x(5,0) & x(5,1) & x(5,2) & x(5,3) \\ x(6,0) & x(6,1) & x(6,2) & x(6,3) \\ x(7,0) & x(7,1) & x(7,2) & x(7,3) \end{pmatrix} \longleftrightarrow \begin{pmatrix} X_r(0,0) & X_r(0,1) & X_r(0,2) & X_r(0,3) \\ X_i(0,0) & X_i(0,1) & X_i(0,2) & X_i(0,3) \\ X_r(1,0) & X_r(1,1) & X_r(1,2) & X_r(1,3) \\ X_i(1,0) & X_i(1,1) & X_i(1,2) & X_i(1,3) \\ X_r(2,0) & X_r(2,1) & X_r(2,2) & X_r(2,3) \\ X_i(2,0) & X_i(2,1) & X_i(2,2) & X_i(2,3) \\ X_r(3,0) & X_r(3,1) & X_r(3,2) & X_r(3,3) \\ X_i(3,0) & X_i(3,1) & X_i(3,2) & X_i(3,3) \\ X_r(4,0) & X_r(4,1) & X_r(4,2) & X_r(4,3) \\ X_i(4,0) & X_i(4,1) & X_i(4,2) & X_i(4,3) \end{pmatrix}$$

Storing a Complex Sequence and its Transform in Complex Data Format

$$\begin{pmatrix} (x_r(0,0), x_i(0,0)) & (x_r(0,1), x_i(0,1)) & \dots & (x_r(n_y-1,0), x_i(n_y-1,0)) \\ \vdots & & & \\ (x_r(n_x-1,0), x_i(n_x-1,0)) & \dots & \dots & (x_r(n_x-1,n_y-1), x_i(n_x-1,n_y-1)) \end{pmatrix}$$

$$\updownarrow$$

$$\begin{pmatrix} (X_r(0,0), X_i(0,0)) & (X_r(0,1), X_i(0,1)) & \dots & (X_r(n_y-1,0), X_i(n_y-1,0)) \\ \vdots & & & \\ (X_r(n_x-1,0), X_i(n_x-1,0)) & \dots & \dots & (X_r(n_x-1,n_y-1), X_i(n_x-1,n_y-1)) \end{pmatrix}$$

Storing Complex Sequence in Real Data Format

$$\begin{pmatrix} x_r(0,0) & x_r(0,1) & \dots & x_r(0,n_y-1) \\ x_r(1,0) & x_r(1,1) & \dots & x_r(1,n_y-1) \\ \vdots & & & \\ x_r(n_x-1,0) & \dots & \dots & x_r(n_x-1,n_y-1) \end{pmatrix} \qquad \begin{pmatrix} x_i(0,0) & x_i(0,1) & \dots & x_i(0,n_y-1) \\ x_i(1,0) & x_i(1,1) & \dots & x_i(1,n_y-1) \\ \vdots & & & \\ x_i(n_x-1,0) & \dots & \dots & x_i(n_x-1,n_y-1) \end{pmatrix}$$

$$\updownarrow$$

$$\begin{pmatrix} X_r(0,0) & X_r(0,1) & \dots & X_r(0,n_y-1) \\ X_r(1,0) & X_r(1,1) & \dots & X_r(1,n_y-1) \\ \vdots & & & \\ X_r(n_x-1,0) & \dots & \dots & X_r(n_x-1,n_y-1) \end{pmatrix} \qquad \begin{pmatrix} X_i(0,0) & X_i(0,1) & \dots & X_i(0,n_y-1) \\ X_i(1,0) & X_i(1,1) & \dots & X_i(1,n_y-1) \\ \vdots & & & \\ X_i(n_x-1,0) & \dots & \dots & X_i(n_x-1,n_y-1) \end{pmatrix}$$

11.1.2.3 Storing the Fourier Coefficients of 3D-FFT

When the Fourier transform of a real data sequence is performed, the transformed data is complex, and the identity shown in Equation (11–14) results from symmetry considerations:

$$H(i,j,k) = H^*(n_x - i, n_y - j, n_z - k) \tag{11–14}$$

for:

$0 \le i \le n_x - 1$
$0 \le j \le n_y - 1$
$0 \le k \le n_z - 1$

where H^* is the complex conjugate of H.

When the Fourier transform of a complex data sequence is performed, the transformed data does not usually exhibit symmetry properties. The elements of the resulting output array are usually unique. As a result, all of the output data needs to be stored. DXML stores all the output data, and the length of the output array is the same as the length of the input array.

Storing Real Sequence in Real Data Format

$$X(i,j,k) = X(n_x - i, n_y - j, n_z - k)$$

For $k = 0$, $x(i,j,0)$ is stored in 2D format.
For $k = \frac{n_z}{2}$, and n_z is even, $x(i,j,\frac{n_z}{2})$ is stored in 2D format.
For $k \neq 0$, $\frac{n_z}{2}$, $1 \le k \le \frac{n_z}{2} - 1$.

This example is $(8,4,k)$.

$$\begin{pmatrix} x(0,0,k) & x(0,1,k) & x(0,2,k) & x(0,3,k) \\ x(1,0,k) & x(1,1,k) & x(1,2,k) & x(1,3,k) \\ x(2,0,k) & x(2,1,k) & x(2,2,k) & x(2,3,k) \\ x(3,0,k) & x(3,1,k) & x(3,2,k) & x(3,3,k) \\ x(4,0,k) & x(4,1,k) & x(4,2,k) & x(4,3,k) \\ x(5,0,k) & x(5,1,k) & x(5,2,k) & x(5,3,k) \\ x(6,0,k) & x(6,1,k) & x(6,2,k) & x(6,3,k) \\ x(7,0,k) & x(7,1,k) & x(7,2,k) & x(7,3,k) \end{pmatrix} \qquad \begin{pmatrix} x(0,0,n_z-k) & x(0,1,n_z-k) & x(0,2,n_z-k) & x(0,3,n_z-k) \\ x(1,0,n_z-k) & x(1,1,n_z-k) & x(1,2,n_z-k) & x(1,3,n_z-k) \\ x(2,0,n_z-k) & x(2,1,n_z-k) & x(2,2,n_z-k) & x(2,3,n_z-k) \\ x(3,0,n_z-k) & x(3,1,n_z-k) & x(3,2,n_z-k) & x(3,3,n_z-k) \\ x(4,0,n_z-k) & x(4,1,n_z-k) & x(4,2,n_z-k) & x(4,3,n_z-k) \\ x(5,0,n_z-k) & x(5,1,n_z-k) & x(5,2,n_z-k) & x(5,3,n_z-k) \\ x(6,0,n_z-k) & x(6,1,n_z-k) & x(6,2,n_z-k) & x(6,3,n_z-k) \\ x(7,0,n_z-k) & x(7,1,n_z-k) & x(7,2,n_z-k) & x(7,3,n_z-k) \end{pmatrix}$$

$$\updownarrow$$

$$\begin{pmatrix} X_r(0,0,k) & X_r(0,1,k) & X_r(0,2,k) & X_r(0,3,k) \\ X_r(1,0,k) & X_r(1,1,k) & X_r(1,2,k) & X_r(1,3,k) \\ X_r(2,0,k) & X_r(2,1,k) & X_r(2,2,k) & X_r(2,3,k) \\ X_r(3,0,k) & X_r(3,1,k) & X_r(3,2,k) & X_r(3,3,k) \\ X_r(4,0,k) & X_r(4,1,k) & X_r(4,2,k) & X_r(4,3,k) \\ X_i(3,0,k) & X_i(3,1,k) & X_i(3,2,k) & X_i(3,3,k) \\ X_i(2,0,k) & X_i(2,1,k) & X_i(2,2,k) & X_i(2,3,k) \\ X_i(1,0,k) & X_i(1,1,k) & X_i(1,2,k) & X_i(1,3,k) \end{pmatrix} \begin{pmatrix} X_i(0,0,k) & X_i(0,3,k) & X_i(0,2,k) & X_i(0,1,k) \\ X_r(1,0,n_z-k) & X_r(1,1,n_z-k) & X_r(1,2,n_z-k) & X_r(1,3,n_z-k) \\ X_r(2,0,n_z-k) & X_r(2,1,n_z-k) & X_r(2,2,n_z-k) & X_r(2,3,n_z-k) \\ X_r(3,0,n_z-k) & X_r(3,1,n_z-k) & X_r(3,2,n_z-k) & X_r(3,3,n_z-k) \\ X_i(4,0,k) & X_i(4,3,k) & X_i(4,2,k) & X_i(4,1,k) \\ X_i(3,0,n_z-k) & X_i(3,1,n_z-k) & X_i(3,2,n_z-k) & X_i(3,3,n_z-k) \\ X_i(2,0,n_z-k) & X_i(2,1,n_z-k) & X_i(2,2,n_z-k) & X_i(2,3,n_z-k) \\ X_i(1,0,n_z-k) & X_i(1,1,n_z-k) & X_i(1,2,n_z-k) & X_i(1,3,n_z-k) \end{pmatrix}$$

The following cases show how the value of X is stored in a location in array A. The index of array A starts at zero. When n_y is odd, cases 3 and 5 do not apply.

1. $i \neq 0$, $i \neq \frac{n_x}{2}$, $1 \leq i \leq \frac{n_x}{2} - 1$:

$$X_r(i,j,k) \rightarrow A(i,j,k)$$

$$X_i(i,j,k) \rightarrow A(n_x - i, j, k)$$

2. $i = 0, j = 0$:

$$X_r(0,0,k) \rightarrow A(0,0,k)$$

$$X_i(0,0,k) \rightarrow A(0,0,n_z - k)$$

3. $i = 0, j = \frac{n_y}{2}$:

$$X_r(0, \frac{n_y}{2}, k) \rightarrow A(0, \frac{n_y}{2}, k)$$

$$X_i(0, \frac{n_y}{2}, k) \rightarrow A(0, \frac{n_y}{2}, n_z - k)$$

4. $i = \frac{n_x}{2}, j = 0$:

$$X_r(\frac{n_x}{2}, 0, k) \rightarrow A(\frac{n_x}{2}, 0, k)$$

$$X_i(\frac{n_x}{2}, 0, k) \rightarrow A(\frac{n_x}{2}, 0, n_z - k)$$

5. $i = \frac{n_x}{2}, j = \frac{n_y}{2}$:

$$X_r(\frac{n_x}{2}, \frac{n_y}{2}, k) \rightarrow A(\frac{n_x}{2}, \frac{n_y}{2}, k)$$

$$X_i(\frac{n_x}{2}, \frac{n_y}{2}, k) \rightarrow A(\frac{n_x}{2}, \frac{n_y}{2}, n_z - k)$$

6. $i = 0, 1 \leq j \leq \frac{n_y}{2} - 1, \frac{n_y}{2} + 1 \leq j \leq n_y - 1$

$$X_r(0,j,k) \rightarrow A_r(0,j,k)$$

$$X_i(0,j,k) \rightarrow A_r(0, n_y - j, n_z - k)$$

7. $i = \frac{n_x}{2}, 1 \le j \le \frac{n_y}{2} - 1, \frac{n_y}{2} + 1 \le j \le n_y - 1$

$$X_r(\frac{n_x}{2}, j, k) \rightarrow A(\frac{n_x}{2}, j, k)$$

$$X_i(\frac{n_x}{2}, j, k) \rightarrow A(\frac{n_x}{2}, n_y - j, n_z - k)$$

Total memory required = $n_x n_y n_z$

Storing Real Sequence in Complex Format

$$X_r(i, j, k) \rightarrow A(2i, j, k) \quad 0 \le i \le \frac{n_x}{2}, 0 \le j \le n_y - 1, 0 \le k \le n_z - 1$$

$$X_i(i, j, k) \rightarrow A(2i + 1, j, k) \quad 0 \le i \le \frac{n_x}{2}, 0 \le j \le n_y - 1, 0 \le k \le n_z - 1$$

$$\begin{pmatrix} X_r(0,0,k) & X_r(0,1,k) & X_r(0,2,k) & X_r(0,3,k) \\ X_i(0,0,k) & X_i(0,1,k) & X_i(0,2,k) & X_i(0,3,k) \\ X_r(1,0,k) & X_r(1,1,k) & X_r(1,2,k) & X_r(1,3,k) \\ X_i(1,0,k) & X_i(1,1,k) & X_i(1,2,k) & X_i(1,3,k) \\ X_r(2,0,k) & X_r(2,1,k) & X_r(2,2,k) & X_r(2,3,k) \\ X_i(2,0,k) & X_i(2,1,k) & X_i(2,2,k) & X_i(2,3,k) \\ X_r(3,0,k) & X_r(3,1,k) & X_r(3,2,k) & X_r(3,3,k) \\ X_i(3,0,k) & X_i(3,1,k) & X_i(3,2,k) & X_i(3,3,k) \\ X_r(4,0,k) & X_r(4,1,k) & X_r(4,2,k) & X_r(4,3,k) \\ X_i(4,0,k) & X_i(4,1,k) & X_i(4,2,k) & X_i(4,3,k) \end{pmatrix}$$

Total memory required = $2(\frac{n_x}{2} + 1)(n_y n_z) = n_x n_y n_z + 2 n_y n_z$

Storing Complex Sequence in Complex Data Format

$$\begin{pmatrix} (x_r(0,0,k), x_i(0,0,k)) & (x_r(0,1,k), x_i(0,1,k)) & \dots & (x_r(n_y-1,0,k), x_i(n_y-1,0,k)) \\ \vdots & & & \\ (x_r(n_x-1,0,k), x_i(n_x-1,0,k)) & \dots & \dots & (x_r(n_x-1,n_y-1,k), x_i(n_x-1,n_y-1,k)) \end{pmatrix}$$

$$\updownarrow$$

$$\begin{pmatrix} (X_r(0,0,k), X_i(0,0,k)) & (X_r(0,1,k), X_i(0,1,k)) & \dots & (X_r(n_y-1,0,k), X_i(n_y-1,0,k)) \\ \vdots & & & \\ (X_r(n_x-1,0,k), X_i(n_x-1,0,k)) & \dots & \dots & (X_r(n_x-1,n_y-1,k), X_i(n_x-1,n_y-1,k)) \end{pmatrix}$$

Storing Complex Sequence in Real Data Format

$$\begin{pmatrix} x_r(0,0,k) & x_r(0,1,k) & \dots & x_r(0,n_y-1,k) \\ x_r(1,0,k) & x_r(1,1,k) & \dots & x_r(1,n_y-1,k) \\ \vdots & & & \\ x_r(n_x-1,0,k) & \dots & \dots & x_r(n_x-1,n_y-1,k) \end{pmatrix} \begin{pmatrix} x_i(0,0,k) & x_i(0,1,k) & \dots & x_i(0,n_y-1,k) \\ x_i(1,0,k) & x_i(1,1,k) & \dots & x_i(1,n_y-1,k) \\ \vdots & & & \\ x_i(n_x-1,0,k) & \dots & \dots & x_i(n_x-1,n_y-1,k) \end{pmatrix}$$

$$\updownarrow$$

$$\begin{pmatrix} X_r(0,0,k) & X_r(0,1,k) & \dots & X_r(0,n_y-1,k) \\ X_r(1,0,k) & X_r(1,1,k) & \dots & X_r(1,n_y-1,k) \\ \vdots & & & \\ X_r(n_x-1,0,k) & \dots & \dots & X_r(n_x-1,n_y-1,k) \end{pmatrix} \begin{pmatrix} X_i(0,0,k) & X_i(0,1,k) & \dots & X_i(0,n_y-1,k) \\ X_i(1,0,k) & X_i(1,1,k) & \dots & X_i(1,n_y-1,k) \\ \vdots & & & \\ X_i(n_x-1,0,k) & \dots & \dots & X_i(n_x-1,n_y-1,k) \end{pmatrix}$$

11.1.2.4 Storing the Fourier Coefficient of Group FFT

Storing the output of a group FFT operation is similar to the methods used for one-dimensional FFT data storage.

Storing Real Sequence in Real Data Format

$$\begin{pmatrix} x_0 x_1 \dots x_{n-1} \\ y_0 y_1 \dots y_{n-1} \\ \vdots \\ z_0 z_1 \dots z_{n-1} \end{pmatrix} \longleftrightarrow \begin{pmatrix} X_r(0) X_r(1) \dots X_r(\frac{n}{2}) X_i(\frac{n}{2}-1) \dots X_i(1) \\ Y_r(0) Y_r(1) \dots Y_r(\frac{n}{2}) Y_i(\frac{n}{2}-1) \dots Y_i(1) \\ \vdots \\ Z_r(0) Z_r(1) \dots Z_r(\frac{n}{2}) Z_i(\frac{n}{2}-1) \dots Z_i(1) \end{pmatrix}$$

Storing Real Sequence in Complex Data Format

$$\begin{pmatrix} x_0 x_1 \dots x_{n-1} \\ y_0 y_1 \dots y_{n-1} \\ \vdots \\ z_0 z_1 \dots z_{n-1} \end{pmatrix} \longleftrightarrow \begin{pmatrix} (X_r(0) X_i(0)) \dots (X_r(\frac{n}{2}) X_i(\frac{n}{2})) \\ (Y_r(0) Y_i(0)) \dots (Y_r(\frac{n}{2}) Y_i(\frac{n}{2})) \\ \vdots \\ (Z_r(0) Z_i(0)) \dots (Z_r(\frac{n}{2}) Z_i(\frac{n}{2})) \end{pmatrix}$$

Storing Complex Sequence in Complex Data Format

$$\begin{pmatrix} (x_r(0), x_i(0)) \dots (x_r(n-1), x_i(n-1)) \\ (y_r(0), y_i(0)) \dots (y_r(n-1), y_i(n-1)) \\ \vdots \\ (z_r(0), z_i(0)) \dots (z_r(n-1), z_i(n-1)) \end{pmatrix}$$

$$\updownarrow$$

$$\begin{pmatrix} (X_r(0), X_i(0)) \dots (X_r(n-1), X_i(n-1)) \\ (Y_r(0), Y_i(0)) \dots (Y_r(n-1), Y_i(n-1)) \\ \vdots \\ (Z_r(0), Z_i(0)) \dots (Z_r(n-1), Z_i(n-1)) \end{pmatrix}$$

Storing Complex Sequence in Real Data Format

$$\begin{pmatrix} x_r(0)x_r(1)\ldots x_r(n-1) \\ y_r(0)y_r(1)\ldots y_r(n-1) \\ \vdots \\ z_r(0)z_r(1)\ldots z_r(n-1) \end{pmatrix} \begin{pmatrix} x_i(0)x_i(1)\ldots x_i(n-1) \\ y_i(0)y_i(1)\ldots y_i(n-1) \\ \vdots \\ z_i(0)z_i(1)\ldots z_i(n-1) \end{pmatrix}$$

$$\updownarrow$$

$$\begin{pmatrix} X_r(0)X_r(1)\ldots X_r(n-1) \\ Y_r(0)Y_r(1)\ldots Y_r(n-1) \\ \vdots \\ Z_r(0)Z_r(1)\ldots Z_r(n-1) \end{pmatrix} \begin{pmatrix} X_i(0)X_i(1)\ldots X_i(n-1) \\ Y_i(0)Y_i(1)\ldots Y_i(n-1) \\ \vdots \\ Z_i(0)Z_i(1)\ldots Z_i(n-1) \end{pmatrix}$$

11.1.3 DXML's FFT Functions

The DXML provides a comprehensive set of Fourier transform functions covering the following options:

- Dimensions: one, two, or three
- Direction: forward or inverse
- Data type: real or complex
- Data format: real or complex
- Precision: single or double

This section describes the effects of these options.

11.1.3.1 Choosing Data Lengths

The data length is the number of elements being transformed. This length determines the duration and method of computation for FFT operations. To save computation time, choose a nonprime value for the data length to make use of the fast algorithm. A prime value is slower because it cannot use the FFT algorithm.

Choose a value according to the following hierarchy, arranged from best performance to worst performance:

1. The data length is a power of 2.
2. The data length is the product of the small primes 2, 3, and 5.
3. The data length is a product of primes which may be greater than 7.
4. The data length is prime.

Although the performance is best when the data length is a power of 2, none of the functions limit the data length to a power of 2 as is commonly found in other FFT libraries.

11.1.3.2 Input and Output Data Format

The permitted format of input and output data is specified by the arguments **input_format** and **output_format**. Table 11–4 shows the values you specify for these arguments for real and complex, forward and inverse transforms.

Table 11–4 Input and Output Format Argument Values

Direction	Input Format	Output Format
Real Transforms		
Forward	'R'	'C'
Backward	'C'	'R'
Either	'R'	'R'
Complex Transforms		
Either	'R'	'R'
	'C'	'C'

If you use an unsupported combination of input and output format, you receive one of the status values listed in Table 11–5.

Table 11–5 Status Values for Unsupported Input and Output Combinations

Value	Function	Meaning
16	DXML_BAD_FORMAT_STRING()	The specified combination of formats is not supported.
18	DXML_BAD_FORMAT_FOR_DIRECTION()	The specified combination of formats is not supported for the specified direction.

Use the supported combinations as shown in Table 11–4.

11.1.3.3 Using the Internal Data Structures

Every time you perform an FFT operation, the software builds an internal data structure. The data structure provides a convenient way of storing attributes of the FFT such as the data length and type of stride allowed, as well as pointers to virtual memory.

If a program performs repeated FFTs, the process is more efficient if the internal data structure is saved and reused. This saves the recalculation of the same internal data structure over and over again. For this reason, DXML provides two ways of performing fast Fourier transforms, each with its own advantage:

- One-step FFT
  If your program performs only one or a few FFT operations, use one subroutine to initialize, apply, and remove the internal data structure.
- Three-step FFT
  If your program repeats the same FFT operation, use the set of three subroutines:
  - The _INIT subroutine builds the internal data structures.

- The _APPLY subroutine uses the internal data structures to compute the FFT.
- The _EXIT subroutine deallocates the virtual memory that was allocated by the _INIT subroutine.

The internal data structures are constant for a specified length of data, the data type, and for one-, two-, or three-dimensional transforms. When you change one of these characteristics, you must reinitialize the data structure. So, after you call the _INIT routine, you can call the _APPLY routine many times, as long as your data length and data type remains the same.

The three-step subroutines each use an **fft_struct** argument to manipulate the internal data structure. You declare the **fft_struct** using the appropriate call for the data format:

```
RECORD /DXML_S_FFT_STRUCTURE/
RECORD /DXML_D_FFT_STRUCTURE/
RECORD /DXML_C_FFT_STRUCTURE/
RECORD /DXML_Z_FFT_STRUCTURE/
```

You do not have to do anything else with this argument. For example, to perform a three-step, one-dimensional, single-precision complex FFT, declare the variable **fft_struct** of type **RECORD /DXML_C_FFT_STRUCTURE/**, as shown in the following code example:

```
 INCLUDE '/usr/include/DXMLDEF.FOR'
 REAL*4 IN_R(N,100),IN_I(N,100),OUT_R(N,100),OUT_I(N,100)
 COMPLEX*8 IN(N,100),OUT(N,100)
 INTEGER*4 STATUS
 CHARACTER*1 DIRECTION
 RECORD /DXML_C_FFT_STRUCTURE/ FFT_STRUCT

 DIRECTION = 'F'
 STATUS = CFFT_INIT(N,FFT_STRUCT,.TRUE.)
 DO I=1,100
     STATUS = CFFT_APPLY('C','C',DIRECTION,IN(1,I),OUT(1,I),
1                    FFT_STRUCT,1)
 ENDDO
 DO I=1,100
      STATUS = CFFT_APPLY('R','R',DIRECTION,IN_R(1,I),IN_I(1,I),
1                    OUT_R(1,I),OUT_I(1,I),FFT_STRUCT,1)

 ENDDO
 STATUS = CFFT_EXIT(FFT_STRUCT)
```

11.1.3.4 Naming Conventions

A Fourier transform subroutine has a name composed of character groups that tell you about the subroutine's operation. Table 11–6 shows the character groups used in the subroutine names and what they mean.

Table 11–6 Naming Conventions: Fourier Transform Functions

Character Group	Mnemonic	Meaning
First group	S	Single-precision real data
	D	Double-precision real data
	C	Single-precision complex data

(continued on next page)

Table 11–6 (Cont.) Naming Conventions: Fourier Transform Functions

Character Group	Mnemonic	Meaning
	Z	Double-precision complex data
Second group	FFT	Fast Fourier Transform
Third group	No mnemonic	One-step operation
	_INIT	Three-step operation: building of internal data structures
	_APPLY	Three-step operation: perform FFT
	_EXIT	Three-step operation: deallocation of virtual memory in internal data structure
Fourth group	No mnemonic	One-dimensional FFT
	_2D	Two-dimensional FFT
	_3D	Three-dimensional FFT
	_GRP	FFT of grouped data

For example, DFFT_APPLY is the DXML function for calculating in double-precision arithmetic the one-dimensional FFT of real data by applying the internal data structures that were calculated in the first step, DFFT_INIT, of this three-step operation.

11.1.3.5 Summary of Fourier Transform Functions

Table 11–7 summarizes the Fourier transform functions that perform the entire transform, either forward or reverse, in one step.

Table 11–7 Summary of One-Step Fourier Transform Functions

Name	Operation
SFFT	Calculates, in single-precision arithmetic, the fast forward or inverse Fourier transform of one-dimensional, real data.
DFFT	Calculates, in double-precision arithmetic, the fast forward or inverse Fourier transform of one-dimensional, real data.
CFFT	Calculates, in single-precision arithmetic, the fast forward or inverse Fourier transform of one-dimensional, complex data.
ZFFT	Calculates, in double-precision arithmetic, the fast forward or inverse Fourier transform of one-dimensional, complex data.
SFFT_2D	Calculates, in single-precision arithmetic, the fast forward or inverse Fourier transform of two-dimensional, real data.
DFFT_2D	Calculates, in double-precision arithmetic, the fast forward or inverse Fourier transform of two-dimensional, real data.
CFFT_2D	Calculates, in single-precision arithmetic, the fast forward or inverse Fourier transform of two-dimensional, complex data.

(continued on next page)

Table 11–7 (Cont.) Summary of One-Step Fourier Transform Functions

Name	Operation
ZFFT_2D	Calculates, in double-precision arithmetic, the fast forward or inverse Fourier transform of two-dimensional, complex data.
SFFT_3D	Calculates, in single-precision arithmetic, the fast forward or inverse Fourier transform of three-dimensional, real data.
DFFT_3D	Calculates, in double-precision arithmetic, the fast forward or inverse Fourier transform of three-dimensional, real data.
CFFT_3D	Calculates, in single-precision arithmetic, the fast forward or inverse Fourier transform of three-dimensional, complex data.
ZFFT_3D	Calculates, in double-precision arithmetic, the fast forward or inverse Fourier transform of three-dimensional, complex data.
SFFT_GRP	Calculates, in single-precision arithmetic, the fast forward or inverse Fourier transform of a group of real data.
DFFT_GRP	Calculates, in double-precision arithmetic, the fast forward or inverse Fourier transform of a group of real data.
CFFT_GRP	Calculates, in single-precision arithmetic, the fast forward or inverse Fourier transform of a group of complex data.
ZFFT_GRP	Calculates, in double-precision arithmetic, the fast forward or inverse Fourier transform of a group of complex data.

Table 11–8 summarizes the three-step Fourier transform functions. Each function is either an initialization step, an application step, or an exit step.

Table 11–8 Summary of Three-Step Fourier Transform Functions

Name	Operation
SFFT_INIT	Calculates internal data structures.
SFFT_APPLY	Applies SFFT_INIT's internal data structure to calculate, in single-precision arithmetic, the fast forward or inverse Fourier transform of one-dimensional, real data.
SFFT_EXIT	Deallocates the virtual memory allocated by SFFT_INIT.
DFFT_INIT	Calculates internal data structures.
DFFT_APPLY	Applies DFFT_INIT's internal data structure to calculate, in double-precision arithmetic, the fast forward or inverse Fourier transform of one-dimensional, real data.
DFFT_EXIT	Deallocates the virtual memory allocated by DFFT_INIT.
CFFT_INIT	Calculates internal data structures.
CFFT_APPLY	Applies CFFT_INIT's internal data structure to calculate, in single-precision arithmetic, the fast forward or inverse Fourier transform of one-dimensional, complex data.
CFFT_EXIT	Deallocates the virtual memory allocated by CFFT_INIT.

(continued on next page)

Table 11–8 (Cont.) Summary of Three-Step Fourier Transform Functions

Name	Operation
ZFFT_INIT	Calculates internal data structures.
ZFFT_APPLY	Applies ZFFT_INIT's internal data structure to calculate, in double-precision arithmetic, the fast forward or inverse Fourier transform of one-dimensional, complex data.
ZFFT_EXIT	Deallocates the virtual memory allocated by ZFFT_INIT.
SFFT_INIT_2D	Calculates internal data structures.
SFFT_APPLY_2D	Applies SFFT_INIT_2D's internal data structure to calculate, in single-precision arithmetic, the fast forward or inverse Fourier transform of two-dimensional, real data.
SFFT_EXIT_2D	Deallocates the virtual memory allocated by SFFT_INIT_2D.
DFFT_INIT_2D	Calculates internal data structures.
DFFT_APPLY_2D	Applies DFFT_INIT_2D's internal data structure to calculate, in double-precision arithmetic, the fast forward or inverse Fourier transform of two-dimensional, real data.
DFFT_EXIT_2D	Deallocates the virtual memory allocated by DFFT_INIT_2D.
CFFT_INIT_2D	Calculates internal data structures.
CFFT_APPLY_2D	Applies CFFT_INIT_2D's internal data structure to calculate, in single-precision arithmetic, the fast forward or inverse Fourier transform of two-dimensional, complex data.
CFFT_EXIT_2D	Deallocates the virtual memory allocated by CFFT_INIT_2D.
ZFFT_INIT_2D	Calculates internal data structures.
ZFFT_APPLY_2D	Applies ZFFT_INIT_2D's internal data structure to calculate, in double-precision arithmetic, the fast forward or inverse Fourier transform of two-dimensional, complex data.
ZFFT_EXIT_2D	Deallocates the virtual memory allocated by ZFFT_INIT_2D.
SFFT_INIT_3D	Calculates internal data structures.
SFFT_APPLY_3D	Applies SFFT_INIT_3D's internal data structure to calculate, in single-precision arithmetic, the fast forward or inverse Fourier transform of three-dimensional, real data.
SFFT_EXIT_3D	Deallocates the virtual memory allocated by SFFT_INIT_3D.
DFFT_INIT_3D	Calculates internal data structures.
DFFT_APPLY_3D	Applies DFFT_INIT_3D's internal data structure to calculate, in double-precision arithmetic, the fast forward or inverse Fourier transform of three-dimensional, real data.
DFFT_EXIT_3D	Deallocates the virtual memory allocated by DFFT_INIT_3D.

(continued on next page)

Table 11–8 (Cont.) Summary of Three-Step Fourier Transform Functions

Name	Operation
CFFT_INIT_3D	Calculates internal data structures.
CFFT_APPLY_3D	Applies CFFT_INIT_3D's internal data structure to calculate, in single-precision arithmetic, the fast forward or inverse Fourier transform of three-dimensional, complex data.
CFFT_EXIT_3D	Deallocates the virtual memory allocated by CFFT_INIT_3D.
ZFFT_INIT_3D	Calculates internal data structures.
ZFFT_APPLY_3D	Applies ZFFT_INIT_3D's internal data structure to calculate, in double-precision arithmetic, the fast forward or inverse Fourier transform of three-dimensional, complex data.
ZFFT_EXIT_3D	Deallocates the virtual memory allocated by ZFFT_INIT_3D.
SFFT_INIT_GRP	Calculates internal data structures.
SFFT_APPLY_GRP	Applies SFFT_INIT_GRP's internal data structure to calculate, in single-precision arithmetic, the fast forward or inverse Fourier transform of grouped, real data.
SFFT_EXIT_GRP	Deallocates the virtual memory allocated by SFFT_INIT_GRP.
DFFT_INIT_GRP	Calculates internal data structures.
DFFT_APPLY_GRP	Applies DFFT_INIT_GRP's internal data structure to calculate, in double-precision arithmetic, the fast forward or inverse Fourier transform of grouped, real data.
DFFT_EXIT_GRP	Deallocates the virtual memory allocated by DFFT_INIT_GRP.
CFFT_INIT_GRP	Calculates internal data structures.
CFFT_APPLY_GRP	Applies CFFT_INIT_GRP's internal data structure to calculate, in single-precision arithmetic, the fast forward or inverse Fourier transform of grouped, complex data.
CFFT_EXIT_GRP	Deallocates the virtual memory allocated by CFFT_INIT_GRP.
ZFFT_INIT_GRP	Calculates internal data structures.
ZFFT_APPLY_GRP	Applies SFFT_INIT_GRP's internal data structure to calculate, in double-precision arithmetic, the fast forward or inverse Fourier transform of grouped, complex data.
ZFFT_EXIT_GRP	Deallocates the virtual memory allocated by ZFFT_INIT_GRP.

11.2 Cosine and Sine Transforms

Similar to the discrete Fourier transform, the Cosine and the Sine transforms decompose a collection of data into a finite sum of sinusoids of different frequencies. The differences among the three transforms are in the assumptions about the data to be transformed. For example, the Fourier transform makes no assumptions about the data as long as the existence conditions for the Fourier integral are satisfied. The Sine transform assumes that the functions to be

transformed are odd. The Cosine transform assumes that the functions to be transformed are even.

11.2.1 Mathematical Definitions of DCT and DST

This section reviews the mathematical definitions of the Sine and the Cosine transforms.

11.2.1.1 One-Dimensional Continuous Cosine and Sine Transforms

The analytical expressions for the one-dimensional forward Cosine transform and the one-dimensional forward Sine transform for continuous functions are commonly given as:

$$C(\omega) = \sqrt{\frac{2}{\pi}} \int_0^\infty h(t)\ cos(\omega t)\ dt \tag{11–15}$$

and:

$$S(\omega) = \sqrt{\frac{2}{\pi}} \int_0^\infty h(t)\ sin(\omega t)\ dt \tag{11–16}$$

respectively. $C(\omega)$ and $S(\omega)$, functions in the frequency domain, are the Cosine and the Sine transforms of $h(t)$. $h(t)$, a function in the time domain, is the waveform to be decomposed into a sum of sinusoids.

Equations for reversing the Cosine and the Sine transforms are as follows:

$$h(t) = \sqrt{\frac{2}{\pi}} \int_0^\infty C(\omega)\ cos(\omega t)\ d\omega \tag{11–17}$$

and:

$$h(t) = \sqrt{\frac{2}{\pi}} \int_0^\infty S(\omega)\ sin(\omega t)\ d\omega \tag{11–18}$$

respectively.

11.2.1.2 One-Dimensional Discrete Cosine and Sine Transforms

Similar to continuous Fourier transforms, continuous Cosine and Sine transforms can be discretized. Unlike the Fourier transforms, there are multiple discretization schemes that lead to multiple definitions of the discrete Cosine and Sine transforms. The simplest discretization uses the following transforms:

Type I Cosine Transform

$$C^I(k) = \alpha_k \sum_{m=0}^{N} \alpha_m\ h(m)\ cos(\frac{\pi mk}{N}) \tag{11–19}$$

where $k = 0, ..., N$.

Type I Sine Transform

$$S^I(k) = \sum_{m=1}^{N-1} h(m)\ sin(\frac{\pi mk}{N}) \tag{11–20}$$

where $k = 1, ..., N-1$ and where

$$\alpha_k = \left\{ \begin{array}{ll} \frac{1}{\sqrt{2}} & k = 0, N \\ 1 & k \neq 0, N \end{array} \right\} \tag{11–21}$$

The inverse formulas for the Type I transforms are the following:

Inverse Type I Cosine Transform

$$h(m) = \alpha_m \frac{2}{N} \sum_{k=0}^{N} \alpha_k C^I(k)\ cos(\frac{\pi mk}{N}) \tag{11–22}$$

where $m = 0, ..., N$.

Inverse Type I Sine Transform

$$h(m) = \frac{2}{N} \sum_{k=1}^{N-1} S^I(k)\ sin(\frac{\pi mk}{N}) \tag{11–23}$$

where $m = 1, ..., N-1$ and with α_k defined in (11–21).

Additionally, DXML implements the following Type II transforms:

Type II Cosine Transform

$$C^{II}(k) = \alpha_k \sum_{m=0}^{N-1} h(m)\ cos[\frac{(2m+1)k\pi}{2N}] \tag{11–24}$$

where $k = 0, ..., N-1$.

Type II Sine Transform

$$S^{II}(k) = \alpha_k \sum_{m=1}^{N} h(m)\ sin[\frac{(2m-1)k\pi}{2N}] \tag{11–25}$$

where $k = 1, ..., N$.

Inverse Type II Cosine Transform

$$h(m) = \frac{2}{N} \sum_{k=0}^{N} \alpha_k\ C^{II}(k)\ cos[\frac{(2m+1)k\pi}{2N}] \tag{11–26}$$

where $m = 0, ..., N-1$.

Inverse Type II Sine Transform

$$h(m) = \frac{2}{N} \sum_{k=1}^{N} \alpha_k\ S^{II}(k)\ sin[\frac{(2m-1)k\pi}{2N}] \tag{11–27}$$

where $m = 1, ..., N$ with α_k defined in (11–21). Although there are two other forms of Cosine and Sine transforms, they are not implemented in DXML. See the references given in Appendix A for information on the other forms of Cosine and Sine transforms.

11.2.1.3 Size of Cosine and Sine Transforms

N, the size of the Cosine and Sine transforms, must be greater than 0 and even.

11.2.1.4 Data Storage

The minimum size and the starting index for the input and output array for each type of Cosine and Sine transform is listed in Table 11–9.

Table 11–9 Size and Starting Index for _FCT and _FST

Transform	Type	Minimum Size	Starting Index
Cosine	I	N+1	0
Sine	I	N-1	1
Cosine	II	N	0
Sine	II	N	1

11.2.2 DXML's FCT and FST Functions

DXML provides the following set of Cosine and Sine transform functions covering the following options:

- Direction: forward or inverse
- Precision: single or double
- Type: I or II

11.2.2.1 Choosing Data Lengths

Since the Cosine and Sine transform functions are built on the FFT functions, the same considerations for choosing the data length for the Fourier transforms should be applied to the Cosine and Sine transforms. See Section 11.1.3.1 for information on choosing the data lengths for FFT.

11.2.2.2 Using the Internal Data Structures

Each time you perform an FCT or an FST operation, the software builds an internal data structure. The data structure provides a convenient way of storing attributes of the FCT or FST operation such as the data length, type of stride allowed, and pointers to virtual memory. If a program performs repeated FCTs or FSTs, the process is more efficient if the internal data structure is saved and re-used. For this reason, DXML provides two ways of performing FCT and FST transforms:

- One-step FCT or FST
  If your program performs only one or a few FCT or FST operations, use one subroutine to initialize, apply, and remove the internal data structure.
- Three-step FCT or FST
  If your program repeats the same FCT or FST operations, use the set of three subroutines:
  - The _INIT subroutine builds the internal data structures.
  - The _APPLY subroutine uses the internal data structures to compute the FCT or FST.
  - The _EXIT subroutine deallocates the virtual memory that was allocated by the _INIT subroutine.

The internal data structures are constant for a specified length of data. When you change the length of data, you must reinitialize the data structure. After you call the _INIT routine, you can call the _APPLY routine repeatedly as long as the data length and transform type remain the same.

Each three-step subroutine uses a structure argument to manipulate the internal data structure. You declare the data structure using the appropriate structure argument for the data format.

For FCT:

```
RECORD /DXML_S_FCT_STRUCTURE/
RECORD /DXML_D_FCT_STRUCTURE/
```

For FST:

```
RECORD /DXML_S_FST_STRUCTURE/
RECORD /DXML_D_FST_STRUCTURE/
```

You do not have to do anything else with this argument. For example, to perform a three-step, one-dimensional, single-precision Type I FST, declare the variable **FST_STRUCT** of type **RECORD /DXML_S_FST_STRUCTURE/**, as shown in the following code example:

```
 REAL*4 IN(N,100),OUT(N,100)
 INTEGER*4 SFST_INIT, SFST_APPLY, SFST_EXIT
 INTEGER*4 STATUS
 RECORD /DXML_S_FST_STRUCTURE/ FST_STRUCT
 CHARACTER*1 DIRECTION

 DIRECTION = 'F'
 STATUS = SFST_INIT(N,FST_STRUCT,1,.TRUE.)
 DO I=1,100
     STATUS = SFST_APPLY(DIRECTION,IN(1,I),OUT(1,I),
1                   FFT_STRUCT,1)
 ENDDO
 STATUS = SFST_EXIT(FFT_STRUCT)
```

11.2.2.3 Naming Conventions

A Cosine or a Sine transform subroutine has a name composed of character groups that tell you about the subroutine's operation. Table 11–10 shows the character groups used in the subroutine names and what they mean.

Table 11–10 Naming Conventions: Cosine and Sine Transform Functions

Character Group	Mnemonic	Meaning
First group	S	Single-precision real data
	D	Double-precision real data
Second group	FCT	Fast Cosine Transform
	FST	Fast Sine Transform
Third group	No mnemonic	One-step operation
	_INIT	Three-step operation: building of internal data structures

(continued on next page)

Table 11–10 (Cont.) Naming Conventions: Cosine and Sine Transform Functions

Character Group	Mnemonic	Meaning
	_APPLY	Three-step operation: perform FCT or FST
	_EXIT	Three-step operation: deallocation of virtual memory in internal data structure

For example, SFST_APPLY is the DXML function for calculating in single-precision arithmetic the one-dimensional FST of real data by applying the internal data structures that were calculated in the first step, SFST_INIT, of this three-step operation.

11.2.2.4 Summary of Cosine and Sine Transform Functions

Table 11–11 summarizes the Cosine and Sine transform functions that perform the entire transform in one step.

Table 11–11 Summary of One-Step Cosine and Sine Transform Functions

Name	Operation
SFCT	Calculates, in single-precision arithmetic, the fast Cosine transform of one-dimensional, real data.
DFCT	Calculates, in double-precision arithmetic, the fast Cosine transform of one-dimensional, real data.
SFST	Calculates, in single-precision arithmetic, the fast Sine transform of one-dimensional, real data.
DFST	Calculates, in double-precision arithmetic, the fast Sine transform of one-dimensional, real data.

Table 11–12 summarizes the three-step Cosine and Sine transform functions. Each function is either an initialization step, an application step, or an exit step.

Table 11–12 Summary of Three-Step Cosine and Sine Transform Functions

Name	Operation
SFCT_INIT	Calculates internal data structures.
SFCT_APPLY	Applies SFCT_INIT's internal data structure to calculate, in single-precision arithmetic, the fast Cosine transform of one-dimensional, real data.
SFCT_EXIT	Deallocates the virtual memory allocated by SFCT_INIT.
DFST_INIT	Calculates internal data structures.
DFST_APPLY	Applies DFST_INIT's internal data structure to calculate, in double-precision arithmetic, the fast Sine transform of one-dimensional, real data.
DFST_EXIT	Deallocates the virtual memory allocated by DFST_INIT.

11.3 Convolution and Correlation

Convolution and correlation are operations that complement the signal processing abilities of the Fourier transform. All number transforms (including the FFT) and most digital filters are convolution operations.

Convolution modifies a signal sequence by weighting the sequence with a function or an additional sequence of numbers. The convolution is used to obtain properties from a signal source (such as the nth derivative), to selectively enhance the signal source (in the case of a filter), or to domain transform the signal source (as in the case of a Fourier transform). Some type of convolution is the basis for most signal processing.

Correlation analyzes the similarity between two signals (as in the case of cross-correlation) or of a signal with itself (as in the case of auto-correlation).

11.3.1 Mathematical Definitions of Correlation and Convolution

Many definitions exist for convolution and correlation. DXML uses very specific definitions given in Sections 11.3.1.1, 11.3.1.2, and 11.3.1.3.

11.3.1.1 Definition of the Discrete Nonperiodic Convolution

The most common definition of a discrete nonperiodic convolution is given by:

$$h_j = \sum_{k=0}^{n_h-1} x_{(j-k)} y_k \qquad (11\text{–}28)$$

for $j = 0, 1, 2, \ldots, n_h - 1$ and $n_h = n_x + n_y - 1$.

Here, n_h is the total number of points to be output from the convolution subroutine, n_x is the number of points in the x array, and n_y is the number of points in the y array.

The y array is often called the filter array because nonrecursive digital filters are commonly made by using convolution of the x data array with special filter coefficients in the y array. For more information, consult the references given in Appendix A.

The definition of convolution given in Equation (11–28) is operational for infinitely long data sets in x and y, but because the data lengths are finite, in practice, the subscript $(j - k)$ will be out of range for the x array for certain values of j and k, and the subscript k will be out of range for the y array for certain values of k.

$$x_k = 0 \quad \text{when} \quad k < 0 \quad \text{or} \quad k > n_x - 1 \qquad (11\text{–}29)$$

$$y_k = 0 \quad \text{when} \quad k < 0 \quad \text{or} \quad k > n_y - 1 \qquad (11\text{–}30)$$

For the general case used in DXML, the definition of nonperiodic convolution can be rewritten as:

$$h_j = \sum_{k=\max\{0, j-n_x+1\}}^{\min\{n_y-1, j\}} x_{(j-k)} y_k \qquad (11\text{–}31)$$

11.3.1.2 Definition of the Discrete Nonperiodic Correlation

For correlation, a similar situation exists. For the idealized case of infinitely long data lengths, the definition of discrete nonperiodic correlation of two data sets, x and y, is given as:

$$h_j = \sum_{k=0}^{n_h-1} x_{(j+k)} y_k \tag{11–32}$$

for $1 - n_y \leq j \leq n_x - 1$

Here, n_h is the total number of points to be output from the correlation subroutine, n_x is the number of points in the x array, and n_y is the number of points in the y array.

Because of the finite lengths of the arrays, the relationships given by Equations (11–29) and (11–30) are used:

$$x_k = 0 \quad \text{when} \quad k < 0 \quad \text{or} \quad k > n_x - 1$$

$$y_k = 0 \quad \text{when} \quad k < 0 \quad \text{or} \quad k > n_y - 1$$

For this case, the definition of nonperiodic correlation can be rewritten as:

$$h_j = \sum_{k=\max\{0,-j\}}^{\min\{n_x-j,n_y\}-1} x_{(j+k)} y_k \tag{11–33}$$

Many variations on the definitions of convolution and correlation given in Equations (11–28) and (11–32) exist, but DXML uses the definitions given by these equations. Some definitions of convolution and correlation contain a normalization factor such as a $1/N$ term in front of the summation symbol where N is usually n_h as given in the DXML definitions. DXML subroutines do not use a normalization factor.

11.3.1.3 Periodic Convolution and Correlation

For periodic convolution, DXML uses the nonperiodic definition with a few differences. For $0 \leq j \leq n - 1$:

$$h_j = \sum_{k=0}^{n-1} x_{(j-k)} y_k \tag{11–34}$$

For periodic correlation, DXML uses the following definition. Again, for $0 \leq j \leq n-1$:

$$h_j = \sum_{k=0}^{n-1} x_{(j+k)} y_k \tag{11–35}$$

x and y are periodic with period n, that is, $x_{(j+n)} = x_j, y_{(j+n)} = y_j$. The data length of the output h array is equal to that of the x and y array.

If the subscript on either x or y is out of range, the value is that of the folded array. Folding is simply a modulus operation which implies periodicity.

As with nonperiodic convolution and correlation, no normalization factor is used in front of the summation symbol in the definition of periodic convolution and correlation.

For more information on periodic convolution and correlation, refer to the references given in Appendix A.

11.3.2 DXML's Convolution and Correlation Subroutines

DXML includes a wide variety of discrete convolution and correlation subroutines that support both periodic (circular) and nonperiodic (linear) convolution and correlation. Each subroutine is available for both real and complex data and for single-precision and double-precision arithmetic.

11.3.2.1 Using FFT Methods for Convolution and Correlation

DXML provides subroutines for calculating discrete convolutions and correlations by using a discrete summing technique. Other techniques that use fast Fourier transforms for calculating convolutions and correlations also exist, but they are not part of DXML.

When data lengths are large and there is a time-critical need for computing convolution and correlation functions, these FFT methods should be used. The data lengths must be large because the FFT methods introduce distortion near the edges of the data, unless there is true periodicity in the data, the data is well behaved near the ends, and many periods are sampled.

For more information on performing convolution and correlation with FFTs, refer to the references given in Appendix A.

11.3.2.2 Naming Conventions

A convolution or correlation subroutine has a name composed of character groups that tell you about the subroutine's operation. Table 11–13 shows the character groups used in the subroutine names and what they mean.

Table 11–13 Naming Conventions: Convolution and Correlation Subroutines

Character Group	Mnemonic	Meaning
First group	S	Single-precision real data
	D	Double-precision real data
	C	Single-precision complex data
	Z	Double-precision complex data
Second group	CONV	Convolution subroutine
	CORR	Correlation subroutine
Third group	_NONPERIODIC	Nonperiodic operation
	_PERIODIC	Periodic operation
	_NONPERIODIC_EXT	Nonperiodic operation with extension
	_PERIODIC_EXT	Periodic operation with extension

For example, SCORR_PERIODIC is the DXML subroutine for calculating in single-precision arithmetic the periodic correlation of two real arrays.

11.3.2.3 Summary of Convolution and Correlation Subroutines

Tables 11–14 and 11–15 summarize the convolution and correlation subroutines.

Table 11–14 Summary of Convolution Subroutines

Subroutine Name	Operation
SCONV_NONPERIODIC	Calculates, in single-precision arithmetic, the nonperiodic convolution of two real arrays.
DCONV_NONPERIODIC	Calculates, in double-precision arithmetic, the nonperiodic convolution of two real arrays.
CCONV_NONPERIODIC	Calculates, in single-precision arithmetic, the nonperiodic convolution of two complex arrays.
ZCONV_NONPERIODIC	Calculates, in double-precision arithmetic, the nonperiodic convolution of two complex arrays.
SCONV_PERIODIC	Calculates, in single-precision arithmetic, the periodic convolution of two real arrays.
DCONV_PERIODIC	Calculates, in double-precision arithmetic, the periodic convolution of two real arrays.
CCONV_PERIODIC	Calculates, in single-precision arithmetic, the periodic convolution of two complex arrays.
ZCONV_PERIODIC	Calculates, in double-precision arithmetic, the periodic convolution of two complex arrays.
SCONV_NONPERIODIC_EXT	Calculates, in single-precision arithmetic, the nonperiodic convolution of two real arrays.
DCONV_NONPERIODIC_EXT	Calculates, in double-precision arithmetic, the nonperiodic convolution of two real arrays.
CCONV_NONPERIODIC_EXT	Calculates, in single-precision arithmetic, the nonperiodic convolution of two complex arrays.
ZCONV_NONPERIODIC_EXT	Calculates, in double-precision arithmetic, the nonperiodic convolution of two complex arrays.
SCONV_PERIODIC_EXT	Calculates, in single-precision arithmetic, the periodic convolution of two real arrays.
DCONV_PERIODIC_EXT	Calculates, in double-precision arithmetic, the periodic convolution of two real arrays.
CCONV_PERIODIC_EXT	Calculates, in single-precision arithmetic, the periodic convolution of two complex arrays.
ZCONV_PERIODIC_EXT	Calculates, in double-precision arithmetic, the periodic convolution of two complex arrays.

Table 11–15 Summary of Correlation Subroutines

Subroutine Name	Operation
SCORR_NONPERIODIC	Calculates, in single-precision arithmetic, the nonperiodic correlation of two real arrays.
DCORR_NONPERIODIC	Calculates, in double-precision arithmetic, the nonperiodic correlation of two real arrays.
CCORR_NONPERIODIC	Calculates, in single-precision arithmetic, the nonperiodic correlation of two complex arrays.

(continued on next page)

Table 11–15 (Cont.) Summary of Correlation Subroutines

Subroutine Name	Operation
ZCORR_NONPERIODIC	Calculates, in double-precision arithmetic, the nonperiodic correlation of two complex arrays.
SCORR_PERIODIC	Calculates, in single-precision arithmetic, the periodic correlation of two real arrays.
DCORR_PERIODIC	Calculates, in double-precision arithmetic, the periodic correlation of two real arrays.
CCORR_PERIODIC	Calculates, in single-precision arithmetic, the periodic correlation of two complex arrays.
ZCORR_PERIODIC	Calculates, in double-precision arithmetic, the periodic correlation of two complex arrays.
SCORR_NONPERIODIC_EXT	Calculates, in single-precision arithmetic, the nonperiodic correlation of two real arrays.
DCORR_NONPERIODIC_EXT	Calculates, in double-precision arithmetic, the nonperiodic correlation of two real arrays.
CCORR_NONPERIODIC_EXT	Calculates, in single-precision arithmetic, the nonperiodic correlation of two complex arrays.
ZCORR_NONPERIODIC_EXT	Calculates, in double-precision arithmetic, the nonperiodic correlation of two complex arrays.
SCORR_PERIODIC_EXT	Calculates, in single-precision arithmetic, the periodic correlation of two real arrays.
DCORR_PERIODIC_EXT	Calculates, in double-precision arithmetic, the periodic correlation of two real arrays.
CCORR_PERIODIC_EXT	Calculates, in single-precision arithmetic, the periodic correlation of two complex arrays.
ZCORR_PERIODIC_EXT	Calculates, in double-precision arithmetic, the periodic correlation of two complex arrays.

11.4 Digital Filtering

Digital filters are subroutines that eliminate certain frequency components from a signal which has been corrupted with unwanted noise. DXML provides a nonrecursive filter (also known as a finite impulse response filter) which can be used in four ways:

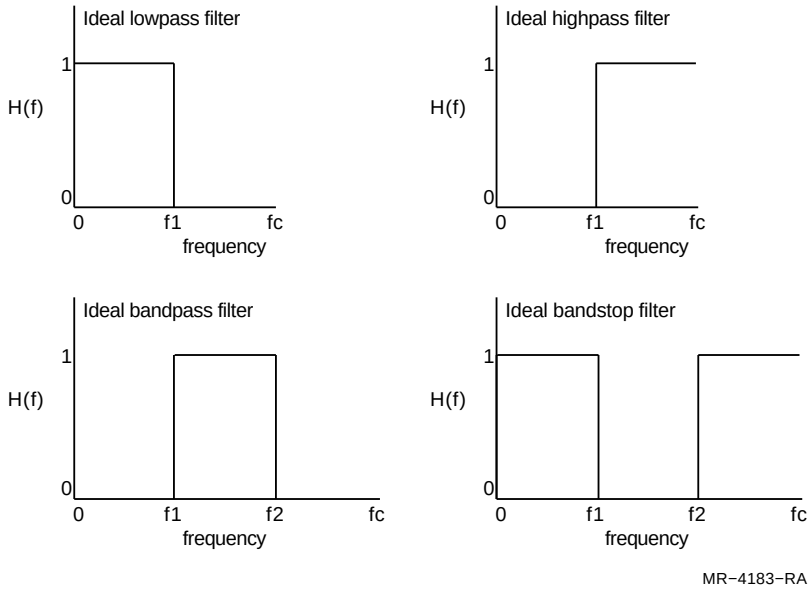
- Lowpass filter
  Eliminates frequency components above one value.
- Highpass filter
  Eliminates frequency components below one value.
- Bandpass filter
  Eliminates frequency components except those within a certain range.
- Bandstop (notch) filter
  Eliminates frequency components within a certain range.

The DXML nonrecursive filter is an adaptation of the $I_0 - sinh$ filter originally proposed by Kaiser. See *Digital Filters* by R.W. Hamming for a complete mathematical description of this filter.

11.4.1 Mathematical Definition of the Nonrecursive Filter

The transfer function of a nonrecursive digital filter, denoted as H(f), determines the range of frequencies that are eliminated by a filter. Putting a sinusoidal function of frequency f into the filter results in the output being the same as the sinusoid, except that its amplitude is multiplied by H(f). The transfer function H(f) can take on any of the forms shown in Figure 11–1.

Figure 11–1 Digital Filter Transfer Function Forms



In Figure 11–1 fc refers to the Nyquist frequency $1/(2\Delta t)$, Δt is the time between data samples, and f1 and f2 refer to the frequency values where filtering is to be applied.

These ideal filters represent an infinitely sharp band; in practice, as shown in Figure 11–2, the vertical lines are skewed.

The filtering discussed here pertains only to nonrecursive filters of the form:

$$y_n = A_0 x_n + \sum_{k=1}^{nterms} A_k \left(x_{(n+k)} + x_{(n-k)} \right) \qquad (11\text{–}36)$$

where y_n is the dependent variable synthesized by the use of previous dependent values x, A_k are the filter-dependent coefficients, and $nterms$ is the number of filter coefficients with A_0 not included.

11.4.2 Controlling Filter Type

In the filter subroutines SFILTER_NONREC and SFILTER_INIT_NONREC, you use the **flow** and **fhigh** arguments to control the type of filtering. Table 11–16 shows the **flow** and **fhigh** argument values associated with particular filtering types.

Table 11–16 Controlling Filtering Type

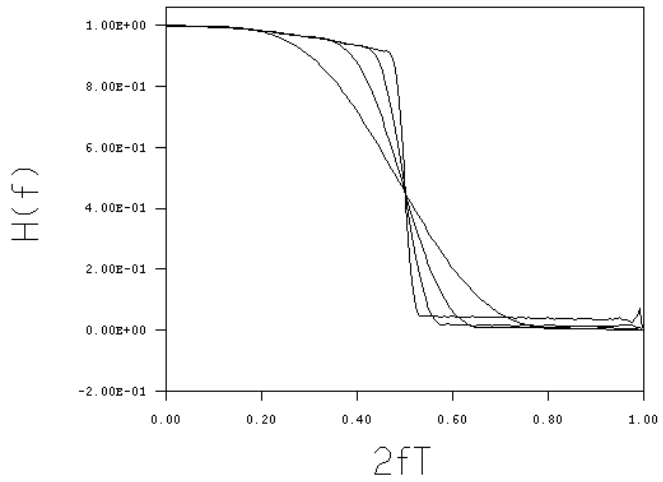
For a filter type of:	Set flow to:	Set fhigh to:
No filtering	0	1
Lowpass filter	0	0 < fhigh < 1
Highpass filter	0 < flow < 1	1
Bandpass filter	0 < flow < fhigh	flow < fhigh < 1
Bandstop filter	fhigh < flow < 1	0 < fhigh < flow

11.4.3 Controlling Filter Sharpness and Smoothness

In the filter subroutines SFILTER_NONREC and SFILTER_INIT_NONREC, you use the **nterms** and **wiggles** arguments to control the sharpness and smoothness of the filter.

Figure 11–2 shows the transfer function of a lowpass nonrecursive filter where **wiggles** = 50.0, **flow** = 0.0, and **fhigh** = 0.5, for **nterms** = 5, 10, 20, and 50.

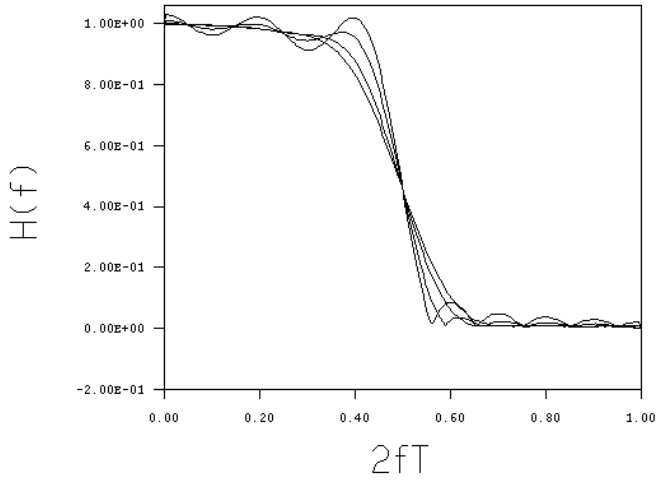
Figure 11–2 Lowpass Nonrecursive Filter for Varying Nterms



The **nterms** argument determines the sharpness of the filter. When **nterms** is increased, the filter cutoff is sharper. Though it seems that using the largest possible value for **nterms** results in a sharper filter, 2(**nterms**) number of data points from the original set are not filtered. If the data set is large, the loss of data caused by the filtering process is inconsequential. However, the loss of data can be detrimental to smaller data sets. In addition, the computational time increases proportionally to the value of **nterms**. Try to make the value of **nterms** as large as possible without losing too many end points or making the computational time too long.

Figure 11–3 shows the transfer function of a lowpass nonrecursive filter where **flow** = 0.0, **fhigh** = 0.5, and **nterms** = 10, for **wiggles** = 0, 30, 50, and 70.

Figure 11–3 Lowpass Nonrecursive Filter for Varying Wiggles



The **wiggles** argument controls the smoothness of the filter. The wiggles, which are related to the size of Gibbs Phenomenon oscillations, are most prominent when the value of the **wiggles** argument is 0.0. As the value of **wiggles** is increased, the oscillations become less noticeable; however, the sharpness of the filter decreases. A good compromise is to set **wiggles** = 50.0.

The size of the oscillations (in -dB units) is related to the value of the **wiggles** argument:

$$|\text{Magnitude of Oscillations}| = 10^{(-\text{wiggles}/20.0)} \qquad (11\text{–}37)$$

11.4.4 DXML's Digital Filter Subroutines

DXML includes three subroutines for nonrecursive filtering. These subroutines are of two types, each of which does the filter operation in a different way:

- Completes the filter operation in one step.
- Completes the filter operation in two steps.

  One subroutine initializes a working array; a second subroutine uses that working array for repeated operations so that the working array need only be calculated once.

The DXML filtering subroutines have single-precision capability; they do not have double-precision capability.

11.4.4.1 Naming Conventions

A filter subroutine has a name composed of character groups that tell you about the subroutine's operation. Table 11–17 shows the character groups used in the subroutine names and what they mean.

Table 11–17 Naming Conventions: Digital Filter Subroutines

Character Group	Mnemonic	Meaning
First group	S	Single-precision real data
Second group	FILTER	Filtering subroutine
Third group	No mnemonic _INIT _APPLY	One-step filter Two-step filter initialization Two-step filter application
Fourth group	_NONREC	Nonrecursive filter

11.4.4.2 Summary of Digital Filter Subroutines

Table 11–18 summarizes the filter subroutines.

Table 11–18 Summary of Digital Filter Subroutines

Subroutine Name	Operation
SFILTER_NONREC	Performs the filter operation in one step
SFILTER_INIT_NONREC	Initializes a working array
SFILTER_APPLY_NONREC	Uses the initialized working array for repeated filtering operations

11.5 Error Handling

The signal processing functions report success or error, using a **status** function value. To include the error code and data structure definitions in a signal processing application program, you must put one of the following lines at the beginning of your program.

- For Fortran:

```
INCLUDE '/usr/include/DXMLDEF.FOR'
```

- For C:

```
#include "/usr/include/dxmldef.h"
```

A callable routine, `dxml_sig_error`, is provided for the signal processing routines. Example 11–1 shows how to use this routine.

Example 11–1 Example of Error Routine for Signal Processing

```
PROGRAM EXAMPLE
INCLUDE '/usr/include/DXMLDEF.FOR'

COMPLEX*8 A(8),OUTA(8),B(8),OUTB(8)
COMPLEX*8 DIFF(8),W,G0,G1,AA
REAL*4 TWOPI,T0
INTEGER*4 T,K,I,NT,STATUS,N

RECORD /DXML_D_FFT_STRUCTURE/ FFT_STRUCT

N = 8
TWOPI=6.283185307
T0=TWOPI/FLOAT(8)
W=CMPLX(COS(T0),(-1.0)*SIN(T0))
AA=(0.9,0.3)

C Compute the raw data for the transform
  B(1)=(1.0,0.0)
  DO 1 T=2,8
1 B(T)=AA**(T-1)

C Calculate the analytical transform of the function
  NT=8
  G0=(1.0,0.0)-AA**NT

  DO 5 I=1,NT
  G1=(1.0,0.0)-AA*(W**(I-1))
5 OUTA(I)=G0/G1

  TYPE 100
100  FORMAT(//,3X,'FOR REAL FORWARD FFT WITH 8 POINTS ',
     1  //,3X,'POINT',T11,'ANALYTICAL RESULT',T37,'COMPUTED RESULT',
     2  T65,'DIFF.',/)

C Compute the transform of the function using DXML routines
    ISTAT = CFFT_INIT (0,FFT_STRUCT,.TRUE.)
    IF (ISTAT.NE.DXML_SUCCESS()) CALL DXML_SIG_ERROR (ISTAT)

    ISTAT = CFFT_APPLY ('C','C','F',B,OUTB,FFT_STRUCT,1)
    IF (ISTAT.NE.DXML_SUCCESS()) CALL DXML_SIG_ERROR (ISTAT)

    ISTAT = CFFT_EXIT (FFT_STRUCT)
    IF (ISTAT.NE.DXML_SUCCESS()) CALL DXML_SIG_ERROR (ISTAT)

C Calculate the difference between the computed and analytical solution
C to the transform
     DO 10 I=1,NT
10   DIFF(I)=OUTB(I)-OUTA(I)

C Print out the results
     DO 20 I=1,NT
     TYPE 130,I,OUTA(I),OUTB(I),DIFF(I)
130  FORMAT(2X,I2,2X,3(2(1X,1PE11.4)))
20   CONTINUE

     STOP
     END
```

Table 11–19 shows the status functions, the value returned, an explanation of the error associated with each value, and the appropriate user action suggested to recover from each error condition.

Table 11–19 DXML Status Functions

Function	Value	Description	User Action
DXML_SUCCESS	0	Successful execution of SIG routines	No action required.
DXML_MAND_ARG	1	Mandatory argument is missing	Check the argument list.
DXML_ILL_TEMP_ARRAY	2	temp_array is corrupted	Check code and correct the error.
DXML_IN_VERSION_SKEW	3	temp_array is from old version	Use the same version of DXML to create and use the temporary array.
DXML_ILL_N_IS_ODD	4	A value is odd	Change **n**, **n1**, or **n2** to an even value.
DXML_ILL_WIGGLES	5	Value is out of range	Change **wiggles** to a value that is in range.
DXML_ILL_FLOW	6	flow is equal to fhigh	Provide different values for the **flow** and **fhigh** arguments.
DXML_ILL_F_RANGE	7	flow or fhigh is out of range	Check values of **flow** and **fhigh** arguments and replace with a value between 0.0 and 1.0, inclusive.
DXML_ILL_N_RANGE	8	n is out of range	Check description of **n** for the allowed length.
DXML_ILL_N_NONREC	9	n is less than (2*nterms+1)	Either replace the current value of **n** with a value greater than the value of (2***nterms**)+1 or make the value of **nterms** smaller.
DXML_ILL_NTERMS	10	nterms is out of range	Replace the current value of the **nterms** argument with a value between 2 and 500, inclusive.
DXML_ILL_LDA	11	lda cannot be less than n	Change **lda** to a value greater than or equal to the number of data points in the row direction.
DXML_INS_RES	12	Virtual memory or pagefile quota is not set high enough for data length	Either change the data length to a number that is not prime or increase the allocated values of the pagefile quota and virtual memory.
DXML_BAD_STRIDE	13	Stride is incorrect	Change the value of **stride** to an integer greater than or equal to 1.
DXML_DIRECTION_NOT_MATCH	14	APPLY/INIT directions different	Change the value of **direction** in either APPLY or INIT to match.
DXML_BAD_DIRECTION_STRING	15	Direction string is incorrect	Change the first letter in value for **direction** to 'F' or 'B'.

(continued on next page)

Table 11–19 (Cont.) DXML Status Functions

Function	Value	Description	User Action
DXML_BAD_FORMAT_STRING	16	Format string is incorrect	Change the first letter in format string to 'R' or 'C'.
DXML_OPTION_NOT_ SUPPORTED	17	I/O combination not supported	Refer to description of subprogram for supported combinations.
DXML_BAD_FORMAT_ DIRECTION	18	Format/direction combination not supported	Refer to description of subprogram for supported combinations.

Fast Fourier Transform Subprograms

This section provides descriptions of the routines for fast transforms. The four versions of each routine are titled by the name using a leading underscore character. The section is ordered in the following way:

- By the type of FFT:

  one-dimensional
  two-dimensional
  three-dimensional
  group

- By function within each type:

  one-step procedure
  initialization
  application
  exit

_FFT
Fast Fourier Transform in One Dimension (Serial and Parallel Versions)

Format

Real transform:

status = {S,D}FFT (input_format, output_format, direction, in, out, n, stride)

Complex transform in complex data format:

status = {C,Z}FFT (input_format, output_format, direction, in, out, n, stride)

Complex transform in real data format:

status = {C,Z}FFT (input_format, output_format, direction, in_real, in_imag, out_real, out_imag, n, stride)

Arguments

input_format, output_format
character*(*)
Identifies the data type of the input and the format to be used to store the data, regardless of the data type. For example, a complex sequence can be stored in real format.

The character 'R' specifies the format as real; the character 'C' specifies the format as complex. As convenient, use either uppercase or lowercase characters, and either spell out or abbreviate the word.

The following table shows the valid values:

Subprogram	Input Format	Output Format	Direction
{S,D}	'R'	'C'	'F'
	'C'	'R'	'B'
	'R'	'R'	'F' or 'B'
{C,Z}	'R'	'R'	'F' or 'B'
	'C'	'C'	'F' or 'B'

For complex transforms, the type of data determines what other arguments are needed. When both the input and output data are real, the complex functions store the data as separate arrays for imaginary and real data so additional arguments are needed.

direction
character*(*)
Specifies the operation as either the forward or inverse transform. Use 'F' or 'f' to specify the forward transform. Use 'B' or 'b' to specify the inverse transform.

in, out
real*4 | real*8 | complex*8 | complex*16
Both the arguments are one-dimensional arrays. The input and output arrays can be the same array. The IN array contains the data to be transformed. The OUT array contains the transformed data.

in_real, out_real, in_imag, out_imag
real*4 | real*8
Use these arguments when performing a complex transform on real data format and storing the result in a real data format.

n
integer*4
Specifies the number of values to be transformed, that is, the length of the array to be transformed, and therefore the required size of the resulting array; $n > 0$.

Subprogram	Input Format	Output Format	Minimum Size
{S,D}	'R'	'C'	n+2 (n must be even)
	'C'	'R'	n+2 (n must be even)
	'R'	'R'	n (n must be even)
{C,Z}	'R'	'R'	n
	'C'	'C'	n

stride
integer*4
Specifies the distance between consecutive elements in the input and output arrays. The distance must be at least 1.

Description

The _FFT functions compute the fast Fourier transform of one-dimensional data in one step. The SFFT and DFFT functions perform the forward Fourier transform of a real sequence and store the result in either real or complex data format. These functions also perform the inverse Fourier transform of a complex sequence into a real sequence.

The CFFT and ZFFT functions perform Fourier transforms on a complex sequence. However, the argument list is different, depending on the data format in which the output data is stored. See Section 11.1.3.2 for an explanation of real and complex data format.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

Return Values

0	DXML_SUCCESS()
4 (real transforms only)	DXML_ILL_N_IS_ODD()
8	DXML_ILL_N_RANGE()
12	DXML_INS_RES()
13	DXML_BAD_STRIDE()
15	DXML_BAD_DIRECTION_STRING()
16	DXML_BAD_FORMAT_STRING()
18 (real transforms only)	DXML_BAD_FORMAT_FOR_DIRECTION()

Example

```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N, STATUS
COMPLEX*16 A(1024), B(1024)
REAL*8 C_REAL(1024),C_IMAG(1024),D_REAL(1024),D_IMAG(1024)
REAL*8 E(1026),F(1026)
REAL*8 G(1024),H(1024)
N = 1024
STATUS = ZFFT('C','C','F',A,B,N,1)
STATUS = ZFFT('R','R','F',C_REAL,C_IMAG,D_REAL,D_IMAG,N,1)
STATUS = DFFT('R','C','F',E,F,N,1)
STATUS = DFFT('C','R','B',F,E,N,1)
STATUS = DFFT('R','R','F',G,H,N,1)
```

This Fortran code computes the following:

- Forward Fourier transform of the complex sequence A to the complex sequence B.
- Forward Fourier transform of the complex sequence C to the complex sequence D. The data C and D are each stored as two real arrays.
- Forward Fourier transform of the real sequence E to the complex sequence F. The data F is stored in complex format.
- Backward Fourier transform of the complex sequence F to the real sequence E.
- Backward Fourier transform of the real sequence G to the complex sequence H stored in real format.

_FFT_INIT
Initialization Step for Fast Fourier Transform in One Dimension (Serial and Parallel Versions)

Format

status = {S,D,C,Z}FFT_INIT (n, fft_struct, stride_1_flag)

Arguments

n
integer*4
Specifies the number of values to be transformed, that is, the length of the array to be transformed; $n > 0$. For real operations, **n** must be even.

fft_struct
record /dxml_s_fft_structure/ for single-precision real operations
record /dxml_d_fft_structure/ for double-precision real operations
record /dxml_c_fft_structure/ for single-precision complex operations
record /dxml_z_fft_structure/ for double-precision complex operations
You must include this argument but it needs no additional definitions. The argument is declared in the program before this function. See Section 11.1.3.3 for more information.

stride_1_flag
logical
Specifies the allowed distance between consecutive elements in the input and output arrays:

TRUE: Stride must be 1.
FALSE: Stride is at least 1.

Description

The _FFT_INIT functions build internal data structures needed to compute fast Fourier transforms of one-dimensional data. These functions are the first step in a three-step procedure. They create the internal data structures, using attributes defined in the file DXMLDEF.FOR.

Use the initialization function that is appropriate for the data format. Then use the corresponding application and exit functions to complete the transform. For example, use SFFT_INIT for the internal data structures used by SFFT_APPLY and end with the SFFT_EXIT function.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

Return Values

0	DXML_SUCCESS()
4(real transforms only)	DXML_ILL_N_IS_ODD()
8	DXML_ILL_N_RANGE()
12	DXML_INS_RES()

_FFT_APPLY
Application Step for Fast Fourier Transform in One Dimension (Serial and Parallel Versions)

Format

Real transform:

status = {S,D}FFT_APPLY (input_format, output_format, direction, in, out, fft_struct, stride)

Complex transform in complex data format:

status = {C,Z}FFT_APPLY (input_format, output_format, direction, in, out, fft_struct, stride)

Complex transform in real data format:

status = {C,Z}FFT_APPLY (input_format, output_format, direction, in_real, in_imag, out_real, out_imag, fft_struct, stride)

Arguments

input_format, output_format
character*(*)
Identifies the data type of the input and the format to be used to store the data, regardless of the data type. For example, a complex sequence can be stored in real format.

The character 'R' specifies the format as real; the character 'C' specifies the format as complex. As convenient, use either uppercase or lowercase characters, and either spell out or abbreviate the word.

The following table shows the valid values:

Subprogram	Input Format	Output Format	Direction
{S,D}	'R'	'C'	'F'
	'C'	'R'	'B'
	'R'	'R'	'F' or 'B'
{C,Z}	'R'	'R'	'F' or 'B'
	'C'	'C'	'F' or 'B'

For complex transforms, the type of data determines what other arguments are needed. When both the input and output data are real, the complex functions store the data as separate arrays for imaginary and real data so additional arguments are needed. See Section 11.1.3.2 for an explanation of real and complex data format.

direction
character*(*)
Specifies the operation as either the forward or inverse transform. Use 'F' or 'f' to specify the forward transform. Use 'B' or 'b' to specify the inverse transform.

in, out
real*4 | real*8 | complex*8 | complex*16
Both the arguments are one-dimensional arrays. The input and output arrays can be the same array. The IN array contains the data to be transformed. The OUT

array contains the transformed data. The size of the output array is determined by the value of **n** provided to the _INIT function.

Subprogram	Input Format	Output Format	Minimum Size
{S,D}	'R'	'C'	n+2 (n must be even)
	'C'	'R'	n+2 (n must be even)
	'R'	'R'	n (n must be even)
{C,Z}	'R'	'R'	n
	'C'	'C'	n

in_real, out_real, in_imag, out_imag
real*4 | real*8
Use these arguments to perform a complex transform on real data format and storing the result in a real data format.

fft_struct
record /dxml_s_fft_structure/ for single-precision real operations
record /dxml_d_fft_structure/ for double-precision real operations
record /dxml_c_fft_structure/ for single-precision complex operations
record /dxml_z_fft_structure/ for double-precision complex operations
The argument refers to the structure created by the _INIT function.

stride
integer*4
Specifies the distance between consecutive elements in the input and output arrays, depending on the value of **stride_1_flag** provided in the _INIT function.

Description

The _FFT_APPLY routine performs the fast Fourier transform of one-dimensional data, in either the forward or backward direction. These routines are the second step of a three-step procedure. The _FFT_APPLY routine computes the fast forward or inverse Fourier transform, using the internal data structures created by the _FFT_INIT subroutine.

Use the _APPLY routines with their corresponding _INIT and _EXIT routines. For example, use SFFT_APPLY after the SFFT_INIT and end with the SFFT_ EXIT routine.

The SFFT_APPLY and DFFT_APPLY functions perform the Fourier transform of a real sequence (forward) or a complex sequence (inverse) into real sequence.

The CFFT_APPLY and ZFFT_APPLY functions perform Fourier transforms of a complex sequence into a complex sequence, storing the output in either real or complex data format. However, the argument list depends on the data format of the output. See Section 11.1.3.2 for an explanation of real and complex data format.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

Return Values

0	DXML_SUCCESS()
12	DXML_INS_RES()
13	DXML_BAD_STRIDE()
15	DXML_BAD_DIRECTION_STRING()
16	DXML_BAD_FORMAT_STRING()
18 (real transform only)	DXML_BAD_FORMAT_FOR_DIRECTION()

Examples

1.
```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N, STATUS
REAL*8 A(1026), B(1026)
RECORD /DXML_D_FFT_STRUCTURE/ FFT_STRUCT
N = 1024
STATUS = DFFT_INIT(N,FFT_STRUCT,.TRUE.)
STATUS = DFFT_APPLY('R','C','F',A,B,FFT_STRUCT,1)
STATUS = DFFT_EXIT(FFT_STRUCT)
```

This Fortran code computes the forward, real FFT of a vector, *a*, with length of 1024. The result of the transform is stored in *b* in complex form. The length of *b* is 1026 to hold 513 complex values.

2.
```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N, STATUS
COMPLEX*8 A(1024), B(1024)
RECORD /DXML_C_FFT_STRUCTURE/ FFT_STRUCT
N = 1024
STATUS = CFFT_INIT(N,FFT_STRUCT,.TRUE.)
STATUS = CFFT_APPLY('C','C','F',A,B,FFT_STRUCT,1)
STATUS = CFFT_EXIT(FFT_STRUCT)
```

This Fortran code computes the forward, complex FFT of a vector *a*, with length of 1024. The result of the transform is stored in *b* in complex form.

_FFT_EXIT
Final Step for Fast Fourier Transform in One Dimension (Serial and Parallel Versions)

Format

status = {S,D,C,Z}FFT_EXIT (fft_struct)

Arguments

fft_struct
record /dxml_s_fft_structure/ for single-precision real operations
record /dxml_d_fft_structure/ for double-precision real operations
record /dxml_c_fft_structure/ for single-precision complex operations
record /dxml_z_fft_structure/ for double-precision complex operations
This argument must be included but it is not necessary to modify it in any way. It refers to the data structure that was specified by the initialization step. See Section 11.1.3.3 for more information on the data structure.

Description

The _FFT_EXIT functions remove the internal data structures created in the _FFT_INIT functions. These functions are the final step in a three-step procedure. They release the virtual memory that was allocated by the _FFT_INIT functions.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

Return Values

0	DXML_SUCCESS()
12	DXML_INS_RES()

_FFT_2D
Fast Fourier Transform in Two Dimensions (Serial and Parallel Versions)

Format

Real transform:

status = {S,D}FFT_2D (input_format, output_format, direction, in, out, ni, nj, lda, ni_stride, nj_stride)

Complex transforms in complex format:

status = {C,Z}FFT_2D (input_format, output_format, direction, in, out, ni, nj, lda, ni_stride, nj_stride)

Complex transform in real data format:

status = {C,Z}FFT_2D (input_format, output_format, direction, in_real, out_real, in_imag, out_imag, ni, nj, lda, ni_stride, nj_stride)

Arguments

input_format, output_format
character*(*)
Identifies the data type of the input and the format to be used to store the data, regardless of the data type. For example, a complex sequence can be stored in real format.

The character 'R' specifies the format as real; the character 'C' specifies the format as complex. As convenient, use either uppercase or lowercase characters, and either spell out or abbreviate the word.

The following table shows the valid values:

Subprogram	Input Format	Output Format	Direction
{S,D}	'R'	'C'	'F'
	'C'	'R'	'B'
	'R'	'R'	'F' or 'B'
{C,Z}	'R'	'R'	'F' or 'B'
	'C'	'C'	'F' or 'B'

For complex data, the type of data determines what other arguments are needed. When both the input and output data are real, the complex routines store the data as separate arrays for imaginary and real data so additional arguments are needed.

direction
character*(*)
Specifies the operation as either the forward or inverse transform. Use 'F' or 'f' to specify the forward transform. Use 'B' or 'b' to specify the inverse transform.

in, out
real*4 | real*8 | complex*8 | complex*16
Both the arguments are two-dimensional arrays. The input and output arrays can be the same array. The IN array contains the data to be transformed. The OUT array contains the transformed data.

in_real, out_real, in_imag, out_imag
real*4 | real*8
Use these arguments when performing a complex transform on real data format and storing the result in a real data format.

ni, nj
integer*4
Specifies the size of the first and second dimension of data in the input array; $ni > 0$, $nj > 0$. For SFFT_2D and DFFT_2D, **ni** must be even.

lda
integer*4
Specifies the number of rows in the IN and OUT arrays; $lda \geq ni$. For {S,D} routines, $lda \geq ni + 2$ when the input format is 'R' and the output format is 'C' or the input format is 'C' and the output format is 'R'.

ni_stride, nj_stride
integer*4
Specifies the distance between consecutive elements in a column and row in the IN and OUT arrays; *ni_stride* ≥ 1, *nj_stride* ≥ 1.

Description

The _FFT_2D routines compute the fast forward or inverse Fourier transform of two-dimensional data in one step. The SFFT_2D and DFFT_2D functions perform the forward Fourier transform of a real sequence and store the result in either real or complex data format. These functions also perform the inverse Fourier transform of a complex sequence into a real sequence.

The CFFT_2D and ZFFT_2D functions perform Fourier transforms on a complex sequence. However, the argument list is different, depending on the data format in which the output data is stored. See Section 11.1.3.2 for an explanation of real and complex data format.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

Return Values

0	DXML_SUCCESS()
4 (with real transforms only)	DXML_ILL_N_IS_ODD()
8	DXML_ILL_N_RANGE()
11	DXML_ILL_LDA()
12	DXML_INS_RES()
13	DXML_BAD_STRIDE()
15	DXML_BAD_DIRECTION_STRING()
16	DXML_BAD_FORMAT_STRING()
18	DXML_BAD_FORMAT_FOR_DIRECTION()

Examples

1.
```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N_I, N_J, STATUS, LDA
REAL*8 A(1026,512), B(1026,513)
N_I = 1024
N_J = 512
LDA = 1026
STATUS = DFFT_2D('R','C','F',A,B,N_I,N_J,LDA,1,1)
```

This Fortran code computes the forward, two-dimensional, real FFT of a 1024x512 matrix *A*. The result of the transform is stored in *B* in complex form. The leading dimension of *B* is 1026 in order to hold the extra complex values (see section on data storage). The input matrix *A* also requires a leading dimension of at least 1026.

2.
```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N_I, N_J, STATUS, LDA
COMPLEX*8 A(1024,512), B(1024,512)
N_I = 1024
N_J = 512
LDA = 1024
STATUS = CFFT_2D('C','C','F',A,B,N_I,N_J,LDA,1,1)
```

This Fortran code computes the forward, two-dimensional, complex FFT of a matrix *A*, of dimension 1024 by 512. The result of the transform is stored in *B* in complex form.

_FFT_INIT_2D
Initialization Step for Fast Fourier Transform in Two Dimensions (Serial and Parallel Versions)

Format

status = {S,D,C,Z}FFT_INIT_2D (ni, nj, fft_struct, ni_stride_1_flag)

Arguments

ni, nj
integer*4
Specifies the size of the first and second dimension of data in the input array; $ni > 0$, $nj > 0$. For SFFT_INIT_2D and DFFT_INIT_2D, **ni** must be even.

fft_struct
record /dxml_s_fft_structure_2d/ for single-precision real operations
record /dxml_d_fft_structure_2d/ for double-precision real operations
record /dxml_c_fft_structure_2d/ for single-precision complex operations
record /dxml_z_fft_structure_2d/ for double-precision complex operations
This argument must be included but needs no additional definitions. The argument is declared in the program before this routine. See Section 11.1.3.3 for more information.

ni_stride_1_flag
logical
Specifies whether to allow a stride of more than 1 between elements in the row:

TRUE: Stride must be 1.
FALSE: Stride is at least 1.

Description

The _FFT_INIT_2D functions build internal data structures needed to compute fast Fourier transforms of two-dimensional data. These routines are the first step in a three-step procedure. They create the internal data structures, using attributes defined in the file DXMLDEF.FOR.

Use the initialization routine that is appropriate for the data format. Then, use the corresponding application and exit steps to complete the procedure. For example, use the SFFT_2D_INIT routine with the SFFT_2D_APPLY and SFFT_2D_EXIT routine.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

Return Values

0	DXML_SUCCESS()
4 (real transform only)	DXML_ILL_N_IS_ODD()
8	DXML_ILL_N_RANGE()
12	DXML_INS_RES()

_FFT_APPLY_2D
Application Step for Fast Fourier Transform in Two Dimensions (Serial and Parallel Versions)

Format

Real transform:

status = {S,D}FFT_APPLY_2D (input_format, output_format, direction, in, out, lda, fft_struct, ni_stride, nj_stride)

Complex transform in complex data format:

status = {C,Z}FFT_APPLY_2D (input_format, output_format, direction, in, out, lda, fft_struct, ni_stride, nj_stride)

Complex transform in real data format:

status = {C,Z}FFT_APPLY_2D (input_format, output_format, direction, in_real, in_imag, out_real, out_imag, lda, fft_struct, ni_stride, nj_stride)

Arguments

input_format, output_format
character*(*)
Identifies the data type of the input and the format to be used to store the data, regardless of the data type. For example, a complex sequence can be stored in real format.

The character 'R' specifies the format as real; the character 'C' specifies the format as complex. As convenient, use either uppercase or lowercase characters, and either spell out or abbreviate the word.

The following table shows the valid values:

Subprogram	Input Format	Output Format	Direction
{S,D}	'R'	'C'	'F'
	'C'	'R'	'B'
	'R'	'R'	'F' or 'B'
{C,Z}	'R'	'R'	'F' or 'B'
	'C'	'C'	'F' or 'B'

For complex data, the type of data determines what other arguments are needed. When both the input and output data are real, the complex routines store the data as separate arrays for imaginary and real data so additional arguments are needed.

direction
character*(*)
Specifies the operation as either the forward or inverse transform. Use 'F' or 'f' to specify the forward transform. Use 'B' or 'b' to specify the inverse transform.

in, out
real*4 | real*8 | complex*8 | complex*16

Both the arguments are two-dimensional arrays. The input and output arrays can be the same array. The IN array contains the data to be transformed. The OUT array contains the transformed data.

in_real, out_real, in_imag, out_imag
real*4 | real*8
Use these arguments when performing a complex transform on real data format and storing the result in a real data format.

lda
integer*4
Specifies the number of rows in the IN and OUT arrays; $lda \geq ni$. For {S,D} routines, $lda \geq ni + 2$ when the input format is 'R' and the output format is 'C' or the input format is 'C' and the output format is 'R'.

fft_struct
record /dxml_s_fft_structure_2d/ for single-precision real operations
record /dxml_d_fft_structure_2d/ for double-precision real operations
record /dxml_c_fft_structure_2d/ for single-precision complex operations
record /dxml_z_fft_structure_2d/ for double-precision complex operations
The argument refers to the structure created by the _2D_INIT routine.

ni_stride, nj_stride
integer*4
Specifies the distance between consecutive elements in a column and row in the IN and OUT arrays; the valid stride depends on the _INIT routine; $ni\_stride \geq 1$, $nj\_stride \geq 1$.

Description

The _FFT_APPLY_2D routines compute the fast Fourier transform of two-dimensional data in either the forward or backward direction. These routines are the second step of the three-step procedure. They compute the fast forward or inverse Fourier transform, using the internal data structures created by the _FFT_2D_INIT subroutine.

Use the _APPLY_2D routines with their corresponding _INIT_2D and _EXIT_2D routines. For example, use SFFT_APPLY after the SFFT_INIT and end with the SFFT_EXIT routine. See Section 11.1.3.2 for an explanation of real and complex data format.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

Return Values

0	DXML_SUCCESS()
11	DXML_ILL_LDA()
12	DXML_INS_RES()
13	DXML_BAD_STRIDE()
15	DXML_BAD_DIRECTION_STRING()

16	DXML_BAD_FORMAT_STRING()
18 (for real transform only)	DXML_BAD_FORMAT_FOR_DIRECTION()

Examples

1.
```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N_I, N_J, STATUS, LDA
REAL*8 A(1026,512), B(1026,513)
RECORD /DXML_D_FFT_STRUCTURE_2D/ FFT_STRUCT
N_I = 1024
N_J = 512
LDA = 1026
STATUS = DFFT_INIT_2D(N_I,N_J,FFT_STRUCT,.TRUE.)
STATUS = DFFT_APPLY_2D('R','C','F',A,B,LDA,FFT_STRUCT,1,1)
STATUS = DFFT_EXIT_2D(FFT_STRUCT)
```

This Fortran code computes the forward, two-dimensional, real FFT of a 1024x512 matrix *A*. The result of the transform is stored in *B* in complex form. The leading dimension of *B* is 1026 in order to hold the extra complex values (see section on data storage). Also the input matrix *A* also requires a leading dimension of at least 1026.

2.
```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N_I, N_J, STATUS, LDA
COMPLEX*16 A(1024,512), B(1024,512)
RECORD /DXML_Z_FFT_STRUCTURE_2D/ FFT_STRUCT
N_I = 1024
N_J = 512
LDA = 1024
STATUS = ZFFT_INIT_2D(N_I,N_J,FFT_STRUCT,.TRUE.)
STATUS = ZFFT_APPLY_2D('C','C','F',A,B,LDA,FFT_STRUCT,1,1)
STATUS = ZFFT_EXIT_2D(FFT_STRUCT)
```

This Fortran code computes the forward, two-dimensional, complex FFT of a matrix *A*, of dimension 1024 by 512. The result of the transform is stored in *B* in complex form.

_FFT_EXIT_2D
Final Step for Fast Fourier Transform in Two Dimensions (Serial and Parallel Versions)

Format

status = {S,D,C,Z}FFT_EXIT_2D (fft_struct)

Arguments

fft_struct
record /dxml_s_fft_structure/ for single-precision real operations
record /dxml_d_fft_structure/ for double-precision real operations
record /dxml_c_fft_structure/ for single-precision complex operations
record /dxml_z_fft_structure/ for double-precision complex operations
This argument must be included but it is not necessary to modify it in any way. It refers to the data structure that was created in the initialization step.

Description

The _FFT_EXIT_2D functions remove the internal data structures created in the _FFT_INIT_2D functions. These functions are the final step in a three-step procedure. They release the virtual memory that was allocated by the _FFT_INIT_2D functions.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

Return Values

0	DXML_SUCCESS()
12	DXML_INS_RES()

_FFT_3D
Fast Fourier Transform in Three Dimensions (Serial and Parallel Versions)

Format

Real transform:

status = {S,D}FFT_3D (input_format, output_format, direction, in, out, ni, nj, nk, lda_i, lda_j, ni_stride, nj_stride, nk_stride)

Complex transforms in complex format:

status = {C,Z}FFT_3D (input_format, output_format, direction, in, out, ni, nj, nk, lda_i, lda_j, ni_stride, nj_stride, nk_stride)

Complex transform in real data format:

status = {C,Z}FFT_3D (input_format, output_format, direction, in_real, in_imag, out_real, out_imag, ni, nj, nk, lda_i, lda_j, ni_stride, nj_stride, nk_stride)

Arguments

input_format, output_format
character*(*)
Identifies the data type of the input and the format to be used to store the data, regardless of the data type. For example, a complex sequence can be stored in real format.

The character 'R' specifies the format as real; the character 'C' specifies the format as complex. As convenient, use either uppercase or lowercase characters, and either spell out or abbreviate the word.

The following table shows the valid values:

Subprogram	Input Format	Output Format	Direction
{S,D}	'R'	'C'	'F'
	'C'	'R'	'B'
	'R'	'R'	'F' or 'B'
{C,Z}	'R'	'R'	'F' or 'B'
	'C'	'C'	'F' or 'B'

For complex data, the type of data determines what other arguments are needed. When both the input and output data are real, the complex routines store the data as separate arrays for imaginary and real data so additional arguments are needed.

direction
character*(*)
Specifies the operation as either the forward or inverse transform. Use 'F' or 'f' to specify the forward transform. Use 'B' or 'b' to specify the inverse transform.

in, out
real*4 | real*8 | complex*8 | complex*16
Both the arguments are three-dimensional arrays. The input and output arrays can be the same array. The IN array contains the data to be transformed. The OUT array contains the transformed data.

in_real, out_real, in_imag, out_imag
REAL*4 | REAL*8
Use these arguments when performing a complex transform on real data format and storing the result in a real data format.

ni, nj, nk
integer*4
Specifies the size of the first, second, and third dimension of data in the input array; $ni > 0$, $nj > 0$, $nk > 0$. For SFFT_3D and DFFT_3D, **ni** must be even.

lda_i, lda_j
integer*4
Specifies the number of rows and columns in the IN and OUT arrays; $lda\_i \geq ni$, $lda\_j \geq nj$. For {S,D} routines, $lda\_i \geq ni + 2$ when the input format is 'R' and the output format is 'C' or the input format is 'C' and the output format 'R'.

ni_stride, nj_stride, nk_stride
integer*4
Specifies the distance between consecutive elements in the IN and OUT arrays; $ni\_stride \geq 1$, $nj\_stride \geq 1$, $nk\_stride \geq 1$.

Description

The _FFT_3D routines compute the fast forward or inverse Fourier transform of three-dimensional data in one step.

The SFFT_3D and DFFT_3D functions perform the forward Fourier transform of a real sequence and store the result in either real or complex data format. These functions also perform the inverse Fourier transform of a complex sequence into a real sequence.

The CFFT_3D and ZFFT_3D functions perform Fourier transforms on a complex sequence. However, the argument list is different, depending on the data format in which the output data is stored. See Section 11.1.3.2 for an explanation of real and complex data format.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

Return Values

0	DXML_SUCCESS()
4 (with real transforms only)	DXML_ILL_N_IS_ODD()
8	DXML_ILL_N_RANGE()
11	DXML_ILL_LDA()
12	DXML_INS_RES()

13	DXML_BAD_STRIDE()
15	DXML_BAD_DIRECTION_STRING()
16	DXML_BAD_FORMAT_STRING()
18	DXML_BAD_FORMAT_FOR_DIRECTION()

Example

```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N_I, N_J, N_K, STATUS, LDA_I, LDA_J
REAL*8 A(130,128,128), B(130,128,128)
N_I = 128
N_J = 128
N_K = 128
LDA_I = 130
LDA_J = 128
STATUS = DFFT_3D('R','C','F',A,B,N_I,N_J,N_K,LDA_I,LDA_J,1,1,1)
```

This Fortran code computes the forward, three-dimensional, real FFT of a 128x128x128 matrix *A*. The result of the transform is stored in *B* in complex form. The leading dimension of *B* is 130 to hold the extra complex values (see section on data storage). Also the input matrix *A* requires a leading dimension of at least 130.

```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N_I, N_J, N_K, STATUS, LDA_I, LDA_J
COMPLEX*8 A(130,128,128), B(130,128,128)
N_I = 128
N_J = 128
N_K = 128
LDA_I = 128
LDA_J = 128
STATUS = CFFT_3D('C','C','F',A,B,N_I,N_J,N_K,LDA_I,LDA_J,1,1,1)
```

This Fortran code computes the forward, three-dimensional, complex FFT of a matrix *A*, of dimension 128x128x128. The result of the transform is stored in *B* in complex form.

_FFT_INIT_3D
Initialization Step for Fast Fourier Transform in Three Dimensions (Serial and Parallel Versions)

Format

status = {S,D,C,Z}FFT_INIT_3D (ni, nj, nk, fft_struct, ni_stride_1_flag)

Arguments

ni, nj, nk
integer*4
Specifies the size of the first, second, and third dimension of data in the input array; $ni > 0$, $nj > 0$, $nk > 0$. For SFFT_INIT_3D and DFFT_INIT_3D, **ni** must be even.

fft_struct
record /dxml_s_fft_structure_3d/ for single-precision real operations
record /dxml_d_fft_structure_3d/ for double-precision real operations
record /dxml_c_fft_structure_3d/ for single-precision complex operations
record /dxml_z_fft_structure_3d/ for double-precision complex operations
This argument must be included but needs no additional definitions. The argument is declared in the program before this routine. See Section 11.1.3.3 for more information.

ni_stride_1_flag
logical
Specifies whether to allow a stride of more than 1 between elements in the row:

TRUE: Stride must be 1.
FALSE: Stride is at least 1.

Description

The _FFT_INIT_3D functions build internal data structures needed to compute fast Fourier transforms of three-dimensional data. These routines are the first step in a three-step procedure. They create the internal data structures, using attributes defined in the file DXMLDEF.FOR.

Use the initialization routine that is appropriate for the data format. Then, use the corresponding application and exit steps to complete the procedure. For example, use the SFFT_3D_INIT routine with the SFFT_3D_APPLY and SFFT_3D_EXIT routine.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

Return Values

0	DXML_SUCCESS()
4 (real transform only)	DXML_ILL_N_IS_ODD()
8	DXML_ILL_N_RANGE()
12	DXML_INS_RES()

_FFT_APPLY_3D
Application Step for Fast Fourier Transform in Three Dimensions (Serial and Parallel Versions)

Format

Real transform:

status = {S,D}FFT_APPLY_3D (input_format, output_format, direction, in, out, lda_i, lda_j, fft_struct, ni_stride, nj_stride, nk_stride)

Complex transforms in complex format:

status = {C,Z}FFT_APPLY_3D (input_format, output_format, direction, in, out, lda_i, lda_j, fft_struct, ni_stride, nj_stride, nk_stride)

Complex transform in real data format:

status = {C,Z}FFT_APPLY_3D (input_format, output_format, direction, in_real, in_imag, out_real, out_imag, lda_i, lda_j, fft_struct, ni_stride, nj_stride, nk_stride)

Arguments

input_format, output_format
character*(*)
Identifies the data type of the input and the format to be used to store the data, regardless of the data type. For example, a complex sequence can be stored in real format.

The character 'R' specifies the format as real; the character 'C' specifies the format as complex. As convenient, use either uppercase or lowercase characters, and either spell out or abbreviate the word.

The following table shows the valid values:

Subprogram	Input Format	Output Format	Direction
{S,D}	'R'	'C'	'F'
	'C'	'R'	'B'
	'R'	'R'	'F' or 'B'
{C,Z}	'R'	'R'	'F' or 'B'
	'C'	'C'	'F' or 'B'

For complex data, the type of data determines what other arguments are needed. When both the input and output data are real, the complex routines store the data as separate arrays for imaginary and real data so additional arguments are needed.

direction
character*(*)
Specifies the operation as either the forward or inverse transform. Use 'F' or 'f' to specify the forward transform. Use 'B' or 'b' to specify the inverse transform.

in, out
real*4 | real*8 | complex*8 | complex*16
Both the arguments are three-dimensional arrays. The input and output arrays can be the same array. The IN array contains the data to be transformed. The OUT array contains the transformed data.

in_real, out_real, in_imag, out_imag
REAL*4 | REAL*8
Use these arguments when performing a complex transform on real data format and storing the result in a real data format.

lda_i, lda_j
integer*4
Specifies the number of rows and columns in the IN and OUT arrays; $lda\_i \geq ni$, $lda\_j \geq nj$. For {S,D} routines, $lda\_i \geq ni + 2$ when the input format is 'R' and the output format is 'C' or the input format is 'C' and the output format is 'R'.

fft_struct
record /dxml_s_fft_structure_3d/ for single-precision real operations
record /dxml_d_fft_structure_3d/ for double-precision real operations
record /dxml_c_fft_structure_3d/ for single-precision complex operations
record /dxml_z_fft_structure_3d/ for double-precision complex operations
The argument refers to the structure created by the _INIT routine.

ni_stride, nj_stride, nk_stride
integer*4
Specifies the distance between consecutive elements in the IN and OUT arrays; the valid stride depends on the _INIT routine. $ni\_stride \geq 1$, $nj\_stride \geq 1$, $nk\_stride \geq 1$.

Description

The _FFT_APPLY_3D routines compute the fast Fourier transform of three-dimensional data in either the forward or backward direction. These routines are the second step of the three-step procedure for the fast Fourier transform of three-dimensional data. They compute the fast forward or inverse Fourier transform, using the internal data structures created by the _FFT_3D_INIT subroutine.

Use the _APPLY_3D routines with their corresponding _INIT_3D and _EXIT_3D routines. For example, use SFFT_APPLY after the SFFT_INIT and end with the SFFT_EXIT routine. See Section 11.1.3.2 for an explanation of real and complex data format.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

Return Values

0	DXML_SUCCESS()
11	DXML_ILL_LDA()
12	DXML_INS_RES()
13	DXML_BAD_STRIDE()
18 (for real transform only)	DXML_BAD_FORMAT_FOR_DIRECTION()
15	DXML_BAD_DIRECTION_STRING()
16	DXML_BAD_FORMAT_STRING()

Example

```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N_I, N_J, N_K, STATUS, LDA_I, LDA_J
REAL*8 A(130,128,128), B(130,128,128)
RECORD /DXML_D_FFT_STRUCTURE_3D/ FFT_STRUCT
N_I = 128
N_J = 128
N_K = 128
LDA_I = 130
LDA_J = 128
STATUS = DFFT_INIT_3D(N_I,N_J,N_K,FFT_STRUCT,.TRUE.)
STATUS = DFFT_APPLY_3D('R','C','F',A,B,LDA_I,LDA_J,FFT_STRUCT,1,1,1)
STATUS = DFFT_EXIT_3D(FFT_STRUCT)
```

This Fortran code computes the forward, three-dimensional, real FFT of a 128x128x128 matrix *A*. The result of the transform is stored in *B* in complex form. The leading dimension of *B* is 130 to hold the extra complex values (see section on data storage). Also the input matrix *A* requires a leading dimension of at least 130.

_FFT_EXIT_3D
Final Step for Fast Fourier Transform in Three Dimensions (Serial and Parallel Versions)

Format

status = {S,D,C,Z}FFT_EXIT_3D (fft_struct)

Arguments

fft_struct
record /dxml_s_fft_structure/ for single-precision real operations
record /dxml_d_fft_structure/ for double-precision real operations
record /dxml_c_fft_structure/ for single-precision complex operations
record /dxml_z_fft_structure/ for double-precision complex operations
This argument must be included but it is not necessary to modify it in any way. It refers to the data structure that was created in the initialization step.

Description

The _FFT_EXIT_3D functions remove the internal data structures created in the _FFT_INIT_3D functions. These functions are the final step in a three-step procedure. They release the virtual memory that was allocated by the _FFT_INIT_3D functions.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

Return Values

0	DXML_SUCCESS()
12	DXML_INS_RES()

_FFT_GRP
Group Fast Fourier Transform in One Dimension

Format

Real transform:

status = {S,D}FFT_GRP (input_format, output_format, direction, in, out, n, grp_size, lda, stride, grp_stride)

Complex transform in complex data format:

status = {C,Z}FFT_GRP (input_format, output_format, direction, in, out, n, grp_size, lda, stride, grp_stride)

Complex transform real data format:

status = {C,Z}FFT_GRP (input_format, output_format, direction, in_real, in_imag, out_real, out_imag, n, grp_size, lda, stride, grp_stride)

Arguments

input_format, output_format
character*(*)
Identifies the data type of the input and the format to be used to store the data, regardless of the data type. For example, a complex sequence can be stored in real format.

The character 'R' specifies the format as real; the character 'C' specifies the format as complex. As convenient, use either uppercase or lowercase characters, and either spell out or abbreviate the word.

The following table shows the valid values:

Subprogram	Input Format	Output Format	Direction
{S,D}	'R'	'C'	'F'
	'C'	'R'	'B'
	'R'	'R'	'F' or 'B'
{C,Z}	'R'	'R'	'F' or 'B'
	'C'	'C'	'F' or 'B'

For complex transforms, the type of data determines what other arguments are needed. When both the input and output data are real, the complex functions store the data as separate arrays for imaginary and real data so additional arguments are needed.

direction
character*(*)
Specifies the operation as either the forward or inverse transform. Use 'F' or 'f' to specify the forward transform. Use 'B' or 'b' to specify the inverse transform.

in, out
real*4 | real*8 | complex*8 | complex*16
Both arguments are two-dimensional arrays. The IN array contains the data to be transformed. The OUT array contains the transformed data. The IN and OUT arrays can be the same array. The IN and OUT arrays must be the same size.

in_real, in_imag, out_real, out_imag
real*4 | real*8
Use these arguments when performing a complex transform on real data format and storing the result in a real data format.

n
integer*4
Specifies the number of elements in the column within each one-dimensional data array; $n > 0$. For SFFT_GRP and DFFT_GRP, **n** must be even.

grp_size
integer*4
Specifies the number of one-dimensional data arrays; *grp_size*> 0.

lda
integer*4
Specifies the the number of rows in two-dimensional data arrays; $lda \geq grp\_size$. Using $lda = grp\_size + \{3 \text{ or } 5\}$ can sometimes achieve better performance by avoiding cache thrashing.

stride
integer*4
Specifies the distance between columns of active data arrays; $stride \geq 1$. **stride** permits columns of IN and OUT arrays to be skipped when they are not part of the group.

grp_stride
integer*4
Specifies the distance between consecutive elements in a row in the IN and OUT arrays; $grp\_stride \geq 1$. **grp_stride** permits rows of IN and OUT arrays to be skipped when they are not part of the group.

Description

_FFT_GRP computes the fast forward or inverse Fourier transform on a group of one-dimensional data arrays. The transform is performed on the first row of elements of one-dimensional data arrays within the group. Data arrays can be skipped by setting the **stride** parameter. The transform then goes to the next row of elements. Similarly, rows of elements can be skipped by setting the **grp_stride** parameter. _FFT_GRP contrasts with _FFT in that _FFT performs a transform on each element of a data array before going to the next data array. Although _FFT_GRP gives the same result as _FFT, _FFT_GRP is more efficient at completing the transform.

Return Values

0	DXML_SUCCESS()
4 (real transforms only)	DXML_ILL_N_IS_ODD()
8	DXML_ILL_N_RANGE()
12	DXML_INS_RES()
13	DXML_BAD_STRIDE()
15	DXML_BAD_DIRECTION_STRING()

16	DXML_BAD_FORMAT_STRING()
18 (real transforms only)	DXML_BAD_FORMAT_FOR_DIRECTION()

Example

```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N,GRP_SIZE
REAL*8 A(100,514),B(100,514)
N=512
LDA=100
GRP_SIZE=100
STATUS=DFFT_GRP('R','C','F',A,B,N,GRP_SIZE,LDA,1,1)
```

This Fortran code computes a set of 100 FFT of size 512. The results of the transforms are stored in *B* in complex format. The second dimension of *A* and *B* is 514 to hold the extra complex values.

_FFT_INIT_GRP
Initialization Step for Group Fast Fourier Transform in One Dimension

Format

status = {S,D,C,Z}FFT_INIT_GRP (n, fft_struct, grp_stride_1_flag, max_grp_size)

Arguments

n
integer*4
Specifies the number of elements in the column within each one-dimensional data array; $n > 0$. For SFFT_INIT_GRP and DFFT_INIT_GRP, **n** must be even.

fft_struct
record /dxml_s_grp_fft_structure/ for single-precision real operations
record /dxml_d_grp_fft_structure/ for double-precision real operations
record /dxml_c_grp_fft_structure/ for single-precision complex operations
record /dxml_z_grp_fft_structure/ for double-precision complex operations
You must include this argument but it needs no additional definitions. The argument is declared in the program before this routine. See Section 11.1.3.3 for more information.

grp_stride_1_flag
logical
Specifies whether to allow a distance greater than 1 between elements:

TRUE: Group stride must be 1.
FALSE: Group stride is at least 1.

max_grp_size
integer*4
Specifies the expected number of sets of data. If unknown, set *max_grp_size* = 0.

Description

The _FFT_INIT_GRP functions build internal data structures needed to compute fast Fourier transforms of one-dimensional data. These routines are the first step in a three-step procedure. They create the internal data structures, using attributes defined in the file DXMLDEF.FOR.

Return Values

0	DXML_SUCCESS()
4 (real transforms only)	DXML_ILL_N_IS_ODD()
8	DXML_ILL_N_RANGE()
12	DXML_INS_RES()

_FFT_APPLY_GRP
Application Step for Group Fast Fourier Transform in One Dimension

Format

Real transform:

status = {S,D}FFT_APPLY_GRP (input_format, output_format, direction, in, out, grp_size, lda, fft_struct, stride, grp_stride)

Complex transform of real data format:

status = {C,Z}FFT_APPLY_GRP (input_format, output_format, direction, in, out, grp_size, lda, fft_struct, stride, grp_stride)

Complex transform of real data to real data:

status = {C,Z}FFT_APPLY_GRP (input_format, output_format, direction, in_real, in_imag, out_real, out_imag, grp_size, lda, fft_struct, stride, grp_stride)

Arguments

input_format, output_format
character*(*)
Identifies the data type of the input and the format to be used to store the data, regardless of the data type. For example, a complex sequence can be stored in real format.

The character 'R' specifies the format as real; the character 'C' specifies the format as complex. As convenient, use either uppercase or lowercase characters, and either spell out or abbreviate the word.

The following table shows the valid values:

Subprogram	Input Format	Output Format	Direction
{S,D}	'R'	'C'	'F'
	'C'	'R'	'B'
	'R'	'R'	'F' or 'B'
{C,Z}	'R'	'R'	'F' or 'B'
	'C'	'C'	'F' or 'B'

For complex transforms, the type of data determines what other arguments are needed. When both the input and output data are real, the complex functions store the data as separate arrays for imaginary and real data so additional arguments are needed.

direction
character*(*)
Specifies the operation as either the forward or inverse transform. Use 'F' or 'f' to specify the forward transform. Use 'B' or 'b' to specify the inverse transform.

in, out
real*4 | real*8 | complex*8 | complex*16
Both arguments are two-dimensional arrays. The IN array contains the data to be transformed. The OUT array contains the transformed data. The IN and OUT arrays can be the same array.

in_real, in_imag, out_real, out_imag
real*4 | real*8
Use these arguments when performing a complex transform on real data format and storing the result in a real data format.

grp_size
integer*4
Specifies the number of one-dimensional data arrays; *grp_size*> 0.

lda
integer*4
Specifies the the number of rows in two-dimensional data arrays; $lda \geq grp\_size$. Using $lda = grp\_size + \{3 \text{ or } 5\}$ can sometimes achieve better performance by avoiding cache thrashing.

fft_struct
record /dxml_s_grp_fft_structure/ for single-precision real operations
record /dxml_d_grp_fft_structure/ for double-precision real operations
record /dxml_c_grp_fft_structure/ for single-precision complex operations
record /dxml_z_grp_fft_structure/ for double-precision complex operations
The argument refers to the structure created by the _INIT routine.

stride
integer*4
Specifies the distance between columns of active data arrays; $stride \geq 1$. **stride** permits columns of IN and OUT arrays to be skipped when they are not part of the group.

grp_stride
integer*4
Specifies the distance between consecutive elements in a row in the IN and OUT arrays; $grp\_stride \geq 1$. **grp_stride** permits rows of IN and OUT arrays to be skipped when they are not part of the group.

Description

The _FFT_APPLY_GRP computes the fast forward or inverse Fourier transform on a group of one-dimensional data arrays. The transform is performed on the first row of elements of one-dimensional data arrays within the group. Data arrays can be skipped by setting the **stride** parameter. The transform then goes to the next row of elements. Similarly, rows of elements can be skipped by setting the **grp_stride** parameter. _FFT_APPLY_GRP contrasts with _FFT_APPLY in that _FFT_APPLY performs a transform on each element of a data array before going to the next data array. Although _FFT_APPLY_GRP gives the same result as _FFT_APPLY, _FFT_APPLY_GRP is more efficient at completing the transform.

Return Values

0	DXML_SUCCESS()
12	DXML_INS_RES()
13	DXML_BAD_STRIDE()
15	DXML_BAD_DIRECTION_STRING()

16	DXML_BAD_FORMAT_STRING()
18 (real transform only)	DXML_BAD_FORMAT_FOR_DIRECTION()

Example

```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 GRP_SIZE,N,STATUS
REAL*8 A(100,514),B(100,514)
RECORD /DXML_D_GRP_FFT_STRUCTURE/FFT_STRUCT
GRP_SIZE=100
N=512
LDA=100
STATUS = DFFT_INIT_GRP(N,FFT_STRUCT,.TRUE.,100)
STATUS = DFFT_APPLY_GRP('R','C','F',A,B,GRP_SIZE,LDA,FFT_STRUCT,1,1)
STATUS = DFFT_EXIT_GRP(FFT_STRUCT)
```

This Fortran code computes a set of 100 FFT of size 512. The results of the transforms are stored in *B* in complex format. The second dimension is 514 to hold the extra complex values.

_FFT_EXIT_GRP
Exit Step for Group Fast Fourier Transform in One Dimension

Format

status = {S,D,C,Z}FFT_EXIT_GRP (fft_struct)

Arguments

fft_struct
record /dxml_s_grp_fft_structure/ for single-precision real operations
record /dxml_d_grp_fft_structure/ for double-precision real operations
record /dxml_c_grp_fft_structure/ for single-precision complex operations
record /dxml_z_grp_fft_structure/ for double-precision complex operations
This argument must be included but it is not necessary to modify it in any way. It refers to the data structure that was created in the initialization step.

Description

The _FFT_EXIT_GRP functions remove the internal data structures created in the _FFT_INIT_GRP functions. These functions are the final step in a three-step procedure. They release the virtual memory that was allocated by the _FFT_INIT_GRP functions.

Return Values

0	DXML_SUCCESS()
12	DXML_INS_RES()

Cosine and Sine Transform Subprograms

This section provides descriptions of the routines for the Cosine and Sine transforms.

_FCT
Fast Cosine Transform in One Dimension

Format

status = {S,D}FCT (direction, in, out, n, type, stride)

Arguments

direction
character*(*)
Specifies the operation as either the forward or inverse transform. Use 'F' or 'f' to specify the forward transform. Use 'B' or 'b' to specify the inverse transform.

in, out
real*4 | real*8
Both the arguments are one-dimensional arrays. The input and output arrays can be the same array. The IN array contains the data to be transformed. The OUT array contains the transformed data. For type-I FCT, the length of the IN,OUT array must $>= N + 1$. For type-II FCT, the length of the IN,OUT array must be $>= N$.

n
integer*4
Specifies the size of the transform. The minimum size of the IN, OUT array is **n**; where **n**> 0 and even.

type
integer*4
Specifies the type of the Cosine transform desired. Currently only type-1 and type-2 transforms are supported.

stride
integer*4
Specifies the distance between consecutive elements in the IN and OUT arrays. The distance must be at least 1.

Description

The _FCT functions compute the fast Cosine transform of one-dimensional data in one step.

Return Values

0	DXML_SUCCESS()
4	DXML_ILL_N_IS_ODD()
8	DXML_ILL_N_RANGE()
12	DXML_INS_RES()
13	DXML_BAD_STRIDE()
15	DXML_BAD_DIRECTION_STRING()
17	DXML_OPTION_NOT_SUPPORTED()

Example

```
INTEGER*4 N
PARAMETER (N=1024)

INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 STATUS
REAL*8 C(0:N),D(0:N)
REAL*4 E(1:N-1),F(1:N-1)

STATUS = DFCT('F',C,D,N,1,1)
STATUS = DFCT('B',D,C,N,1,1)
STATUS = SFCT('F',E,F,N,2,1)
STATUS = SFCT('B',F,E,N,2,1)
```

This Fortran code computes the following:

- Forward Type-1 Cosine transform of the real sequence C to real sequence D.
- Backward Type-1 Cosine transform of the real sequence D to real sequence C.
- Forward Type-2 Cosine transform of the real sequence E to real sequence F.
- Backward Type-2 Cosine transform of the real sequence F to real sequence E.

_FCT_INIT
Initialization Step for Fast Cosine Transform in One Dimension

Format

status = {S,D}FCT_INIT (n, fct_struct, type, stride_1_flag)

Arguments

n
integer*4
Specifies the size of the transform. **n** must be even and > 0.

fct_struct
record /dxml_s_fct_structure/ for single-precision operations
record /dxml_d_fct_structure/ for double-precision operations

type
integer*4
Specifies the type of Cosine transform desired. Currently only type-1 and type-2 transforms are supported.

stride_1_flag
logical
Specifies the allowed distance between consecutive elements in the input and output arrays:

TRUE: Stride must be 1.
FALSE: Stride is at least 1.

Description

The _FCT_INIT functions build internal data structures needed to compute fast Cosine transforms of one-dimensional data. These functions are the first step in a three-step procedure. They create the internal data structures, using attributes defined in the file DXMLDEF.FOR.

Use the initialization function that is appropriate for the data format. Then use the corresponding application and exit functions to complete the transform. For example, use SFCT_INIT for the internal data structures used by SFCT_APPLY and end with the SFCT_EXIT function.

Return Values

0	DXML_SUCCESS()
4	DXML_ILL_N_IS_ODD()
8	DXML_ILL_N_RANGE()
12	DXML_INS_RES()
17	DXML_OPTION_NOT_SUPPORTED()

_FCT_APPLY
Application Step for Fast Cosine Transform in One Dimension

Format

status = {S,D}FCT_APPLY (direction, in, out, fct_struct, stride)

Arguments

direction
character*(*)
Specifies the operation as either the forward or inverse transform. Use 'F' or 'f' to specify the forward transform. Use 'B' or 'b' to specify the inverse transform.

in, out
real*4 | real*8
Both the arguments are one-dimensional arrays. The input and output arrays can be the same array. The IN array contains the data to be transformed. The OUT array contains the transformed data.

fct_struct
record /dxml_s_fct_structure/ for single-precision operations
record /dxml_d_fct_structure/ for double-precision operations

stride
integer*4
Specifies the distance between consecutive elements in the input and output arrays, depending on the value of **stride_1_flag** provided in the _INIT function.

Description

The _FCT_APPLY functions compute the fast Cosine transform of one-dimensional data in three steps.

Return Values

0	DXML_SUCCESS()
12	DXML_INS_RES()
13	DXML_BAD_STRIDE()
15	DXML_BAD_DIRECTION_STRING()
17	DXML_OPTION_NOT_SUPPORTED()

Example

```
INTEGER*4 N
PARAMETER (N=1024)

INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 STATUS
RECORD /DXML_S_FCT_STRUCTURE/ D_FCT_STRUCT
RECORD /DXML_D_FCT_STRUCTURE/ D_FCT_STRUCT
REAL*8 C(0:N),D(0:N)
REAL*4 E(0:N-1),F(0:N-1)
```

```
STATUS = DFCT_INIT(N,D_FCT_STRUCT,1,.TRUE.)
STATUS = DFCT_APPLY('F',C,D,D_FCT_STRUCT,1)
STATUS = DFCT_APPLY('B',D,C,D_FCT_STRUCT,1)
STATUS = DFCT_EXIT(D_FCT_STRUCT)

STATUS = SFCT_INIT(N,S_FCT_STRUCT,2,.TRUE.)
STATUS = SFCT_APPLY('F',E,F,S_FCT_STRUCT,1)
STATUS = SFCT_APPLY('B',F,E,S_FCT_STRUCT,1)
STATUS = SFCT_EXIT(D_FCT_STRUCT)
```

This Fortran code computes the following:

- Forward Type-1 Cosine transform of the real sequence C to real sequence D.
- Backward Type-1 Cosine transform of the real sequence D to real sequence C.
- Forward Type-2 Cosine transform of the real sequence E to real sequence F.
- Backward Type-2 Cosine transform of the real sequence F to real sequence E.

_FCT_EXIT
Final Step for Fast Cosine Transform in One Dimension

Format

status = {S,D}FCT_EXIT (fct_struct)

Arguments

fct_struct
record /dxml_s_fct_structure/ for single-precision operations
record /dxml_d_fct_structure/ for double-precision operations

Description

The _FCT_EXIT functions remove the internal data structures created in the _FCT_INIT functions. These functions are the final step in a three-step procedure. They release the virtual memory that was allocated by the _FCT_INIT functions.

Return Values

0	DXML_SUCCESS()
12	DXML_INS_RES()

_FST
Fast Sine Transform in One Dimension

Format

status = {S,D}FST (direction, in, out, n, type, stride)

Arguments

direction
character*(*)
Specifies the operation as either the forward or inverse transform. Use 'F' or 'f' to specify the forward transform. Use 'B' or 'b' to specify the inverse transform.

in, out
real*4 | real*8
Both the arguments are one-dimensional arrays. The input and output arrays can be the same array. The IN array contains the data to be transformed. The OUT array contains the transformed data.

n
integer*4
Specifies the size of the transform. The minimum size of the IN, OUT array is **n**; where **n**> 0 and even.

type
integer*4
Specifies the type of Sine transform desired. Currently only type-1 and type-2 transforms are supported.

stride
integer*4
Specifies the distance between consecutive elements in the input and output arrays. The distance must be at least 1.

Description

The _FST functions compute the fast type-1 and type-2 Sine transform of one-dimensional data in one step.

Return Values

0	DXML_SUCCESS()
4	DXML_ILL_N_IS_ODD()
8	DXML_ILL_N_RANGE()
12	DXML_INS_RES()
13	DXML_BAD_STRIDE()
15	DXML_BAD_DIRECTION_STRING()
17	DXML_OPTION_NOT_SUPPORTED()

Example

```
INTEGER*4 N
PARAMETER (N=1024)

INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 STATUS
REAL*8 C(1:N),D(1:N)
REAL*4 E(1:N),F(1:N)

STATUS = DFST('F',C,D,N,1,1)
STATUS = DFST('B',D,C,N,1,1)
STATUS = SFST('F',E,F,N,2,1)
STATUS = SFST('B',F,E,N,2,1)
```

This Fortran code computes the following:

- Forward Type-1 Sine transform of the real sequence C to real sequence D.
- Backward Type-1 Sine transform of the real sequence D to real sequence C.
- Forward Type-2 Sine transform of the real sequence E to real sequence F.
- Backward Type-2 Sine transform of the real sequence F to real sequence E.

_FST_INIT
Initialization Step for Fast Sine Transform in One Dimension

Format

status = {S,D}FST_INIT (n, fst_struct, type, stride_1_flag)

Arguments

n
integer*4
Specifies the size of the transform. **n** must be even and > 0.

fst_struct
record /dxml_s_fst_structure/ for single-precision operations
record /dxml_d_fst_structure/ for double-precision operations

type
integer*4
Specifies the type of Sine transform desired. Currently only type-1 and type-2 transforms are supported.

stride_1_flag
logical
Specifies the allowed distance between consecutive elements in the input and output arrays:

TRUE: Stride must be 1.
FALSE: Stride is at least 1.

Description

The _FST_INIT functions build internal data structures needed to compute fast Sine transforms of one-dimensional data. These functions are the first step in a three-step procedure. They create the internal data structures, using attributes defined in the file DXMLDEF.FOR.

Use the initialization function that is appropriate for the data format. Then use the corresponding application and exit functions to complete the transform. For example, use SFST_INIT for the internal data structures used by SFST_APPLY and end with the SFST_EXIT function.

Return Values

0	DXML_SUCCESS()
4	DXML_ILL_N_IS_ODD()
8	DXML_ILL_N_RANGE()
12	DXML_INS_RES()
17	DXML_OPTION_NOT_SUPPORTED()

_FST_APPLY
Application Step for Fast Sine Transform in One Dimension

Format

status = {S,D}FST_APPLY (direction, in, out, fst_struct, stride)

Arguments

direction
character*(*)
Specifies the operation as either the forward or inverse transform. Use 'F' or 'f' to specify the forward transform. Use 'B' or 'b' to specify the inverse transform.

in, out
real*4 | real*8
Both the arguments are one-dimensional arrays. The input and output arrays can be the same array. The IN array contains the data to be transformed. The OUT array contains the transformed data.

fst_struct
record /dxml_s_fst_structure/ for single-precision operations
record /dxml_d_fst_structure/ for double-precision operations

stride
integer*4
Specifies the distance between consecutive elements in the input and output arrays, depending on the value of **stride_1_flag** provided in the _INIT function.

Description

The _FST_APPLY functions compute the fast Sine transform of one-dimensional data in three steps.

Return Values

0	DXML_SUCCESS()
12	DXML_INS_RES()
13	DXML_BAD_STRIDE()
15	DXML_BAD_DIRECTION_STRING()
17	DXML_OPTION_NOT_SUPPORTED()

Example

```
INTEGER*4 N
PARAMETER (N=1024)

INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 STATUS
RECORD /DXML_S_FST_STRUCTURE/ D_FST_STRUCT
RECORD /DXML_D_FST_STRUCTURE/ D_FST_STRUCT
REAL*8 C(1:N),D(1:N)
REAL*4 E(1:N),F(1:N)
```

```
STATUS = DFST_INIT(N,D_FST_STRUCT,1,.TRUE.)
STATUS = DFST_APPLY('F',C,D,D_FST_STRUCT,1)
STATUS = DFST_APPLY('B',D,C,D_FST_STRUCT,1)
STATUS = DFST_EXIT(D_FST_STRUCT)

STATUS = SFST_INIT(N,S_FST_STRUCT,2,.TRUE.)
STATUS = SFST_APPLY('F',E,F,S_FST_STRUCT,1)
STATUS = SFST_APPLY('B',F,E,S_FST_STRUCT,1)
STATUS = SFST_EXIT(D_FST_STRUCT)
```

This Fortran code computes the following:

- Forward Type-1 Sine transform of the real sequence C to real sequence D.
- Backward Type-1 Sine transform of the real sequence D to real sequence C.
- Forward Type-2 Sine transform of the real sequence E to real sequence F.
- Backward Type-2 Sine transform of the real sequence F to real sequence E.

_FST_EXIT
Final Step for Fast Sine Transform in One Dimension

Format

status = {S,D}FST_EXIT (fst_struct)

Arguments

fst_struct
record /dxml_s_fst_structure/ for single-precision operations
record /dxml_d_fst_structure/ for double-precision operations

Description

The _FST_EXIT functions remove the internal data structures created in the _FST_INIT functions. These functions are the final step in a three-step procedure. They release the virtual memory that was allocated by the _FST_INIT functions.

Return Values

0	DXML_SUCCESS()
12	DXML_INS_RES()

Convolution and Correlation Subprograms

This section provides descriptions of the convolution and correlation subroutines. The four versions of each routine are titled by name using a leading underscore character. The section is ordered in the following way:

- By type of subprogram:

  Convolution
  Correlation

- By function within each type:

  Nonperiodic
  Periodic

- Extensions to each subprogram

  The extended versions of each subprogram perform the same operation as the corresponding short subprogram but the argument list controls the operation, in the following way:

  - Specify a stride for both the input and output arrays
  - Scale the output by either a scalar or a vector
  - Add an output array to prior output, instead of overwriting
  - Operate on portions of the output array, instead of the entire array

_CONV_NONPERIODIC
Nonperiodic Convolution

Format

{S,D,C,Z}CONV_NONPERIODIC (x, y, out, nx, ny, status)

Arguments

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the data to be convolved.
On exit, **x** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the convolution or "filter" function that is to be convolved with the data from the X array.
On exit, **y** is unchanged.

out
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array OUT of length $nx + ny - 1$.
On exit, **out** contains the convolved data.

nx
integer*4
On entry, the number of values to be convolved, that is, the length of the X array; $nx > 0$.
On exit, **nx** is unchanged.

ny
integer*4
On entry, the length of the array containing the convolution function; $ny > 0$.
On exit, **ny** is unchanged.

status
integer*4

0 DXML_SUCCESS()
8 DXML_ILL_N_RANGE()

Description

The _CONV_NONPERIODIC routines compute the nonperiodic convolution of two arrays using a discrete summing technique.

Example

```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N_F, N_M, STATUS
REAL*4 A(500), B(15000), C(15499)
N_A = 500
N_B = 15000
CALL SCONV_NONPERIODIC(A,B,C,N_A,N_B,STATUS)
```

This Fortran code computes the nonperiodic convolution of two vectors of real numbers, a and b, with lengths of 500 and 15000, respectively. The result is stored in c with length of 15499.

_CONV_PERIODIC
Periodic Convolution

Format

{S,D,C,Z}CONV_PERIODIC (x, y, out, n, status)

Arguments

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the data to be convolved.
On exit, **x** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the convolution or "filter" function that is to be convolved with the data from the X array.
On exit, **y** is unchanged.

out
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array OUT of length n.
On exit, **out** is overwritten and contains the convolved data.

n
integer*4
On entry, the length of the input arrays X and Y and the length of the output array OUT; $n > 0$.
On exit, **n** is unchanged.

status
integer*4

0 DXML_SUCCESS()
8 DXML_ILL_N_RANGE()

Description

The _CONV_PERIODIC functions compute the periodic convolution of two arrays using a discrete summing technique.

Example

```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N, STATUS
REAL*8 A(15000), B(15000), C(15000)
N = 15000
CALL DCONV_PERIODIC(A,B,C,N,STATUS)
```

This Fortran code computes the periodic convolution of two vectors of double-precision real numbers, a and b, with lengths of 15000. The result is stored in c with length of 15000.

_CORR_NONPERIODIC
Nonperiodic Correlation

Format

{S,D,C,Z}CORR_NONPERIODIC (x, y, out, nx, ny, status)

Arguments

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the data to be correlated.
On exit, **x** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the function to be correlated with the data from the X array.
On exit, **y** is unchanged.

out
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array OUT with the length nx.
On exit, **out** contains the positive correlation coefficients.

nx
integer*4
On entry, the number of values to be correlated, that is, the length of the X array; $nx > 0$.
On exit, **nx** is unchanged.

ny
integer*4
On entry, the length of the Y array containing the correlation function; $ny > 0$.
On exit, **ny** is unchanged.

status
integer*4

0 DXML_SUCCESS()
8 DXML_ILL_N_RANGE()

Description

The _CORR_NONPERIODIC functions compute the nonperiodic correlation of two arrays using a discrete summing technique.

Example

```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N_F, N_M, STATUS
REAL*8 A(500), B(15000), C(500)
N_A = 500
N_B = 15000
CALL DCORR_NONPERIODIC(A,B,C,N_A,N_B,STATUS)
```

This Fortran code computes the nonperiodic correlation of two vectors of double-precision real numbers, a and b, with lengths of 500 and 15000, respectively. The result is stored in c with length of 15499.

_CORR_PERIODIC
Periodic Correlation

Format

{S,D,C,Z}CORR_PERIODIC (x, y, out, n, status)

Arguments

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the data to be correlated.
On exit, **x** is unchanged.

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the correlation function to be correlated with the data from the X array.
On exit, **y** is unchanged.

out
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array OUT of length n.
On exit, **out** contains the positive correlation coefficients.

n
integer*4
On entry, the length of the input arrays X and Y and the length of the output array OUT; $n > 0$.
On exit, **n** is unchanged.

status
integer*4

0 DXML_SUCCESS()
8 DXML_ILL_N_RANGE()

Description

The _CORR_PERIODIC computes the periodic correlation of two arrays using a discrete summing technique.

Example

```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N, STATUS
REAL*8 A(15000), B(15000), C(15000)
N = 15000
CALL DCORR_PERIODIC(A,B,C,N,STATUS)
```

This Fortran code computes the periodic correlation of two vectors of double-precision real numbers, a and b, with lengths of 15000. The result is stored in c with length of 15000.

_CONV_NONPERIODIC_EXT
Extended Nonperiodic Convolution

Format

status = {S,D,C,Z}CONV_NONPERIODIC_EXT (x, nx_stride, y, ny_stride, out, out_stride, nx, ny, n_out_start, n_out_end, add_flag, scale_flag, scale, scale_stride)

Arguments

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the data to be convolved.
On exit, **x** is unchanged.

nx_stride
integer*4
Distance between elements in the X array; *nx_stride* > 0

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the convolution or "filter" function that is to be convolved with the data from the X array.
On exit, **y** is unchanged.

ny_stride
integer*4
Distance between elements in the Y array; *ny_stride* > 0

out
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array OUT of length $nx + ny - 1$.
On exit, **out** contains the convolution coefficients.

out_stride
integer*4
Specifies the distance between elements in the OUT array; *out_stride* > 0

nx, ny
integer*4
Specifies the number of values to be operated on; $nx, ny > 0$

n_out_start, n_out_end
integer*4
Specifies the range of coefficients computed; *n_out_end* > *n_out_start*. The OUT array has zero values for indices less than 0 or greater than $nx + ny - 2$.

For example, in the case of nx = 50 and ny = 100, the range of locations is 0 through 148. If you specify *n_out_start* = 5 and *n_out_end* = 10, the convolution function generates numbers for OUT(5) through OUT(10) and puts the results in location 0 through 5 of the OUT array.

You can also specify a range that is larger than the array, creating null elements in the output array. For example, using the same input array, you can specify *n_out_start* = -10 and *n_out_end* = 200. The convolution function can generate values for 0 through 148, putting them in location 10 through 158 of the output array. It puts null elements in the locations between 0 and 9 and between 159 and 200.

add_flag
logical*4
Defines the operation of the convolution to add output to an existing OUT array, without overwriting it:

TRUE: Add the result of the operation to OUT array.
FALSE: Overwrite the existing OUT array.

scale_flag
logical*4
Defines the operation of the convolution to scale the output:

TRUE: Scale the output.
FALSE: Do not scale.

scale
real*4 | real*8 | complex*8 | complex*16
The value used to scale the output. The type of value depends on **scale_stride**.

scale_stride
integer*4
Defines how the scale operation is performed. *scale_stride*≥ 0:

= 0 : Scale by a scalar value
> 0: Scale by a vector, used as the stride of **scale**

Description

The _CONV_NONPERIODIC_EXT functions compute the nonperiodic convolution with options to control the result.

Return Values

0	DXML_SUCCESS()
8	DXML_ILL_N_RANGE()
13	DXML_BAD_STRIDE()

Example

```
  INCLUDE '/usr/include/DXMLDEF.FOR'
  INTEGER*4 N,STATUS
  REAL*8 A(50),B(100),C(6),SCALE_VALUE

  SCALE_VALUE = 2.0
  STATUS = DCONV_NONPERIODIC_EXT(A,1,B,1,C,1,50,100,5,10,.FALSE.,.TRUE.,
* SCALE_VALUE,0)
```

This Fortran code computes six values of a nonperiodic convolution of two vectors, C(5) to C(10), of double-precision real numbers, a and b, with lengths of 50 and 100, respectively. The result is scaled by 2.0 and stored in c with length of 6.

_CONV_PERIODIC_EXT
Extended Periodic Convolution

Format

status = {S,D,C,Z}CONV_PERIODIC_EXT (x, nx_stride, y, ny_stride, out, out_stride, n, n_out_start, n_out_end, add_flag, scale_flag, scale, scale_stride)

Arguments

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the data to be convolved.
On exit, **x** is unchanged.

nx_stride
integer*4
Distance between elements in the X array; *nx_stride* > 0

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the convolution or "filter" function that is to be convolved with the data from the X array.
On exit, **y** is unchanged.

ny_stride
integer*4
Distance between elements in the Y array; *ny_stride* > 0

out
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array OUT of length n.
On exit, **out** contains the convoluted data.

out_stride
integer*4
Specifies the distance between elements in the OUT array; *out_stride* > 0

n
integer*4
Specifies the number of values to be operated on; $n > 0$

n_out_start, n_out_end
integer*4
Specifies the range of coefficients computed; *n_out_end* > *n_out_start*. The OUT array has zero values for indices less than 0 or greater than $n-1$.

For example, in the case of n = 100, the locations range from 0 through 99. If you specify *n_out_start* = 5 and *n_out_end* = 10, the convolution function generates numbers for OUT(5) through OUT(10) and puts the results in location 0 through 5 of the OUT array.

You can also specify a range that is larger than the array. For example, using the same input array, you can specify *n_out_start* = -10 and *n_out_end* = 200. The convolution function can generate values OUT(0) through OUT(99), putting them

in location 10 through 109 of the output array. The locations outside of the range do not get null values; they are not affected.

add_flag
logical*4
Defines the operation of the convolution to add output to an existing OUT array, without overwriting it:

TRUE: Add the result of the operation to OUT array.
FALSE: Overwrite the existing OUT array.

scale_flag
logical*4
Defines the operation of the convolution to scale the output:

TRUE: Scale the output.
FALSE: Do not scale.

scale
real*4 | real*8 | complex*8 | complex*16
The value by which to scale the output.

scale_stride
integer*4
Defines how the scale operation is performed. *scale_stride*$\geq$ 0:

= 0 : Scale by a scalar value
> 0: Scale by a vector, used as the stride of **scale**

Description

The _CONV_PERIODIC_EXT functions compute the periodic convolution with options to control the result.

Return Values

0	DXML_SUCCESS()
8	DXML_ILL_N_RANGE()
13	DXML_BAD_STRIDE()

Example

```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N,STATUS
REAL*8 A(100),B(100),C(6),SCALE_VALUE

SCALE_VALUE = 2.0
STATUS = DCONV_PERIODIC_EXT(A,1,B,1,C,1,100,5,10,.FALSE.,.TRUE.,SCALE_VALUE,0)
```

This Fortran code computes six values of a periodic convolution of two vector, C(5) to C(10), of double-precision real numbers, a and b, with length of 100. The result is scaled by 2.0 and stored in c with length of 6.

_CORR_NONPERIODIC_EXT
Extended Nonperiodic Correlation

Format

status = {S,D,C,Z}CORR_NONPERIODIC_EXT (x, nx_stride, y, ny_stride, out, out_stride, nx, ny, n_out_start, n_out_end, add_flag, scale_flag, scale, scale_stride)

Arguments

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the data to be correlated.
On exit, **x** is unchanged.

nx_stride
integer*4
Distance between elements in the X array; *nx_stride* > 0

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the function that is to be correlated with the data from the X array.
On exit, **y** is unchanged.

ny_stride
integer*4
Distance between elements in the Y array; *ny_stride* > 0

out
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array OUT of length $nx + ny - 1$.
On exit, **out** contains the correlated data.

out_stride
integer*4
Specifies the distance between elements in the OUT array; *out_stride* > 0

nx, ny
integer*4
Specifies the number of values to be operated on; $nx, ny > 0$

n_out_start, n_out_end
integer*4
Specifies the range of coefficients computed; *n_out_end* > *n_out_start*. The OUT array has zero values for indices less than $1 - ny$ or greater than $nx - 1$.

For example, in the case of nx = 50 and ny = 100, the range of locations is -99 through 49. If you specify *n_out_start* = 5 and *n_out_end* = 10, the correlation function generates numbers for OUT(5) through OUT(10) and puts the results in location 0 through 5 of the OUT array.

You can also specify a range that is larger than the array, creating null elements in the output array. For example, using the same input array, you can specify *n_out_start* = -110 and *n_out_end* = 60. The correlation function can generate values for OUT(-99) through OUT(49), putting the results in locations 11 through 159. The locations 0 through 10 and 160 through 170 have no values, because OUT(-110) through OUT(-100) and OUT(50) through OUT(60) are out of range.

add_flag
logical*4
Defines the operation of the correlation to add output to an existing OUT array, without overwriting it:

TRUE: Add the result of the operation to OUT array.
FALSE: Overwrite the existing OUT array.

scale_flag
logical*4
Defines the operation of the correlation to scale the output:

TRUE: Scale the output.
FALSE: Do not scale.

scale
real*4 | real*8 | complex*8 | complex*16
The value by which to scale the output.

scale_stride
integer*4
Defines how the scale operation is performed. *scale_stride*$\geq$ 0:

= 0 : Scale by a scalar value
> 0: Scale by a vector, used as the stride of **scale**

Description

The _CORR_NONPERIODIC_EXT functions compute the nonperiodic correlation with options to control the result.

Return Values

0	DXML_SUCCESS()
8	DXML_ILL_N_RANGE()
13	DXML_BAD_STRIDE()

Example

```
  INCLUDE '/usr/include/DXMLDEF.FOR'
  INTEGER*4 N,STATUS
  REAL*8 A(50),B(100),C(6),SCALE_VALUE

  SCALE_VALUE = 2.0
  STATUS = DCORR_NONPERIODIC_EXT(A,1,B,1,C,1,50,100,-99,-94,
* .FALSE.,.TRUE.,SCALE_VALUE,0)
```

_CORR_NONPERIODIC_EXT

This Fortran code computes six values of a nonperiodic correlation of two vectors, C(-99) to C(-94), of double-precision real numbers, *a* and *b*, with lengths of 50 and 100, respectively. The result is scaled by 2.0 and stored in *c* with a length of 6.

_CORR_PERIODIC_EXT
Extended Periodic Correlation

Format

status = {S,D,C,Z}CORR_PERIODIC_EXT (x, nx_stride, y, ny_stride, out, out_stride, n, n_out_start, n_out_end, add_flag, scale_flag, scale, scale_stride)

Arguments

x
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the data to be correlated.
On exit, **x** is unchanged.

nx_stride
integer*4
Distance between elements in the X array; *nx_stride* > 0

y
real*4 | real*8 | complex*8 | complex*16
On entry, an array containing the function that is to be correlated with the data from the X array.
On exit, **y** is unchanged.

ny_stride
integer*4
Distance between elements in the Y array; *ny_stride* > 0

out
real*4 | real*8 | complex*8 | complex*16
On entry, a one-dimensional array OUT of length n.
On exit, **out** contains the correlated data.

out_stride
integer*4
Specifies the distance between elements in the OUT array; *out_stride* > 0

n
integer*4
Specifies the number of values to be operated on; $n > 0$

n_out_start, n_out_end
integer*4
Specifies the range of coefficients computed; *n_out_end* > *n_out_start*. The OUT array has zero values for indices less than 0 or greater than $n - 1$.

For example, in the case of n = 100, the locations range from 0 through 99. If you specify *n_out_start* = 5 and *n_out_end* = 10, the correlation function generates numbers for OUT(5) through OUT(10) and puts the results in location 0 through 5 of the OUT array.

You can also specify a range that is larger than the array. For example, using the same input array, you can specify *n_out_start* = -10 and *n_out_end* = 200. The convolution function can generate values OUT(0) through OUT(99), putting them

in location 10 through 109 of the output array. The locations outside of the range do not get null values; they are not affected.

add_flag
logical*4
Defines the operation of the function to add the output to an existing OUT array, without overwriting it:

TRUE: Add the result of the operation to OUT array.
FALSE: Overwrite the existing OUT array.

scale_flag
logical*4
Defines the operation of the function to multiply the output by a factor:

TRUE: Scale the output.
FALSE: Do not scale.

scale
real*4 | real*8 | complex*8 | complex*16
The value by which to scale the output.

scale_stride
integer*4
Defines how the scale operation is performed. *scale_stride* $\geq$ 0:

= 0 : Scale by a scalar value
> 0: Scale by a vector, used as the stride of **scale**

Description

The _CORR_PERIODIC_EXT functions compute the periodic correlation with options to control the result.

Return Values

0	DXML_SUCCESS()
8	DXML_ILL_N_RANGE()
13	DXML_BAD_STRIDE()

Example

```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N,STATUS
REAL*8 A(100),B(100),C(6),SCALE_VALUE

SCALE_VALUE = 2.0
STATUS = DCORR_PERIODIC_EXT(A,1,B,1,C,1,100,5,10,.FALSE.,.TRUE.,SCALE_VALUE,0)
```

This Fortran code computes six values of a periodic convolution of two vectors, C(5) to C(10), of double-precision real numbers, a and b, with length of 100. The result is scaled by 2.0 and stored in c with a length of 6.

Filter Subprograms

This section provides descriptions of the filter subroutines. The filter subroutines are ordered by the type of filter: the one-step filter subprogram, then the two-step filter subprograms.

SFILTER_NONREC
Nonrecursive Filter

The SFILTER_NONREC subroutine performs filtering in lowpass, highpass, bandpass, or bandstop (notch) mode.

Format

SFILTER_NONREC (in, out, n, flow, fhigh, wiggles, nterms, [status])

Arguments

in
REAL*4
On entry, a one-dimensional array IN containing the data to be filtered.
On exit, **in** is unchanged.

out
REAL*4
On entry, a one-dimensional array OUT containing the filtered data. The IN and OUT arrays can be the same array.
On exit, **out** is overwritten and contains the filtered data.

n
INTEGER*4
On entry, the number of values to be filtered; $n > 2$ and $n \geq 2(nterms) + 1$.
On exit, **n** is unchanged.

flow
REAL*4
On entry, the lower frequency of the filter, given as a fraction of the Nyquist sampling frequency $1/(2\Delta t)$; $0.0 \leq flow \leq 1.0$.
On exit, **flow** is unchanged.

fhigh
REAL*4
On entry, the upper frequency of the filter, given as a fraction of the Nyquist sampling frequency $1/(2\Delta t)$; $0.0 \leq fhigh \leq 1.0$.
On exit, **fhigh** is unchanged.

wiggles
REAL*4
On entry, a number in -dB units which is proportional to the oscillation from the Gibbs phenomenon; $0.0 \leq wiggles \leq 500.0$.
On exit, **wiggles** is unchanged.

nterms
INTEGER*4
On entry, the order of the filter, that is, the number of filter coefficients in the filter equation; $2 \leq nterms \leq 500$ and $n \geq 2(nterms) + 1$.
On exit, **nterms** is unchanged.

status
INTEGER*4
On entry, an optional variable with an unspecified value.
On exit, **status** is overwritten. When the subroutine is called, **status** is defined and its integer value describes the status of the operation. The following table shows the status function names, their associated integer values, and the status description associated with each integer:

Status Function	Value Returned	Description
DXML_SUCCESS()	0	Successful execution
DXML_MAND_ARG()	1	Mandatory argument is missing
DXML_ILL_WIGGLES()	5	wiggles is out of range
DXML_ILL_FLOW()	6	flow is equal to fhigh
DXML_ILL_F_RANGE()	7	flow or fhigh is out of range
DXML_ILL_N_RANGE()	8	n is out of range
DXML_ILL_N_NONREC()	9	n is less than (2*nterms+1)
DXML_ILL_NTERMS()	10	nterms is out of range

Description

The SFILTER_NONREC subroutine performs nonrecursive filtering in either lowpass, highpass, bandpass, or bandstop (notch) mode. See Table 11–16 for information on controlling the filter type with the **flow** and **fhigh** arguments.

Example

```
INCLUDE '/usr/include/DXMLDEF.FOR'
INTEGER*4 N, NTERMS, STATUS
REAL*4    SA(510), SB(510), FLOW, FHIGH, WIGGLES
FLOW =    0.0
FHIGH =   0.75
WIGGLES = 200.0
N =       200
NTERMS =  20
CALL SFILTER_NONREC(SA, SB, N, FLOW, FHIGH, WIGGLES, NTERMS, STATUS)
```

This Fortran code filters the 200 values in array SA in lowpass mode.

SFILTER_INIT_NONREC
Initialization Step for Nonrecursive Filter

The SFILTER_INIT_NONREC subroutine computes a working array that is used by DXML_FILTER_APPLY_NONREC.

Format

SFILTER_INIT_NONREC (n, flow, fhigh, wiggles, nterms, temp_array, [status])

Arguments

n
INTEGER*4
On entry, the number of values to be filtered; $n > 2$ and $n \geq 2(nterms) + 1$.
On exit, **n** is unchanged.

flow
REAL*4

On entry, the lower frequency of the filter, given as a fraction of the Nyquist sampling frequency $1/(2\Delta t)$; $0.0 \leq flow \leq 1.0$.
On exit, **flow** is unchanged.

fhigh
REAL*4
On entry, the upper frequency of the filter, given as a fraction of the Nyquist sampling frequency $1/(2\Delta t)$; $0.0 \leq fhigh \leq 1.0$.
On exit, **fhigh** is unchanged.

wiggles
REAL*4
On entry, a number in -dB units that is proportional to the oscillation from the Gibbs phenomenon; $0.0 \leq wiggles \leq 500.0$.
On exit, **wiggles** is unchanged.

nterms
INTEGER*4
On entry, the order of the filter, that is, the number of filter coefficients in the filter equation; $2 \leq nterms \leq 500$ and $n \geq 2(nterms) + 1$.
On exit, **nterms** is unchanged.

temp_array
REAL*4
On entry, a temporary array of length 510 used for temporary storage.
On exit, **temp_array** is overwritten and contains the internally-generated filter coefficients.

status
INTEGER*4
On entry, an optional variable with an unspecified value.
On exit, **status** is overwritten. When the subroutine is called, **status** is defined and its integer value describes the status of the operation. The following table shows the status function names, their associated integer values, and the status description associated with each integer:

Status Function	Value Returned	Description
DXML_SUCCESS()	0	Successful execution
DXML_MAND_ARG()	1	Mandatory argument is missing
DXML_ILL_WIGGLES()	5	wiggles is out of range
DXML_ILL_FLOW()	6	flow is equal to fhigh
DXML_ILL_F_RANGE()	7	flow or fhigh is out of range
DXML_ILL_N_RANGE()	8	n is out of range
DXML_ILL_N_NONREC()	9	n is less than (2*nterms+1)
DXML_ILL_NTERMS()	10	nterms is out of range

Description

The SFILTER_INIT_NONREC subroutine computes a working array that is used by the SFILTER_APPLY_NONREC subroutine. See Table 11–16 for information on controlling the filter type with the **flow** and **fhigh** arguments.

SFILTER_APPLY_NONREC
Application Step for Nonrecursive Filter

The SFILTER_APPLY_NONREC subroutine performs filtering in lowpass, highpass, bandpass, or bandstop (notch) mode by using the working array that was computed by SFILTER_INIT_NONREC.

Format

SFILTER_APPLY_NONREC (in, out, temp_array, [status])

Arguments

in
REAL*4
On entry, an array IN containing the data to be filtered.
On exit, **in** is unchanged.

out
REAL*4
On entry, a one-dimensional array OUT to contain the filtered data. The IN and OUT arrays can be the same array.
On exit, **out** is overwritten and contains the filtered data.

temp_array
REAL*4
On entry, the temporary array of length 510 used for temporary storage of the filter coefficients generated by the DXML_FILTER_INIT_NONREC subroutine.
On exit, **temp_array** is unchanged.

status
INTEGER*4
On entry, an optional variable with an unspecified value.
On exit, **status** is overwritten. When the subroutine is called, **status** is defined and its integer value describes the status of the operation. The following table shows the status function names, their associated integer values, and the status description associated with each integer:

Status Function	Value Returned	Description
DXML_SUCCESS()	0	Successful execution
DXML_MAND_ARG()	1	Mandatory argument is missing
DXML_ILL_TEMP_ARRAY()	2	temp_array is corrupted or incorrect

Description

The SFILTER_APPLY_NONREC subroutine uses the working array that was computed by SFILTER_INIT_NONREC for repeated filtering operations.

12

Using the Iterative Solvers for Sparse Linear Systems

DXML provides subprograms for the iterative solution of sparse linear systems of equations via preconditioned conjugate-gradient-like methods.

This chapter provides information on the following topics:

- Introduction to iterative solvers (Section 12.1)
- Interface to the iterative solver, including the concepts of matrix-free formulation of an iterative method and preconditioning (Section 12.2)
- Storage schemes for sparse matrices (Section 12.3.1)
- Types of preconditioners (Section 12.3.2)
- Iterative methods (Section 12.4)
- Naming conventions (Section 12.5)
- Iterative solver subroutine summary (Section 12.6)
- Error handling (Section 12.7)
- Hints on using the iterative solvers (Section 12.8)
- Examples of the use of iterative solvers (Section 12.9)

The reference descriptions of the subprograms for the iterative solvers are at the end of this chapter.

Many iterative solver subprograms have been parallelized for improved peformance on multiprocessor systems. For a list of these subprograms, parallel performance considerations, and information about using the parallel library, see Chapter 4.

12.1 Introduction

Many applications in science and engineering require the solution of linear systems of equations:

$$Ax = b \qquad (12–1)$$

where A is an n by n matrix and x and b are n vectors. Often, these systems occur in the innermost loop of the application, and for good overall performance of the application, it is essential that the linear system solver be efficient. Depending on the application, the system may be solved either once, or many times with different right-hand sides.

The linear systems of equations that arise from science and engineering applications are usually sparse, that is, the coefficient matrix A has a large number of zero elements. Substantial savings in compute time and memory requirements can be realized by storing and operating on only the nonzero elements of A. Solution techniques that exploit this sparsity of the matrix A are referred to as sparse solvers.

12.1.1 Methods for Solutions

Methods for the solution of linear systems of equations can be broadly classified into two categories:

- Direct Methods
  These methods first factor the coefficient matrix A into its triangular factors and then perform a forward and backward solve with the triangular factors to get the required solution. The solution is obtained in a finite number of operations, usually known apriori, and is guaranteed to be as accurate as the problem definition.

  See Chapter 13 for details about direct methods.

- Iterative Methods
  These methods start with an initial guess to the solution, and proceed to calculate solution vectors that approach the exact solution with each iteration. The process is stopped when a given convergence criterion is satisfied. The number of iteration steps required for convergence varies with the coefficient matrix, the initial guess and the convergence criterion—thus an apriori estimate of the number of operations is not possible.

  The convergence of iterative techniques is often accelerated by means of a preconditioner. Instead of solving (12–1), the iterative technique is applied to a system derived from the original system (12–1), with better convergence properties. As a result of preconditioning, the number of operations per iteration is increased, but the corresponding reduction in the number of iterations required for convergence usually leads to an overall reduction in the total time required for solution.

Currently, DXML provides iterative methods only for real double-precision data.

While direct methods for the solution of sparse linear systems are relatively well understood and their algorithms are for general purpose, iterative methods are not so well established and their algorithms tend to be more special purpose. It is well known that there is no general effective iterative algorithm for the solution of an arbitrary sparse linear system, only collections of algorithms each suitable for a particular class of problem. Additionally, there are no strict convergence theorems for some of the iterative methods, and choosing a good iterative technique and preconditioner, both in terms of its convergence properties and its performance on a given architecture, is more of an art than a science. This choice depends on various factors such as the problem being solved, the data structures used, the architecture of the machine, and the amount of memory available.

Despite these drawbacks, for certain classes of problems, an appropriate iterative technique can yield an approximation to the solution significantly faster than a direct method. Also, iterative methods typically require less memory than direct methods and hence can be the only means of solution for some large problems. In an attempt to compensate for the lack of robustness of any single iterative method and preconditioner, DXML provides a variety of methods and preconditioners. All these methods belong to the class of preconditioned conjugate-gradient-type

methods. A good introduction to these iterative methods and preconditioners is given in the bibliography in Appendix A.

12.2 Interface to the Iterative Solver

The interface to the iterative solver requires as input, the matrix *A*, the right hand side *b* and an initial guess to the solution. However, due to the large number of storage schemes in use for the storage of sparse matrices, it is not possible to provide an interface that allows all possible storage schemes. DXML resolves this issue by using the matrix-free formulation of the iterative method, as suggested in the proposed iterative standard [Ashby and Seager 1990].

All the iterative techniques provided in DXML refer to the coefficient matrix *A*, or matrices derived from it such as the preconditioner, only in the following three operations:

- Creation of the preconditioner
- Multiplication of the coefficient matrix by a vector
- Application of the preconditioner

The preconditioner is created from the coefficient matrix prior to a call to the iterative solver routine.

The matrix-free formulation of an iterative method separates the operations of matrix-vector product and the application of the preconditioner from the rest of the iterative solver by considering them as subroutines that are called by the iterative algorithm. These subroutines have a standard interface independent of the storage scheme used for the coefficient matrix. By writing the subroutines to provide the required functionality for the storage scheme under consideration, the same iterative solver can be used for matrices stored using different storage schemes.

While the matrix-free formulation has the advantage of making the iterative solver independent of the matrix storage format and the problem being solved, it has the disadvantage that you have to write the subroutines that perform the operations on the matrix and the preconditioner. To alleviate this disadvantage somewhat, DXML provides subroutines for the matrix-vector product, the creation of the preconditioners and the application of the preconditioners for a select set of storage schemes and preconditioners. Additionally, driver routine DITSOL_DRIVER is provided that simplifies these tasks.

Thus, you have the option of either storing the coefficient matrix in one of the storage schemes provided by DXML and using the subroutines provided, or using your own storage scheme for the matrix and writing the routines for the matrix operations. You also have the option of not storing either the coefficient matrix or the preconditioner, but instead providing the required functionality by some indirect means.

The examples in Section 12.9 illustrate the various ways in which the iterative solvers can be used. These examples, along with the hints in Section 12.8 on the use of the iterative solvers, explain the variety of options provided by DXML. If you are unfamiliar either with the issues involved in iterative methods or with the concept of matrix-free formulation of an iterative method, then the information in these sections should prove helpful.

The next sections further explain the implications of the use of a matrix-free formulation of an iterative method and describe the interface for the iterative solver and the routines that provide the matrix operations. To keep the discussion general, the iterative solver routine is referred to as SOLVER — this is a generic name and in practice you would call the iterative solver routine by a name reflective of the iterative method.

12.2.1 Matrix-Vector Product

The iterative solvers provided in DXML require the evaluation of matrix-vector products of the form:

$$v = A * u$$

or:

$$v = A^T * u$$

where u and v are vectors of length n. This functionality is provided to the routine SOLVER via the subroutine MATVEC, which has the following standard parameter list:

```
MATVEC (JOB, IPARAM, RPARAM, A, IA, W, U, V, N)
```

Table 12–1 describes each parameter and its data type.

Table 12–1 Parameters for the MATVEC Subroutine

Argument Data Type	Description
job integer*4	On entry, defines the operation to be performed: **job** = 0 : $v = A * u$ **job** = 1 : $v = A^T * u$ **job** = 2 : $v = w - A * u$ **job** = 3 : $v = w - A^T * u$ On exit, **job** is unchanged.
iparam integer*4	On entry, a one-dimensional array of length at least 50, containing the parameters passed to the routine SOLVER. See Table 12–4 for details. On exit, the first 50 elements of IPARAM are unchanged.
rparam real*8	On entry, a one-dimensional array of length at least 50, containing the parameters passed to the routine SOLVER. See Table 12–5 for details. On exit, the first 50 elements of RPARAM are unchanged.
a real*8	On entry, a one-dimensional array for the nonzero elements of the matrix A. If MATVEC does not require this matrix to be explicitly stored, **a** is a dummy argument. Array A can be used to provide workspace. On exit, any information related to matrix A is unchanged.
ia integer*4	On entry, a one-dimensional array for auxiliary information about the matrix A or the array A. If this information is not needed, **ia** is a dummy argument. Array IA can be used to provide workspace. On exit, ia is unchanged.
w real*8	On entry, a one-dimensional array of length at least n that contains the vector w when **job** = 2 or 3. The elements of array W are accessed with unit increment. On exit, **w** is unchanged.

(continued on next page)

Table 12–1 (Cont.) Parameters for the MATVEC Subroutine

Argument Data Type	Description
u real*8	On entry, a one-dimensional array of length at least n that contains the vector u. The elements of array U are accessed with unit increment. On exit, **u** must be unchanged.
v real*8	On entry, a one dimensional array of length at least n. On exit, array V contains the vector defined by **job**. The elements of array V are accessed with unit increment.
n integer*4	On entry, the order of the matrix A. On exit, **n** is unchanged.

The routine MATVEC is an input parameter to the routine SOLVER and should be declared external in your calling program. It could either provide the required functionality itself, or act as an interface to a routine that provides the required functionality. Suppose the iterative solver requires MATVEC to provide the functionality for **job** = 0. Then you would write the routine MATVEC as follows:

```
SUBROUTINE MATVEC(JOB,IPARAM,RPARAM,A,IA,W,U,V,N)
        .
        .
        . (any initializations here)
        .     if necessary
        .
        .
IF (JOB.EQ.0) CALL USER_MATVEC(...)
RETURN
END
```

where USER_MATVEC is your routine for evaluating the vector $A * u$ and returning the result in vector v. This enables you to have a routine USER_MATVEC, which has a parameter list different from the parameter list of MATVEC. It also allows you to call one of the matrix-vector product routines included in DXML instead of USER_MATVEC, provided you have stored the coefficient matrix using one of DXML's sparse matrix storage schemes. In either case, you have to provide the routine MATVEC, with the required standard parameter list. If you use a storage scheme for the coefficient matrix different from those provided by DXML, or choose not to store the matrix at all, then it is your responsibility to provide the functionality required by MATVEC. If, however, you use a storage scheme provided by DXML for the coefficient matrix, then you provide a routine MATVEC that essentially calls the appropriate DXML routine.

The examples in Section 12.9 illustrate the different ways in which the interface provided by the routine MATVEC can be used to provide the required functionality. The reference descriptions at the end of this chapter describe the matrix-vector product routines for the various storage schemes supported by DXML.

12.2.2 Preconditioning

Preconditioning is a technique used for improving the convergence of an iterative method by applying the method to a system derived from the original with better convergence properties. The convergence of the iterative methods provided in DXML depends on the condition number and the distribution of the eigenvalues of the coefficient matrix A. A distribution where the eigenvalues are clustered is favorable for fast convergence of the iterative method.

A preconditioned iterative method applies the iterative method to an equivalent system derived from (12–2) as follows:

$$Q_L^{-1} * A * Q_R^{-1} * Q_R * x = Q_L^{-1} * b$$

that is:

$$A' * x' = b' \tag{12–2}$$

where:

$$A' = Q_L^{-1} * A * Q_R^{-1}$$

$$x' = Q_R * x$$

and:

$$b' = Q_L^{-1} * b$$

Q_L and Q_R are n by n matrices. The matrix $Q = Q_L * Q_R$ is called the preconditioning matrix or the preconditioner. The matrices Q_L and Q_R are derived from the coefficient matrix A such that the matrix A' is close to the identity matrix and thus has eigenvalues clustered around unity. As a result of preconditioning, the iterative method applied to (12–2), with the coefficient matrix A', right hand side b' and solution x', usually converges faster than when it is applied to (12–1).

The implementation of the iterative method for solving (12–2) does not involve the explicit formation of A'. Instead, the matrices Q_L, Q_R and A appear in operations of the form:

$$v = Q_R^{-1} * u$$

$$v = A * u$$

and:

$$v = Q_L^{-1} * u$$

The computation per iteration, in the preconditioned case, is more expensive than in the unpreconditioned case. This increase in the computation per iteration is usually offset by a reduction in the number of iterations required for convergence, leading to a reduction in the total computation.

The matrices Q_L and Q_R form a good preconditioner if they satisfy the following properties:

- Q is a good approximation to A so that A' is close to the identity matrix.
- Q_R and Q_L are easily obtainable.
- Solving systems of the form $Q_L * u = v$ or $Q_R * u = v$ is easy as the preconditioner is applied via the solution of these systems.
- The storage costs of Q_L and Q_R are not excessive.

Preconditioners can be divided into three broad classes depending on the manner in which they are applied:

- Left preconditioning:
$$(Q_L^{-1} * A) * x = (Q_L^{-1} * b)$$
- Right preconditioning:
$$(A * Q_R^{-1}) * (Q_R * x) = b$$

- Split preconditioning:

$$(Q_L^{-1} * A * Q_R^{-1}) * (Q_R * x) = (Q_L^{-1} * b)$$

Left and right preconditioning can be considered as special cases of split preconditioning with Q_R and Q_L being the identity matrices, respectively.

In the case of left preconditioning, the residual of the system (12–2), evaluated by the iterative method is not the same as the residual of the original system (12–1). The unpreconditioned residual, r, and the preconditioned residual, r', are related as follows:

$$r' = Q_L^{-1} * (b - A * x) = Q_L^{-1} * r$$

Similarly, the solution x' evaluated using right preconditioning is related to the true solution, x, as follows:

$$x' = Q_R * x$$

It follows that in the case of split preconditioning, neither the true solution nor the true residual are obtained directly from the application of the iterative technique to (12–2). Thus, the use of preconditioning implies that extra computation has to be done to recover the true residual and solution from the residual and solution of the equivalent system (12–2). However, there is a special case of split preconditioning that allows the iterative method to obtain the true residual and the true solution directly, when applied to a symmetric positive definite (SPD) matrix A. This is the case where the preconditioner is symmetric positive definite as well and hence can be written as:

$$Q = B * B^T$$

for some matrix B, that is:

$$Q_L = Q_R^T = B.$$

The interface to the preconditioning operations is provided by the routines PCONDL and PCONDR for left and right preconditioning, respectively. The two routines have similar parameter lists, differing only in the matrices Q_L and Q_R:

```
SUBROUTINE PCONDR (JOB, IPARAM, RPARAM, QR, IQR, A, IA, W, U, V, N)
SUBROUTINE PCONDL (JOB, IPARAM, RPARAM, QL, IQL, A, IA, W, U, V, N)
```

Table 12–2 describes each parameter and its data type.

Table 12–2 Parameters for the PCONDR and PCONDL Subroutines

Argument Data Type	Description
job integer*4	On entry, defines the operation to be performed: **job** = 0 : $v = Q_R^{-1} * u$ **job** = 1 : $v = Q_R^{-T} * u$ **job** = 2 : $v = w - Q_R^{-1} * u$ **job** = 3 : $v = w - Q_R^{-T} * u$ On exit, **job** is unchanged.

(continued on next page)

Table 12–2 (Cont.) Parameters for the PCONDR and PCONDL Subroutines

Argument Data Type	Description
iparam integer*4	On entry, a one-dimensional integer array of length at least 50, containing the parameters passed to the routine SOLVER. See Table 12–4 for details. On exit, the first 50 elements of IPARAM are unchanged.
rparam real*8	On entry, a one-dimensional real array of length at least 50, containing the parameters passed to the routine SOLVER. See Table 12–5 for details. On exit, the first 50 elements of RPARAM are unchanged.
qr real*8	On entry, a one-dimensional array for the nonzero elements of the matrix Q_R. If PCONDR does not require this matrix to be explicitly stored, **qr** is a dummy argument. **qr** can also be used to provide workspace for the routine PCONDR. On exit, any information related to the matrix Q_R is unchanged.
iqr integer*4	On entry, a one-dimensional array for auxiliary information about the matrix Q_R or the array QR. If no information is required, **iqr** is a dummy argument. **iqr** can also be used to provide workspace. On exit, information related to the matrix Q_R or the array QR is unchanged.
a real*8	On entry, a one-dimensional array for the nonzero elements of the matrix A. If PCONDR does not require the matrix A to be explicitly stored, **a** is a dummy argument. Array A can also be used to provide workspace. On exit, any information related to the matrix A is unchanged.
ia integer*4	On entry, a one-dimensional array for auxiliary information about the matrix A or the array A. If no information is needed, **ia** is a dummy argument. Array IA can also be used to provide workspace. On exit, any information related to the matrix A or the array A is unchanged.
w real*8	On entry, a one-dimensional array of length at least n that contains the vector w when **job** = 2 or 3. The elements of array W are accessed with unit increment. On exit, **w** is unchanged.
u real*8	On entry, a one-dimensional array of length at least n that contains the vector u. The elements of array U are accessed with unit increment. On exit, **u** must be unchanged.
v real*8	On entry, a one-dimensional array of length at least n. On exit, array V contains the vector defined by **job**. The elements of array V are accessed with unit increment.
n integer*4	On entry, the order of the matrix A. On exit, **n** is unchanged.

PCONDL and PCONDR are input parameters to SOLVER and, if used, must be declared external in your program. If only one of these preconditioning options is used, then the argument for the other is a dummy input parameter to SOLVER. If no preconditioning is used, both PCONDL and PCONDR are dummy parameters. The routines PCONDL and PCONDR are called by the iterative solver only if you request their use by setting an appropriate parameter for preconditioning, as explained in Section 12.2.4. This implies that you do not have to provide dummy routines for either PCONDL or PCONDR if they are not being used by SOLVER.

The preconditioning routines, PCONDL and PCONDR, only apply the preconditioner; you are responsible for setting up the preconditioner before the call to the routine SOLVER. The pointers to the matrix A, that is, arrays A and IA are passed to the preconditioner for use by those routines that are dependent on A, such as polynomial preconditioners. If the preconditioning

routine does not use A, both A and IA may be dummy arguments. Any workspace for use by the preconditioner can be passed through the arrays QL, QR, IQL, and IQR.

In the case of split SPD preconditioning, the iterative technique requires the solution of a system of the form $Q * u = v$. An explicit split of the preconditioning matrix Q into Q_L and Q_R is not required. As a result, only one of the routines PCONDL or PCONDR is needed. DXML provides the required functionality through the routine PCONDL, that is, PCONDL provides the solution of the system $Q * u = v$ when the **job** argument is set to zero and PCONDR is not used.

If the iterative solver is called with preconditioning, then you must provide the appropriate routines PCONDL and PCONDR with the standard parameter list. As in the case of MATVEC, the required functionality could be provided either by your own routines or by calls to the appropriate DXML routines from within PCONDL and PCONDR. The examples in Section 12.9 illustrate the different ways in which the interface provided by the routines PCONDL and PCONDR can be used to provide the required functionality. Section 12.3.2 and the reference descriptions at the end of this chapter describe the routines for creating and applying the preconditioners for the various storage schemes supported by DXML.

12.2.3 Stopping Criterion

An important aspect of any iterative solver is the stopping criterion, that is, the conditions that determine when the iterations are stopped. The stopping criterion usually has two parts: a quantity that is measured and a positive constant ϵ that the quantity is measured against to determine convergence. The choice of each is crucial as the stopping criterion should accurately reflect when a suitable approximation to the solution has been obtained.

In order for the evaluation of the stopping criterion to form a small fraction of the total computation, it should be easy to obtain the quantity that is measured from the iterative technique. Additionally, the constant ϵ must be chosen appropriately. A large value might result in a solution that does not have the required accuracy, while a small value might imply extra computation to achieve unneeded extra accuracy. Setting epsilon too small might also prevent the stopping criterion from ever being satisfied.

DXML provides you with the option of either writing your own stopping criterion or using one of the stopping criteria provided. The latter are based on the residual, r_i, of the system (12–1) at the i-th step of the iteration:

$$r_i = b - A * x_i$$

and the preconditioned residual, r_i', of the system (12–2) at the i-th step of the iteration:

$$r_i' = Q_L^{-1} * (b - A * x_i)$$

The four stopping criteria provided by DXML are as follows:

- Stopping Criterion 1:

$$||r_i||_2 \leq \epsilon \qquad (12–3)$$

- Stopping Criterion 2:

$$\frac{||r_i||_2}{||b||_2} \leq \epsilon \qquad (12–4)$$

- Stopping Criterion 3:

$$||r_i'||_2 \leq \epsilon \tag{12–5}$$

- Stopping Criterion 4:

$$\frac{||r_i'||_2}{||b'||_2} \leq \epsilon \tag{12–6}$$

where:

$$r_i' = Q_L^{-1} * r_i$$

$$b' = Q_L^{-1} * b$$

and:

$$||r_i||_2 = \sqrt{\sum_{j=1}^{n} r_i^2(j)}$$

You can choose one of these stopping criteria by setting the value of an appropriate input parameter as explained in Section 12.2.3. In the case of an unpreconditioned iterative technique, the choice of stopping criterion (12–5) or (12–6) defaults to (12–3) or (12–4), respectively. Stopping criteria (12–4) and (12–6) require that the denominator should be nonzero to avoid a division by zero error. Some stopping criteria might require more computation than others depending on what can be easily obtained from the iterative technique. For example, the left hand side of (12–3) is calculated during the unpreconditioned conjugate gradient technique and hence very little extra computation is needed for the evaluation of (12–3) or (12–4).

DXML also provides you with the option of implementing your own stopping criterion via the routine MSTOP, which has the following standard parameter list:

```
SUBROUTINE MSTOP (IPARAM, RPARAM, X, R, Z, B, N)
```

Table 12–3 describes each parameter and its data type.

Table 12–3 Parameters for the MSTOP Subroutine

Argument Data Type	Description
iparam integer*4	On entry, a one-dimensional array of length at least 50, containing the parameters passed to the routine SOLVER. See Table 12–4 for details. On exit, the first 50 elements of IPARAM are unchanged.
rparam real*8	On entry, a one-dimensional array of length at least 50, containing the parameters passed to the routine SOLVER. See Table 12–5 for details. On exit, the first 50 elements of RPARAM are unchanged, with the exception of RPARAM(2) that contains the left side of the stopping criterion as evaluated by MSTOP.
x real*8	On entry, a one-dimensional array of length at least n that contains the approximation to the solution obtained at the iteration number, $iters$ in IPARAM(10). On exit, **x** is unchanged.

(continued on next page)

Table 12–3 (Cont.) Parameters for the MSTOP Subroutine

Argument Data Type	Description
r real*8	On entry, a one-dimensional array of length at least n that contains the true residual of the system (12–1) obtained at the iteration number, $iters$ in IPARAM(10). See the reference descriptions of the iterative solvers for conditions under which **r** is defined. On exit, **r** must be unchanged.
z real*8	On entry, a one-dimensional array of length at least n that contains the preconditioned residual of the system (12–2) obtained at the iteration number, $iters$ in IPARAM(10). See the reference descriptions of the iterative solvers for conditions under which **z** is defined. On exit, **z** is unchanged.
b real*8	On entry, a one-dimensional real array of length at least n that contains the right-hand side of the system (12–1). On exit, **b** must be unchanged.
n integer*4	On entry, the order of the matrix A. On exit, **n** is unchanged.

By providing an interface to the routine MSTOP, DXML allows you to use a stopping criterion derived from the vectors x, r, z and b, which is different from the standard stopping criteria provided by DXML. Based on the input parameters, the routine MSTOP should evaluate the left side of the convergence test and return the value in RPARAM(2) as explained in Section 12.2.4. The iteration count parameter, iters (IPARAM(10)), allows you to evaluate quantities depending upon the iteration, such as the initial residual norm, or print out information on each iteration. MSTOP is an input parameter to the routine SOLVER and, if used, should be declared external in your calling program.

12.2.4 Parameters for the Iterative Solver

The interface to the iterative solver provided by DXML allows you to pass information to the solver such as the maximum number of iterations allowed and the I/O unit number for output, as well as to obtain information back from the solver such as the number of iterations required for convergence and error messages. This information is passed via two arrays — IPARAM for integer parameters and RPARAM for real parameters. These arrays are of length at least 50, with the first 30 elements reserved for use by the proposed standard and next 20 for use by DXML. Tables 12–4 and 12–5 describe the elements of the parameter arrays.

Table 12–4 Integer Parameters for the Iterative Solver

Parameter	Variable	Description
IPARAM(1)	$nipar$	Length of the array IPARAM, ≥ 50.
IPARAM(2)	$nrpar$	Length of the array RPARAM, ≥ 50.
IPARAM(3)	$niwk$	Length of the array IWORK, size varies with iterative solvers.
IPARAM(4)	$nrwk$	Length of the array RWORK, size varies with iterative solvers.

(continued on next page)

Table 12–4 (Cont.) Integer Parameters for the Iterative Solver

Parameter	Variable	Description
IPARAM(5)	*iounit*	I/O unit for providing information generated by SOLVER. If *iounit* > 0, output is written to UNIT = iounit, which must be opened in the calling program. If *iounit* < 0, no output is generated by the routine SOLVER.
IPARAM(6)	*iolevel*	Determines the kind of information output when *iounit* > 0: *iolevel* $= 0$: Fatal error messages only *iolevel* $= 1$: Warning messages and minimum output *iolevel* $= 2$: Reasonable summary *iolevel* ≥ 3 : More detailed information
IPARAM(7)	*ipcond*	Determines the form of preconditioning: *ipcond* $= 0$: no preconditioning; IQR, IQL, QR, QL, PCONDL, PCONDR are dummy arguments *ipcond* $= 1$: left preconditioning; IQR, QR, PCONDR are dummy arguments *ipcond* $= 2$: right preconditioning; IQL, QL, PCONDL are dummy arguments *ipcond* $= 3$: split preconditioning *ipcond* $= 4$: SPD split preconditioning; IQR, QR, PCONDR are dummy arguments
IPARAM(8)	*istop*	Determines the stopping criterion: *istop* $= 0$: user-supplied routine MSTOP *istop* $= 1$: stopping criterion (12–3) (default) *istop* $= 2$: stopping criterion (12–4) *istop* $= 3$: stopping criterion (12–5); if no preconditioning used, defaults to *istop* $= 1$ *istop* $= 4$: stopping criterion (12–6); if no preconditioning used, defaults to *istop* $= 2$
IPARAM(9)	*itmax*	Maximum number of iterations allowed for convergence. If convergence is not achieved in *itmax* iterations the solver returns with error flag set. Default = 100
IPARAM(10)	*iters*	Number of iterations required to satisfy the convergence criterion.
IPARAM(31)	*nz*	Parameter related to the number of nonzeros stored for the matrix. See the storage schemes in Section 12.3.1 for more details.
IPARAM(32)	*ndim*	Leading dimension of 2 dimensional arrays. See the storage schemes in Section 12.3.1 for more details.
IPARAM(33)	*ndeg*	Degree of the polynomial used for polynomial preconditioning. Default = 1
IPARAM(34)	*kprev*	Number of previous residual vectors used in the iterative solver DITSOL_PGMRES. See the reference descriptions for details.

(continued on next page)

Table 12–4 (Cont.) Integer Parameters for the Iterative Solver

Parameter	Variable	Description
IPARAM(35)	$istore$	Storage scheme used in the driver routine: $istore = 1$: SDIA storage scheme, lower triangular part is stored (Section 12.3.1.1) $istore = 2$: SDIA storage scheme, upper triangular part is stored (Section 12.3.1.1) $istore = 3$: UDIA storage scheme (Section 12.3.1.2) $istore = 4$: GENR storage scheme (Section 12.3.1.3)
IPARAM(36)	$iprec$	Preconditioner used in the driver routine: $iprec = 1$: diagonal preconditioner (Section 12.3.2.1) $iprec = 2$: polynomial preconditioner (Section 12.3.2.2) $iprec = 3$: ILU preconditioner (Section 12.3.2.3)
IPARAM(37)	$isolve$	Iterative solver used in the driver routine: $isolve = 1$: Conjugate gradient method $isolve = 2$: Least squares conjugate gradient method $isolve = 3$: Bi-conjugate gradient method $isolve = 4$: Conjugate gradient squared method $isolve = 5$: Generalized minimum residual method $isolve = 6$: Transpose-free quasiminimal residual method

Table 12–5 Real Parameters for the Iterative Solver

Parameter	Variable	Description
RPARAM(1)	$errtol(\epsilon)$	User-supplied tolerance for convergence.
RPARAM(2)	$stptst$	Quantity that determines the convergence of the iterative technique. (lefthand side of stopping criterion)

The elements of the arrays IPARAM and RPARAM not defined in Table 12–4 and Table 12–5 have no variable assigned to them at present, but are reserved for future use by DXML. The arrays IPARAM and RPARAM are passed to the routine SOLVER and all the routines called by it (MATVEC, PCONDL, PCONDR and MSTOP). If necessary, you can use these arrays to pass additional information to these routines. For example, if you declare these arrays to be of dimension 100, then the elements from 51 to 100 can be used to pass information to the routines MATVEC, PCONDL, PCONDR and MSTOP. However, the first 50 elements are for the exclusive use of the proposed standard and DXML.

DXML allows you to set the variables in the IPARAM and RPARAM arrays to their default values by a call to the routines DITSOL_DEFAULTS with the following interface:

```
SUBROUTINE DITSOL_DEFAULTS (IPARAM, RPARAM)
```

Table 12–6 defines the default values set by the routine DITSOL_DEFAULTS. After a call to DITSOL_DEFAULTS, you can change any of the parameters as required. It is your responsibility to ensure that the variables in the arrays IPARAM and RPARAM have been assigned appropriate values before a call to the iterative solver routine. The examples in Section 12.9 illustrate the use of the routine DITSOL_DEFAULTS further.

Table 12–6 Default Values for Parameters

Parameter	Variable	Default Value
iparam(1)	*nipar*	50
iparam(2)	*nrpar*	50
iparam(5)	*iounit*	6
iparam(6)	*iolevel*	0
iparam(7)	*ipcond*	0
iparam(8)	*istop*	1
iparam(9)	*itmax*	100
iparam(33)	*ndeg*	1
rparam(1)	*errtol*	1.0e-6

12.2.5 Argument List for the Iterative Solver

The matrix-free formulation of the iterative method adopted by DXML implies that the routine SOLVER has, as input parameters, the routines MATVEC, PCONDL, PCONDR, and MSTOP. Additionally, the parameter list also contains the arrays A, IA, QL, IQL, QR, and IQR for use by the routines for the matrix operations (MATVEC, PCONDL, PCONDR) as well as input parameters such as the size of the system, the right side, and the initial approximation. The approximation to the solution obtained by the solver is returned in the vector x. Real and integer workspace is provided via arrays RWORK and IWORK, respectively and real and integer parameters via arrays RPARAM and IPARAM, respectively. This results in the following interface for the routine SOLVER:

```
SUBROUTINE SOLVER (MATVEC, PCONDL, PCONDR, MSTOP, A, IA, X, B, N, QL, IQL,
                   QR, IQR, IPARAM, RPARAM, IWORK, RWORK, IERROR)
```

Table 12–7 describes each parameter and its data type.

Table 12–7 Parameters for the SOLVER Subroutine

Argument Data Type	Description
matvec character	On entry, a user-supplied name for the routine that evaluates the matrix-vector product. **matvec** uses the standard interface and must be declared external in your calling program. See Table 12–1. On exit, **matvec** is unchanged.
pcondl character	On entry, a user-supplied name for the routine that applies left preconditioning. **pcondl** uses the standard interface. See Table 12–2. If left preconditioning is not used, **pcondl** is a dummy parameter. If used, **pcondl** must be declared external in your calling program. The variable *ipcond* in IPARAM(7), must be set appropriately for access to the PCONDL routine. On exit, **pcondl** is unchanged.

(continued on next page)

Table 12–7 (Cont.) Parameters for the SOLVER Subroutine

Argument Data Type	Description
pcondr character	On entry, a user-supplied name for the routine that applies right preconditioning. **pcondr** uses the standard interface. See Table 12–2. If right preconditioning is not used, **pcondr** is a dummy parameter. If used, **pcondr** must be declared external in your calling program. The variable *ipcond*, IPARAM(7), must be set appropriately for access to the PCONDR routine. On exit, **pcondr** is unchanged.
mstop character	On entry, a user-supplied name for the routine that evaluates a stopping criterion defined by you. **mstop** has the standard interface. See Table 12–3. If used, it must be declared external in your calling program. The variable *istop*, IPARAM(8), must be set appropriately for access to the MSTOP routine. If not used, that is, you use one of the DXML stopping criteria, **mstop** is a dummy parameter. On exit, **mstop** is unchanged.
a real*8	On entry, a one-dimensional array for the nonzero elements of matrix A. If the MATVEC, PCONDL and PCONDR routines do not require this matrix to be explicitly stored, array A may be a dummy array. Array A may also be used to provide workspace. On exit, any information related to the matrix A is unchanged.
ia integer*4	On entry, a one-dimensional array for auxiliary information about the matrix A, or the array A. If the information is not needed, array IA may be a dummy array. Array IA may also be used to provide workspace. On exit, any information related to the matrix A or the array A is unchanged.
x real*8	On entry, a one-dimensional array of length at least n that contains the initial approximation to the solution. The elements of array X are accessed with unit increment. On exit, **x** is the approximation to the solution obtained by the routine SOLVER.
b real*8	On entry, a one-dimensional array of length at least n that contains the right side of the system (12–1). The elements of array B are accessed with unit increment. On exit, **b** is unchanged.
n integer*4	On entry, the order of the matrix A. On exit, **n** is unchanged.
ql real*8	On entry, a one-dimensional array for the nonzero elements of the matrix Q_L. If PCONDL does not require this matrix to be explicitly stored, **ql** is a dummy array. Array QL may also be used to provide workspace. On exit, any information related to the matrix Q_L is unchanged.
iql integer*4	On entry, a one-dimensional array for auxiliary information about the matrix Q_L or the array QL. If no information is needed, IQL is a dummy array. Array IQL can also be used to provide workspace. On exit, any information related to the matrix Q_L or the array QL is unchanged.
qr real*8	On entry, a one-dimensional array for the nonzero elements of the matrix Q_R. If PCONDR does not require this matrix to be explicitly stored, QR is a dummy array. QR can also be used to provide workspace. On exit, any information related to matrix Q_R is unchanged.

(continued on next page)

Table 12–7 (Cont.) Parameters for the SOLVER Subroutine

Argument Data Type	Description
iqr integer*4	On entry, a one-dimensional array for auxiliary information about the matrix Q_R or the array QR. If no information is needed, IQR is a dummy array. Array IQR can also be used to provide workspace. On exit, any information related to the matrix Q_R or the array QR is unchanged.
iparam integer*4	On entry, a one-dimensional array of length at least 50, containing the parameters passed to the routine SOLVER. On exit, the first 50 elements of IPARAM are unchanged, with the exception of *iters* (IPARAM(10)), which is then equal to the number of iterations required for convergence.
rparam real*8	On entry, a one-dimensional array of length at least 50, containing the parameters passed to the routine SOLVER. On exit, the first 50 elements of rparam are unchanged, with the exception of RPARAM(2) that is equal to the left side of the stopping criterion.
iwork integer*4	On entry, a one-dimensional array of length *niwk*, IPARAM(3), used as an integer workspace. On exit, the data in IWORK is overwritten.
rwork real*8	On entry, a one-dimensional array of length *nrwk*, IPARAM(4), used as a real workspace. On exit, the data in RWORK is overwritten.
ierror integer*4	On entry, a scalar value that receives the value of the error flag. On exit, the error flag returned by the routine SOLVER.

In addition to the error messages output by the iterative solver, based on the variable, iolevel (IPARAM(6)), the routine SOLVER also returns an error flag, ierror. It is your responsibility to check the error flag on exit from SOLVER and ensure that the solution procedure ended normally. This is especially true if you have disabled all error messages by setting a negative value for iounit (IPARAM(5)).

12.3 Matrix Operations

The matrix-free formulation of the iterative method adopted by DXML isolates the operations involving the coefficient matrix, A, and the preconditioning matrices, Q_L and Q_R, by separating them into subroutines that form the matrix-vector product, create the preconditioner and apply the preconditioner. This formulation allows you to use any storage scheme for storing the coefficient matrix and the preconditioner, but has the drawback that the routines MATVEC, PCONDL, PCONDR, and the routine for the creation of the preconditioner have to be written by you.

As an alternative, DXML provides you with the option of using routines written to implement the matrix operations for three matrix storage schemes. In addition to the matrix-vector product operations, DXML provides routines for the creation and application of three preconditioners for each of the three storage schemes. Calls to these DXML routines can be used in the routines MATVEC, PCONDL, and PCONDR to implement the desired operation. The only restriction is that the coefficient matrix be stored in one of the storage schemes provided by DXML.

12.3.1 Storage Schemes for Sparse Matrices

A sparse matrix is a matrix that has very few nonzero elements. By storing and operating on only the nonzero elements, it is possible to achieve substantial savings in memory requirements and computation. In addition to the nonzero elements, storage is also required for information that determines the position of each nonzero element in the matrix.

Sparse matrices can be broadly classified as either structured or unstructured matrices. A structured sparse matrix is one where the distribution of nonzero elements in the matrix has a specific structure. For example, the matrix A shown in (12–7) has its nonzero elements along the diagonals of the matrix:

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ 0 & a_{22} & a_{23} & 0 & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} & 0 & 0 \\ a_{41} & 0 & 0 & a_{44} & a_{45} & 0 \\ 0 & a_{52} & 0 & 0 & a_{55} & a_{56} \\ 0 & 0 & a_{63} & 0 & 0 & a_{66} \end{bmatrix} \tag{12–7}$$

This structure can be exploited to reduce the amount of additional information that is necessary for the determination of the position of the nonzero elements within the matrix. For example, consider the elements in the superdiagonal of the matrix A:

$$\begin{pmatrix} a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \end{pmatrix}$$

As the elements all lie on a diagonal, the position of each element relative to the previous element is known (that is, the row and column indices of an element on the diagonal are one higher than the row and column indices of the preceding element in the diagonal). If the row and column indices of the first element a_{12} are known, the positions of the other elements in the diagonal are also known. Thus, substantial savings in the storage requirements can be achieved by storing only the position of the first element in each diagonal.

Unstructured sparse matrices, such as the matrix A in (12–8), do not have a structure to the distribution of the nonzero elements:

$$A = \begin{bmatrix} a_{11} & 0 & a_{13} & 0 & 0 \\ 0 & a_{22} & a_{23} & 0 & 0 \\ 0 & 0 & a_{33} & 0 & 0 \\ a_{41} & 0 & 0 & a_{44} & a_{45} \\ 0 & 0 & a_{53} & 0 & a_{55} \end{bmatrix} \tag{12–8}$$

In such cases, each nonzero element is stored along with its row and column indices. Some savings in storage is possible if the elements in a row (or column) are stored contiguously. In such cases, only the row (or column) index for the first nonzero element of the row (or column) is stored.

DXML subprograms that operate on sparse matrices store them in one of three ways:

- Symmetric diagonal storage
- Unsymmetric diagonal storage
- General storage by rows

12.3.1.1 SDIA: Symmetric Diagonal Storage Scheme

Symmetric matrices whose nonzero elements lie along a few diagonals can be stored using a scheme that stores only the diagonals and the distance of each diagonal from the main diagonal. Each diagonal is stored in its entirety, along with any zeros. The distance of a diagonal from the main diagonal is positive if the diagonal is above the main diagonal and negative if the diagonal is below the main diagonal. The main diagonal itself has a distance of zero.

An n by n matrix is stored in a two dimensional array with at least n rows and as many columns as the number of nonzero diagonals in the strict upper (or lower) triangular part plus 1 (for the main diagonal). As the matrix is symmetric, either the upper or lower triangular part can be stored. Both the elements above the main diagonal and those below the main diagonal retain the row corresponding to the row in the original matrix. Thus, the matrix A in (12–9) can be stored in the upper triangular form as shown in (12–10):

$$A = \begin{bmatrix} a_{11} & 0 & 0 & a_{14} & a_{15} & 0 \\ 0 & a_{22} & 0 & 0 & a_{25} & a_{26} \\ 0 & 0 & a_{33} & 0 & 0 & a_{36} \\ a_{14} & 0 & 0 & a_{44} & 0 & 0 \\ a_{15} & a_{25} & 0 & 0 & a_{55} & 0 \\ 0 & a_{26} & a_{36} & 0 & 0 & a_{66} \end{bmatrix} \tag{12–9}$$

$$AD = \begin{bmatrix} a_{11} & a_{14} & a_{15} \\ a_{22} & a_{25} & a_{26} \\ a_{33} & a_{36} & * \\ a_{44} & * & * \\ a_{55} & * & * \\ a_{66} & * & * \end{bmatrix} \tag{12–10}$$

$$\text{INDEX} = (0, 3, 4)$$

The first element of INDEX corresponds to the main diagonal of A, which is stored in AD(*,1). The second element of INDEX implies that the superdiagonal that is 3 away from the main diagonal is stored in AD(*,2) and so on. The positive elements in INDEX indicate that the upper triangular part is being stored.

The matrix A can also be stored in the lower triangular form as shown in (12–11):

$$AD = \begin{bmatrix} a_{11} & * & * \\ a_{22} & * & * \\ a_{33} & * & * \\ a_{44} & a_{14} & * \\ a_{55} & a_{25} & a_{15} \\ a_{66} & a_{36} & a_{26} \end{bmatrix} \tag{12–11}$$

$$\text{INDEX} = (0, -3, -4)$$

The asterisk (*) indicates that the element does not belong to the matrix A. These elements should be set to zero in the array AD. The negative elements in INDEX indicate that the lower triangular part is being stored.

The array AD is dimensioned as $ndim$ by nz, where nz is the number of diagonals stored and $ndim$ is the declared leading dimension of AD as given in the calling program. The INDEX array has dimension nz. Note that nz can be at most n for an n by n system.

The characteristics of this storage scheme are as follows:

- The diagonals can be stored in any order.
- Either the upper or the lower triangular part is stored. Thus the elements in INDEX after the first element, are all positive or all negative.
- Elements which are part of AD, but not part of A, should be set equal to zero. These are the elements denoted by the asterisk (*).

12.3.1.2 UDIA: Unsymmetric Diagonal Storage Scheme

This storage scheme follows the same principle as the symmetric diagonal storage scheme, but both the upper and the lower triangular halves of the matrix are stored. Thus, the matrix A in (12–12) is stored as shown in (12–13):

$$A = \begin{bmatrix} a_{11} & 0 & 0 & a_{14} & a_{15} & 0 \\ 0 & a_{22} & 0 & 0 & a_{25} & a_{26} \\ a_{31} & 0 & a_{33} & 0 & 0 & a_{36} \\ 0 & a_{42} & 0 & a_{44} & 0 & 0 \\ a_{51} & 0 & a_{53} & 0 & a_{55} & 0 \\ 0 & a_{62} & 0 & a_{64} & 0 & a_{66} \end{bmatrix} \tag{12–12}$$

$$AD = \begin{bmatrix} a_{11} & a_{14} & a_{15} & * & * \\ a_{22} & a_{25} & a_{26} & * & * \\ a_{33} & a_{36} & * & a_{31} & * \\ a_{44} & * & * & a_{42} & * \\ a_{55} & * & * & a_{53} & a_{51} \\ a_{66} & * & * & a_{64} & a_{62} \end{bmatrix} \tag{12–13}$$

$$INDEX = (0, 3, 4, -2, -4)$$

The array AD is dimensioned $ndim$ by nz, where nz is the number of diagonals stored and $ndim$ is the leading dimension of the matrix, as given in the calling program. The array INDEX has dimension nz. For an n by n system, nz can be at most $2n - 1$.

The characteristics of this storage scheme are as follows:

- The diagonals can be stored in any order.
- Elements which are part of AD, but not part of A, should be set equal to zero. These are the elements denoted by the asterisk (*).

12.3.1.3 GENR: General Storage Scheme by Rows

This storage scheme can be used for storing general unstructured matrices. Each nonzero element is stored along with its row and column indices. Thus there is a single array of matrix elements, AR, along with an array of row indices, IA, and an array of the corresponding column indices, JA.

As the matrix is stored by rows, the row index for all the nonzero elements in a row is stored only once. The i-th element of array IA points to the start of the i-th row in arrays JA and AR. For example if IA(3) = 6, then the third row is stored starting from the 6-th element in AR and JA. As IA(4) points to the start of the

fourth row, the number of elements in the third row is given by IA(4)-IA(3). Thus the matrix A in (12–14) is stored using three vectors as shown in (12–15):

$$A = \begin{bmatrix} a_{11} & 0 & 0 & a_{14} & a_{15} & 0 \\ 0 & a_{22} & a_{23} & 0 & 0 & a_{26} \\ a_{31} & 0 & a_{33} & 0 & a_{35} & a_{36} \\ 0 & a_{42} & 0 & a_{44} & 0 & 0 \\ a_{51} & 0 & 0 & 0 & a_{55} & 0 \\ 0 & 0 & a_{63} & a_{64} & 0 & a_{66} \end{bmatrix} \tag{12–14}$$

$$\begin{aligned} \mathrm{AR} &= (a_{11}, a_{14}, a_{15}, a_{22}, a_{23}, a_{26}, a_{31}, a_{33}, a_{35}, a_{36}, a_{42}, a_{44}, a_{51}, a_{55}, a_{63}, a_{64}, a_{66}) \\ \mathrm{JA} &= (1, 4, 5, 2, 3, 6, 1, 3, 5, 6, 2, 4, 1, 5, 3, 4, 6) \\ \mathrm{IA} &= (1, 4, 7, 11, 13, 15, 18) \end{aligned} \tag{12–15}$$

The dimension of AR and JA is at least nz, where nz is the number of nonzero elements in the matrix. IA is of length $n+1$ (for an n by n matrix), and the last element is $nz+1$. This helps in determining the end of the last row of the matrix and the number of nonzero elements in the last row. To store all the indices in one array, both IA and JA are stored in an array INDEX, with the first $n+1$ location used for IA and the rest of the vector used for JA. The length of INDEX is at least $nz+n+1$. Thus, for the previous example, the array INDEX is as follows:

$$\mathrm{INDEX} = (1, 4, 7, 11, 13, 15, 18, 1, 4, 5, 2, 3, 6, 1, 3, 5, 6, 2, 4, 1, 5, 3, 4, 6)$$

where the first 7 elements form IA and the rest form JA.

12.3.2 Types of Preconditioners

For each of the three storage schemes for sparse matrices, DXML provides the routines to create and apply the following three preconditioners:

- Diagonal preconditioner
- Neumann polynomial preconditioner
- Incomplete LU preconditioner

12.3.2.1 DIAG: Diagonal Preconditioner

This is the simplest of the three preconditioners, with the preconditioning matrix Q chosen as the diagonal of the coefficient matrix A. There is no explicit split of Q into Q_L and Q_R. As the diagonal preconditioner approximates the matrix A by just its diagonal, it is usually not a good preconditioner for a general system (12–1).

12.3.2.2 POLY: Polynomial Preconditioner

The polynomial preconditioner is derived from the matrix A by first splitting it into its diagonal and off-diagonal parts and then considering the inverse of A as a truncated version of a polynomial series expansion. Let the coefficient matrix A be written as:

$$\begin{aligned} A &= D - C \\ &= (I - C * D^{-1}) * D \\ &= (I - B) * D \end{aligned}$$

where D is the diagonal of A, $-C$ is the matrix of off-diagonal elements, and $B = C * D^{-1}$. By a polynomial series expansion, the inverse of A can be written as:

$$A^{-1} = D^{-1} * (I - B)^{-1} = D^{-1} * (I + B + B^2 + B^3 + ...).$$

A polynomial preconditioner of degree m essentially considers Q^{-1} to be a truncated version of the series expansion, that is:

$$Q^{-1} = D^{-1} * (I + B + B^2 + B^3 + ... + B^m).$$

This polynomial expansion of the inverse of A is called the Neumann polynomial. The preconditioner is obtained in the form Q^{-1}, not Q, and there is no explicit split into the matrices Q_L and Q_R.

The effectiveness of polynomial preconditioners depends on how closely Q^{-1} approximates A^{-1}. This is determined by the matrix A itself as well as the degree of the polynomial. While a higher degree polynomial usually indicates a better approximation, it also involves extra computation per iteration. For preconditioning with a higher degree polynomial to be effective, the reduction in iterations must be sufficient to offset the extra computation per iteration.

12.3.2.3 ILU: Incomplete LU Preconditioner

The incomplete LU preconditioner obtains a factorization of A into lower and upper triangular factors, the matrices L and U, respectively, such that the following conditions are satisfied:

- Matrices L and U have the same nonzero structure as the matrix A.
- Nonzero elements of matrix A are equal to the corresponding element of the product $L * U$.

Thus $A \approx L * U = Q$ and $Q^{-1} = U^{-1} * L^{-1}$.

The factorization is referred to as 'incomplete' as the product $L*U$ has nonzeros in locations where the matrix A has zeros, that is, the product $L * U$ is not identical to the matrix A as in the case of a 'complete' LU factorization.

If A is a symmetric matrix, as in the SDIA storage scheme, an incomplete Cholesky decomposition is computed with:

$$U = L^T$$

Incomplete factorizations usually form good preconditioners especially if the extra nonzero elements in the product $L * U$ are relatively small.

12.4 Iterative Solvers

DXML provides six iterative solvers based on the conjugate-gradient and conjugate-gradient-like techniques:

- DITSOL_PCG: Conjugate gradient method
- DITSOL_PLSCG: Least squares conjugate gradient method
- DITSOL_PBCG: Biconjugate gradient method
- DITSOL_PCGS: Conjugate gradient squared method
- DITSOL_PGMRES: Generalized minimum residual method
- DITSOL_PTFQMR: Transpose-free quasiminimal residual method

Each solver is applicable to a class of problems determined by the properties of the coefficient matrix A in (12–1) or the preconditioned matrix A' in (12–2) if preconditioning is used. The reference descriptions of the iterative solver subprograms at the end of this chapter outline the conditions under which each method can be applied.

DXML provides you with the option of using each iterative method without preconditioning as well as with right, left and split preconditioning. Table 12–8 indicates which forms of preconditioning are available for each method.

Table 12–8 Preconditioners for the Iterative Methods

Method	None	Left	Right	Split	SPD Split
DITSOL_PCG	X				X
DITSOL_PLSCG	X	X	X	X	
DITSOL_PBCG	X	X	X	X	
DITSOL_PCGS	X	X	X	X	
DITSOL_PGMRES	X	X	X	X	
DITSOL_PTFQMR	X	X	X	X	

12.4.1 Driver Routine

DXML includes a driver routine, DITSOL_DRIVER, that incorporates the calls to the MATVEC, PCONDL, and PCONDR routines so that you do not have to write these routines. The parameter list of the driver routine is identical to the parameter list of SOLVER, with the exception of the external routines MATVEC, PCONDL and PCONDR, which now refer to routines provided by DXML.

By setting values of appropriate parameters in the array IPARAM, you can choose a solver, a storage scheme and a preconditioner. The preconditioner must be created prior to the call to the driver routine. The driver routine does not allow the use of the matrix-free formulation of the iterative solver. The reference description of the driver routine at the end of this chapter provides further details.

12.5 Naming Conventions

The DXML routines can be broadly classified into two: routines related to the iterative solver and independent of the matrix, and routines that perform the matrix operations and are thus identified with a storage scheme.

Each routine name starts with the character D to indicate double-precision real routines. The next character group determines the operation being performed, namely, iterative solver, matrix-vector product, creation of the preconditioner and application of the preconditioner. Depending on the operation, the third, fourth and fifth character groups are chosen to reflect the iterative method, the storage scheme or the preconditioner.

Table 12–9 shows the naming conventions used for the iterative solver subroutines. Each routine name is obtained by concatenating the appropriate options from a character group, each character group separated by an underscore character.

Table 12–9 Naming Conventions: Iterative Solver Routines

Character Group	Mnemonic	Meaning
first	D	double precision
second	ITSOL MATVEC CREATE APPLY	function of the routine - iterative solver, matrix vector product, creation of preconditioner, or application of preconditioner.
third	DEFAULTS DRIVER PCG PLSCG PBCG PCGS PGMRES PTFQMR	options for the iterative solver
	SDIA UDIA GENR	storage scheme options for matrix vector product
	DIAG POLY ILU	preconditioner options for creation and application of preconditioners
fourth	ALL[1] SDIA UDIA GENR	storage scheme options for creation and application of preconditioners
fifth	L U	application of ILU preconditioner for UDIA and GENR storage schemes

[1]This option is for the application of the diagonal preconditioner only.

For example, DMATVEC_UDIA is the routine that obtains the matrix-vector product for the matrix stored using the unsymmetric diagonal (UDIA) storage scheme. Similarly, DAPPLY_ILU_GENR_L applies the incomplete LU (ILU) preconditioner for a matrix stored using the general storage by rows (GENR) scheme. The L indicates that the lower triangular part of the LU preconditioner is being considered.

12.6 Summary of Iterative Solver Subroutines

Tables 12–10, 12–11, 12–12, and 12–13 summarize the subroutines for the iterative solvers and the matrix operations.

Table 12–10 Summary of Iterative Solver Routines

Routine	Operation
DITSOL_DEFAULTS	Set the default values in the arrays IPARAM and RPARAM
DITSOL_DRIVER	Driver routine for the iterative solvers
DITSOL_PCG	Apply the preconditioned conjugate gradient method
DITSOL_PLSCG	Apply the preconditioned least squares conjugate gradient method
DITSOL_PBCG	Apply the preconditioned biconjugate gradient method
DITSOL_PCGS	Apply the preconditioned conjugate gradient squared method
DITSOL_PGMRES	Apply the preconditioned generalized minimum residual method
DITSOL_PTFQMR	Apply the preconditioned transpose-free quasiminimal residual method

Table 12–11 Summary of Matrix-Vector Product Routines

Routine	Operation
DMATVEC_SDIA	Matrix vector product for the symmetric diagonal storage scheme
DMATVEC_UDIA	Matrix vector product for the unsymmetric diagonal storage scheme
DMATVEC_GENR	Matrix vector product for the general storage by rows scheme

Table 12–12 Summary of Preconditioner Creation Routines

Routine	Operation
DCREATE_DIAG_SDIA	Create the diagonal preconditioner for the symmetric diagonal storage scheme
DCREATE_DIAG_UDIA	Create the diagonal preconditioner for the unsymmetric diagonal storage scheme
DCREATE_DIAG_GENR	Create the diagonal preconditioner for the general storage by rows scheme
DCREATE_POLY_SDIA	Create the polynomial preconditioner for the symmetric diagonal storage scheme
DCREATE_POLY_UDIA	Create the polynomial preconditioner for the unsymmetric diagonal storage scheme
DCREATE_POLY_GENR	Create the polynomial preconditioner for the general storage by rows scheme
DCREATE_ILU_SDIA	Create the incomplete LU preconditioner for the symmetric diagonal storage scheme

(continued on next page)

Table 12–12 (Cont.) Summary of Preconditioner Creation Routines

Routine	Operation
DCREATE_ILU_UDIA	Create the incomplete LU preconditioner for the unsymmetric diagonal storage scheme
DCREATE_ILU_GENR	Create the incomplete LU preconditioner for the general storage by rows scheme

Table 12–13 Summary of Preconditioner Application Routines

Routine	Operation
DAPPLY_DIAG_ALL	Apply the diagonal preconditioner for all storage schemes
DAPPLY_POLY_SDIA	Apply the polynomial preconditioner for the symmetric diagonal storage scheme
DAPPLY_POLY_UDIA	Apply the polynomial preconditioner for the unsymmetric diagonal storage scheme
DAPPLY_POLY_GENR	Apply the polynomial preconditioner for the general storage by rows scheme
DAPPLY_ILU_SDIA	Apply the incomplete LU preconditioner for the symmetric diagonal storage scheme
DAPPLY_ILU_UDIA_L	Apply the incomplete LU preconditioner for the unsymmetric diagonal storage scheme (operates on the L part)
DAPPLY_ILU_UDIA_U	Apply the incomplete LU preconditioner for the unsymmetric diagonal storage scheme (operates on the U part)
DAPPLY_ILU_GENR_L	Apply the incomplete LU preconditioner for the general storage by rows scheme (operates on the L part)
DAPPLY_ILU_GENR_U	Apply the incomplete LU preconditioner for the general storage by rows scheme (operates on the U part)

12.7 Error Handling

The six iterative solver subroutines include an error flag in the argument list. This is not an optional argument. On exit from the iterative solver routine, check its value to ensure that the solver converged normally.

The error flag can have various values. A return value of 0 indicates a normal return from the solver, with the iterations converging under the specified conditions. A negative return value implies a fatal error such as incorrect input, insufficient workspace, or use of a method inapplicable to the problem being solved. In such cases, a fatal error message is printed out describing the problem and control is returned to the calling routine. A positive return value of the error flag indicates a warning, such as a method terminating after reaching the maximum number of iterations. This is a correctable error if the maximum number of iterations is set to a low value. However, it could also signal a more fatal error such as the stagnation of the quantity being measured for convergence.

Fatal error messages are always printed out. The only exception to this is when the value of the unit number for output (iounit) is negative. In such cases, no output is produced by the solver and therefore it is important to check the error flag on exit.

In addition to the solvers, DXML provides routines for the creation and application of various preconditioners for matrices stored using various storage schemes. The preconditioner is created by a call to the appropriate routine prior to calling the iterative solver. You must ensure that the preconditioner used does indeed exist. For example, in the case of diagonal preconditioning, the elements along the diagonal of the matrix must be nonzero. The routines that generate the preconditioners do not explicitly check for the existence of the preconditioner. Therefore, you could get an internal exception error, such as division by zero or square-root of a negative number, that terminates the execution of the program. In contrast, an error in the iterative solver does not terminate the execution. Instead, the error flag is set and control returned to the calling program.

Table 12–14 shows the values of the error flags, an explanation of each value and the action needed to recover from each error. The values –2100 through –2104 indicate a breakdown in the iterative process caused by either an inappropriate input parameter or the use of an inappropriate method for the problem. The remaining values, all negative, indicate that input data to the solver is invalid.

Table 12–14 Error Flags for Sparse Iterative Solver Subprograms

Error Flag	Description	User Action
0	Normal exit	No action required.
2001	Method did not converge	Increase value of **itmax**; check that **rparam(2)** is not stagnating.
–2001	Invalid **ipcond**	Refer to subprogram description for valid values and rerun with acceptable value.
–2002	Invalid **iolevel**	"
–2003	Invalid **nipar**	"
–2004	Invalid **nrwrk**	"
–2005	Invalid **istop**	"
–2006	Invalid **errtol**	"
–2007	Invalid **n**	"
–2008	Invalid **nrpar**	"
–2009	Zero denominator in the stopping criterion	Rerun with a different stopping criterion.
–2010	Invalid **isolve**	Refer to Table 12–4 for valid values and rerun with acceptable value.
–2011	Invalid **iprec**	Refer to Table 12–4 for valid values and rerun with acceptable value.
–2012	Invalid **istore**	Refer to Table 12–4 for valid values and rerun with acceptable value.

(continued on next page)

Table 12–14 (Cont.) Error Flags for Sparse Iterative Solver Subprograms

Error Flag	Description	User Action
–2100	Preconditioner is not positive-definite	Rerun with a different preconditioner or solver.
–2101	Matrix is not positive-definite	Rerun with a different solver.
–2102	Breakdown in generation of direction vector	Rerun with a different solver, preconditioner, or starting guess.
–2103	Breakdown in update to solution vector	Rerun with a different solver, preconditioner, or starting guess.
–2104	Breakdown in *gmres* iteration	Rerun with a different solver, preconditioner, starting guess, or **kprev**.
–2200	memory allocation routine in the parallel version failed	Increase allocated values of pagefile quota and virtual memory, or reduce the number of processors, or use serial version of the solver.

12.8 Hints on the Use of the Iterative Solver

The iterative solvers included in DXML provide you with a wide choice of iterative methods and preconditioners. Additional flexibility is provided via the adoption of a matrix-free formulation of the iterative method. This allows you to use any storage scheme for the matrix, but implies that you have to write the routines for the matrix operations. You also have the option of using routines included in DXML for the matrix operations, but this restricts you to the storage schemes and preconditioners provided by DXML. It is also possible to mix the two approaches and use a storage scheme included in DXML, but provide your own routines to create and apply the preconditioner of your choice.

DXML provides further flexibility by allowing you to set various parameters such as the form of preconditioner, the stopping criterion, the maximum number of iterations allowed, the degree of the polynomial for polynomial preconditioning and so on. These enable you to control the iterative procedure and fine tune it to suit the needs of your application.

The steps in using the iterative solver can be summarized as follows:

- Choose the storage scheme for the coefficient matrix A.

  You can either choose one of the schemes provided by DXML (SDIA, UDIA, or GENR) or choose your own storage scheme.

- Select an iterative method (DITSOL_PCG, DITSOL_PLSCG, DITSOL_PBCG, DITSOL_PCGS, DITSOL_PGMRES, or DITSOL_PTFQMR), a form of preconditioner (none, left, right, split, or SPD split), and if necessary, a preconditioner (DIAG, POLY, or ILU).

  The preconditioner and the form of preconditioning should match. For example, DIAG and POLY preconditioners cannot be used in the split form as they do not explicitly generate the matrices Q_L and Q_R. However, they can be used in SPD split preconditioning as it requires only the matrix Q. The form of preconditioning is chosen via the parameter ipcond (IPARAM(7)). You also have the option of providing your own preconditioner.

- Write the routines MATVEC (required) and, if necessary, the routines PCONDL and PCONDR.

  The reference descriptions for each iterative method at the end of this chapter describe the functionality that must be provided by these routines. If you have chosen your own storage scheme for the coefficient matrix or chosen your own preconditioner, then you must provide the functionality required by the routines MATVEC, PCONDL, and PCONDR. If, however, you have chosen one of the preconditioners and storage schemes included in DXML, the functionality required by these routines is provided via a call to the appropriate DXML routines. In this case, you should be consistent in the use of storage schemes, that is, the same storage schemes should be used in all operations.

  Whether you use the routines for the matrix operations provided by DXML or not, it is your responsibility to provide the routine MATVEC and, if applicable, the routines PCONDL and PCONDR, with the standard interface.

  You also have the option of using the driver routine, DITSOL_DRIVER, in which case, you do not have to provide the routines MATVEC, PCONDL and PCONDR; instead you use the versions of these routines provided by DXML.

- Assign values to the variables in array IPARAM and RPARAM.

  This could be done via a call to the routine DITSOL_DEFAULTS. The routine DITSOL_DEFAULTS does not set the values of all parameters in the arrays IPARAM and RPARAM and it is your responsibility to ensure that all appropriate variables have been assigned valid values before a call to the iterative solver. Information such as the size of the work arrays is provided in the routine descriptions at the end of this chapter. In addition to the assignment of values to the variables, any associated setup should also be done at this time such as the opening of files for I/O. If you choose to implement your own stopping criterion, you must provide the routine MSTOP, with the standard interface given in Table 12–3.

- Generate the preconditioner.

  This is done either by a call to one of DXML's routines, if you are using a storage scheme and preconditioner provided by DXML, or by writing your own routine. Unlike the routines for the application of the preconditioner, there is no standard interface for the routine to generate the preconditioner as it is formed before the call to the iterative solver. However, the preconditioner must be generated in a manner that would be consistent with the use of the standard interface for the routines PCONDL and PCONDR.

  If you use a DXML routine to generate the preconditioner, it is your responsibility to ensure that the preconditioner does exist. For example, in the case of diagonal preconditioning, the diagonal elements of the coefficient matrix A must be nonzero. If a DXML routine is called to create a preconditioner and it does not exist, the routine terminates with an appropriate system message such as a division by zero error or a square-root of a negative number error.

  If you are using the driver routine, DITSOL_DRIVER, you must create the appropriate preconditioner using the appropriate storage scheme, prior to the call to the driver routine.

- Call the appropriate iterative solver routine with the standard interface given in Table 12–7.

On exit from the solver, the error flag, ierror, should be checked to ensure that the iterations converged normally. A return value of 0 indicates a normal exit from the routine SOLVER, with the iterations converging under the specified conditions. A negative return value implies a fatal error such as incorrect input or insufficient workspace, while a positive return value indicates a warning such as the solver terminating after the maximum number of iterations.

The best solver and preconditioner for a given problem is very dependent on the problem itself. Often, a choice is made based on prior experience. In the absence of this, the following factors may influence the selection of a particular solver or preconditioner:

- Applicability of a solver

  Each solver is applicable under certain conditions. Some solvers can be applied to symmetric matrices only; others require the evaluation of both $A * x$ and $A^T * x$. The reference description for each solver includes the conditions under which it is applicable. These must be considered in the choice of a solver for a problem.

- Effectiveness of a preconditioner

  Based on the properties of the problem, some preconditioners may be more effective than others in improving the convergence of a solver. However, the increase in time per iteration due to the use of the preconditioner must also be taken into account. For example, increasing the degree of the polynomial in polynomial preconditioning will increase the time per iteration, but may not always reduce the number of iterations sufficiently to lead to an overall reduction in the execution time.

- Amount of memory available

  Different solvers and preconditioners require different amounts of memory. For example, the generalized minimum residual method (DITSOL_PGMRES) allows you to use a variable number of previous residual vectors. As this number increases, the work done per iteration, and the memory required, also increase. Often, but not always, there is a corresponding increase in the convergence rate. A limited amount of memory may preclude the choice of a sufficiently large number of previous residual vectors to ensure convergence.

In addition to choosing a solver and a preconditioner, you also have the choice of a stopping criterion. It is important to select carefully the condition that determines when the iterative process will be terminated. DXML includes four stopping criteria. It also allows you the option of writing your own version. In some cases, a stopping criterion may be obtained at a low cost from the iterative process. But, this may not be the most effective criterion for judging the convergence of your particular problem. In the initial stages of experimenting with different stopping criteria, it may help to calculate the residual of the system explicitly at the end of the iterative process in order to determine if the choice of stopping criteria was an appropriate one.

The preceding guidelines should be taken into consideration in your choice of a solver, preconditioner and stopping criterion. As this choice is very dependent on the problem, some experimentation may be necessary to determine the most efficient method. DXML provides you a flexible interface that allows different solvers, preconditioners and stopping criterion to be tested easily. In addition, the input parameters allow you to generate extensive information on the solution process. These can be used to gain a better insight into the behavior of various

solvers and preconditioners for your problem, helping you to choose a combination that will provide a solution efficiently.

12.9 A Look at Some Iterative Solvers

To illustrate the use of the sparse iterative solvers and the concepts of matrix-free formulation and preconditioning, consider the linear system of equations derived from the discretization of the Laplace's equation on the unit square with Dirichlet boundary conditions, using the standard five point central differencing scheme:

$$-u_{xx} - u_{yy} = 0$$

$$0 \leq x \leq 1$$

$$0 \leq y \leq 1$$

Assuming nx grid points in x direction and ny grid points in the y direction, the coefficient matrix A has the following form for $nx = ny = 4$:

```
 4 -1       | -1          |             |
-1  4 -1    |    -1       |             |
   -1  4 -1 |       -1    |             |
      -1  4 |          -1 |             |
 -----------|-------------|-------------|-----------
-1          |  4 -1       |-1           |
   -1       | -1  4 -1    |   -1        |
      -1    |    -1  4 -1 |      -1     |
         -1 |       -1  4 |         -1  |
 -----------|-------------|-------------|-----------
            | -1          | 4 -1        |-1
            |    -1       |-1  4 -1     |   -1
            |       -1    |   -1  4 -1  |      -1
            |          -1 |      -1  4  |         -1
 -----------|-------------|-------------|-----------
            |             |-1           | 4 -1
            |             |   -1        |-1  4 -1
            |             |      -1     |   -1  4 -1
            |             |         -1  |      -1  4
```

The elements not defined in the matrix are zero. The exact solution is assumed to be all 1.0 and the starting guess is the zero vector.

Examples 12–1, 12–2, 12–3, and 12–4 illustrate the use of the iterative solvers, the various storage schemes and the preconditioners. Each example is self-contained, with comments indicating the operations being performed. The output files created by the example programs illustrate the information that can be obtained from the iterative solvers by the setting of appropriate parameters.

These examples illustrate the use of the DXML iterative solver routines from Fortran, C, and C++ codes. Additional examples are included on-line in the `/usr/examples/dxml` directory.

Example 12–1 Iterative Solver with User-Defined Routines (Fortran Code)

```
      PROGRAM EXAMPLE_ITSOL
C
C ***** THIS PROGRAM ILLUSTRATES THE FOLLOWING:
C
C       (1) USE OF THE SOLVER TO SOLVE THE TEST PROBLEM VIA
C           PRECONDITIONED CONJUGATE GRADIENT METHOD, USING A
C           USER-DEFINED STORAGE SCHEME.
C       (2) USE OF A USER-DEFINED ROUTINE MATVEC
C       (3) USE OF A USER DEFINED ROUTINE MSTOP
C       (4) USE OF USER DEFINED ROUTINE PCONDL
C       (5) USE OF THE ROUTINE DITSOL_DEFAULTS
C       (6) INFORMATION PRINTED OUT FOR IOLEVEL = 3
C
      IMPLICIT REAL*8 (A-H, O-Z)
C
      PARAMETER (NMAX = 100)
C
      REAL*8 X(NMAX), XO(NMAX), RHS(NMAX), QL(NMAX)
      REAL*8  RPARAM(50), RWORK(4*NMAX), DUM, TEMP
C
      INTEGER IPARAM(50), IA(2), IDUM
      INTEGER I,NX, NY, NXNY
C
      COMMON /MATRIX/ A1(NMAX), A2(NMAX), A3(NMAX), A4(NMAX), A5(NMAX)
C
      EXTERNAL MATVEC, PCONDL, USER_MSTOP
C
C ***** SET UP PROBLEM SIZE
C
      NX = 10
      NY = 10
      NXNY = NX*NY
C
C ***** SET THE PARAMETERS (INTEGER AND REAL)
C
      CALL DITSOL_DEFAULTS (IPARAM, RPARAM)
C
C ***** CHANGE ANY VALUES THAT ARE DIFFERENT FROM THE DEFAULT
C       ASSIGN VALUES TO PARAMETERS NOT SET BY ROUTINE DITSOL_DEFAULTS
C           CHANGE IOUNIT TO 7
C           CHANGE IPCOND TO 4 (FOR SPD SPLIT PRECONDITIONING)
C           CHANGE IOLEVEL TO 3
C           CHANGE ISTOP TO 0 (FOR USER DEFINED MSTOP)
C       A NONZERO VALUE OF ISTOP WILL SELECT ONE OF THE STANDARD
C       STOPPING CRITERIA
C
      IPARAM(3) = 0
      IPARAM(4) = 4*NXNY
      IPARAM(5) = 7
      IPARAM(6) = 3
      IPARAM(7) = 4
      IPARAM(8) = 0
```

(continued on next page)

Example 12–1 (Cont.) Iterative Solver with User-Defined Routines (Fortran Code)

```
C
C ***** SETUP OUTPUT FILE
C
      IOUNIT = IPARAM(5)
      OPEN (UNIT=IOUNIT,FILE='OUTPUT.DATA',STATUS='UNKNOWN')
      REWIND IOUNIT
C
      WRITE (IOUNIT,101)
  101 FORMAT (/,2X,'SOLVING EXAMPLE PROBLEM WITH SPD SPLIT
     $ PRECONDITIONED CG',/,2X,'DIAGONAL PRECONDITIONING USED ',/)
C
C ***** GENERATE THE MATRIX (USE IA TO PASS NX AND NY TO SOLVER)
C
      CALL GENMAT(NX, NY, NXNY)
      IA(1) = NX
      IA(2) = NY
C
C ***** GENERATE XO, THE TRUE SOLUTION
C
      DO I = 1, NXNY
          XO(I) = 1.0D0
      END DO
C
C ***** OBTAIN THE RIGHT HAND SIDE
C
      CALL MATVEC (0, IPARAM, RPARAM, A, IA, DUM, XO, RHS, NXNY)
C
C ***** OBTAIN INITIAL GUESS (ALL ZEROS)
C
      DO I = 1, NXNY
         X(I) = 0.0D0
      END DO
C
C ***** GENERATE THE DIAGONAL PRECONDITIONER
C
      CALL GEN_PCOND (NXNY, QL)
C
C ***** CALL THE SOLVER
C
       CALL DITSOL_PCG ( MATVEC, PCONDL, DUM, USER_MSTOP, A, IA,
     $            X, RHS, NXNY, QL, IDUM, DUM, IDUM,
     $            IPARAM, RPARAM, IDUM, RWORK, IERROR)
C
C ***** PRINT OUT THE SOLUTION
C
      WRITE (IOUNIT,102)
102   FORMAT (/,5X,'TRUE SOLUTION',5X,'SOLUTION FROM SOLVER',
     $5X,'ABS. DIFFERENCE'/)
      WRITE (IOUNIT,103) (XO(I),X(I),ABS(XO(I)-X(I)),I=1,NXNY)
103   FORMAT (/,3(5X,E15.8))
C
C ***** FIND MAX ERROR IN SOLUTION
C
      TEMP = ABS(XO(1) - X(1))
```

(continued on next page)

Example 12–1 (Cont.) Iterative Solver with User-Defined Routines (Fortran Code)

```
      DO I = 2, NXNY
         TEMP = MAX( TEMP, ABS(XO(I) - X(I)) )
      END DO
      WRITE (IOUNIT,104) TEMP
104   FORMAT (/,2X,'MAX ERROR IN SOLUTION= ', E15.8,/)
C
      STOP
      END
C
C
C
      SUBROUTINE MATVEC(JOB, IPARAM, RPARAM, A, IA, W, X, Y, N)
C
C ***** MULTIPLY THE VECTOR X BY THE MATRIX TO OBTAIN VECTOR Y
C       ONLY JOB = 0 NEEDED FOR THIS EXAMPLE
C
      IMPLICIT REAL*8 (A-H,O-Z)
C
      REAL*8 X(*), Y(*), A(*)
      INTEGER IA(*)
C
      CALL MULA (IA(1), IA(2), N, X, Y)
C
      RETURN
      END
C
C
C
      SUBROUTINE PCONDL(JOB, IPARAM, RPARAM, QL, IQL, A, IA,
     $                  W, X, Y, N)
C
C ***** CALL THE LEFT PRECONDITIONER
C       ONLY JOB = 0 NEEDED FOR THIS EXAMPLE
C
      IMPLICIT REAL*8 (A-H,O-Z)
      REAL*8 X(*), Y(*),  QL(*)
      INTEGER N
C
C ***** DIAGONAL PRECONDITIONING
C
      CALL APPLY_PCOND_DIA (N, X, Y, QL)
C
      RETURN
      END
C
C
C
      SUBROUTINE GEN_PCOND (N, QL)
C
C ***** GENERATE THE LEFT PRECONDITIONER
C
      IMPLICIT REAL*8 (A-H,O-Z)
```

(continued on next page)

Example 12–1 (Cont.) Iterative Solver with User-Defined Routines (Fortran Code)

```
      PARAMETER (NMAX = 100)
      REAL*8 QL(*)
      COMMON /MATRIX/ A1(NMAX), A2(NMAX), A3(NMAX), A4(NMAX), A5(NMAX)
      INTEGER N
C
      DO I = 1, N
          QL(I) = 1.0D0 / A3(I)
      END DO
C
      RETURN
      END
C
C
C
      SUBROUTINE APPLY_PCOND_DIA (N, X, Y, QL)
C
C ***** APPLY THE DIAGONAL  PRECONDITIONER
C
      IMPLICIT REAL*8 (A-H,O-Z)
      REAL*8 X(*), Y(*), QL(*)
C
      DO I = 1, N
          Y(I) = X(I) * QL(I)
      END DO
C
      RETURN
      END
C
C
C
      SUBROUTINE GENMAT (NX, NY, NXNY)
C
C ***** GENERATE THE MATRIX FOR THE EXAMPLE
C
      IMPLICIT REAL*8 (A-H,O-Z)
      PARAMETER (NMAX = 100)
      COMMON /MATRIX/ A1(NMAX), A2(NMAX), A3(NMAX), A4(NMAX), A5(NMAX)
C
      DO I = 1, NXNY
          A3(I) = 4.0D0
          A1(I) = 0.0D0
          A2(I) = 0.0D0
          A4(I) = 0.0D0
          A5(I) = 0.0D0
      END DO
C
      DO J = 1, NY
          DO I = 2, NX
              K = (J-1)*NX+I
              A2(K) = -1.0D0
          END DO
          DO I = 1, NX-1
              K = (J-1)*NX+I
              A4(K) = -1.0D0
          END DO
      END DO
C
      DO J = 2, NY
          DO I = 1, NX
              K = (J-1)*NX+I
```

(continued on next page)

Example 12–1 (Cont.) Iterative Solver with User-Defined Routines (Fortran Code)

```
                A1(K) = -1.0D0
            END DO
        END DO
C
        DO J = 1, NY-1
            DO I = 1, NX
                K = (J-1)*NX+I
                A5(K) = -1.0D0
            END DO
        END DO
C
        RETURN
        END
C
C
C
        SUBROUTINE MULA (NX, NY, NXNY, TMP1, TMP2)
C
C ***** TO OBTAIN THE MATRIX VECTOR MULTIPLY
C
        IMPLICIT REAL*8 (A-H,O-Z)
        PARAMETER (NMAX = 100)
        REAL*8 TMP1(*), TMP2(*)
        INTEGER NX,NY,NXNY
        COMMON /MATRIX/ A1(NMAX), A2(NMAX), A3(NMAX), A4(NMAX), A5(NMAX)
C
        DO I = 1, NXNY
            TMP2(I) = A3(I)*TMP1(I)
        END DO
C
        DO I = 1, NXNY-1
            TMP2(I) = TMP2(I) +
       $              A4(I)*TMP1(I+1)
        END DO
C
        DO I = 2, NXNY
            TMP2(I) = TMP2(I) +
       $              A2(I)*TMP1(I-1)
        END DO
C
        DO I = 1, NXNY-NX
            TMP2(I) = TMP2(I) +
       $              A5(I)*TMP1(I+NX)
        END DO
C
        DO I = NX+1, NXNY
            TMP2(I) = TMP2(I) +
       $              A1(I)*TMP1(I-NX)
        END DO
C
        RETURN
        END
C
C
C
```

(continued on next page)

Example 12–1 (Cont.) Iterative Solver with User-Defined Routines (Fortran Code)

```
      SUBROUTINE USER_MSTOP (IPARAM, RPARAM, X, R, Z, B, N)
C
C ***** ROUTINE FOR THE USER PROVIDED STOPPING CRITERION
C
      IMPLICIT REAL*8 (A-H,O-Z)
      REAL*8 X(*), B(*), R(*), Z(*), RPARAM(*), STPTST
      INTEGER ITERS, IPARAM(*)
C
      IOUNIT = IPARAM(5)
      ITERS = IPARAM(10)
      IF (ITERS.EQ.0) THEN
          WRITE(IOUNIT, 100)
  100     FORMAT(/,2X,'USING USER DEFINED STOPPING CRITERION',/)
      END IF
C
C ***** USER DEFINED STOPPING CRITERION USES MAX NORM OF RESIDUAL FOR
C       STPTST
C
      STPTST = ABS (R(1))
      DO I = 2, N
         STPTST = MAX ( STPTST,ABS(R(I)) )
      END DO
C
      RPARAM(2) = STPTST
C
      RETURN
      END
```

Output from Example 1

```
SOLVING EXAMPLE PROBLEM WITH SPD SPLIT PRECONDITIONED CG
DIAGONAL PRECONDITIONING USED
 METHOD USED : CG WITH SPD SPLIT PRECONDITIONING
 ORDER OF SYSTEM =    100
 STOPPING CRITERION USED :     0
 MAXIMUM ITERATIONS ALLOWED:   100
 TOLERANCE FOR CONVERGENCE :   0.10000000e-05
USING USER DEFINED STOPPING CRITERION
    ITERATION =          0    STOPPING TEST = 0.20000000e+01
    ITERATION =          1    STOPPING TEST = 0.92307692e+00
    ITERATION =          2    STOPPING TEST = 0.58793970e+00
    ITERATION =          3    STOPPING TEST = 0.63509006e+00
    ITERATION =          4    STOPPING TEST = 0.42973063e+00
    ITERATION =          5    STOPPING TEST = 0.48724587e+00
    ITERATION =          6    STOPPING TEST = 0.41781094e+00
    ITERATION =          7    STOPPING TEST = 0.50288290e+00
    ITERATION =          8    STOPPING TEST = 0.17070529e+00
    ITERATION =          9    STOPPING TEST = 0.38502953e-01
    ITERATION =         10    STOPPING TEST = 0.16076790e-01
    ITERATION =         11    STOPPING TEST = 0.75548469e-02
    ITERATION =         12    STOPPING TEST = 0.15431168e-02
    ITERATION =         13    STOPPING TEST = 0.14025362e-03
    ITERATION =         14    STOPPING TEST = 0.42745516e-05
    ITERATION =         15    STOPPING TEST = 0.63333204e-15
 SOLUTION OBTAINED AFTER          15  ITERATIONS
 NORMAL EXIT FROM SOLVER
 FINAL VALUE OF STOPPING TEST = 0.63333204e-15
   TRUE SOLUTION     SOLUTION FROM SOLVER     ABS. DIFFERENCE
    0.10000000e+01      0.10000000e+01      0.33306691e-15
    0.10000000e+01      0.10000000e+01      0.44408921e-15
                               :
                               : (EDITED FOR BREVITY)
    0.10000000e+01      0.10000000e+01      0.33306691e-15
    0.10000000e+01      0.10000000e+01      0.33306691e-15
MAX ERROR IN SOLUTION=  0.23314684e-14
```

Example 12–2 Iterative Solver with DXML Routines (Fortran Code)

```
      PROGRAM EXAMPLE_ITSOL

C
C ***** THIS PROGRAM ILLUSTRATES THE FOLLOWING:
C
C       (1) USE OF THE SOLVER TO SOLVE THE TEST PROBLEM VIA
C           PRECONDITIONED GMRES METHOD METHOD, WITH SPLIT INCOMPLETE
C           CHOLESKY PRECONDITIONING. THE MATRIX IS STORED USING
C           THE UNSYMMETRIC DIAGONAL FORMAT
C       (2) USE OF ROUTINE MATVEC (DMATVEC_UDIA)
C       (3) USE OF ROUTINES PCONDL AND PCONDR TO CALL ROUTINES FOR
C           APPLYING THE PRECONDITIONER (DAPPLY_ILU_UDIA_L AND
C           DAPPLY_ILU_UDIA_U)
C       (5) USE OF THE ROUTINE DITSOL_DEFAULTS
C
      IMPLICIT REAL*8 (A-H, O-Z)
C
      PARAMETER (NMAX = 100)
      PARAMETER (NDIM = 100)
      PARAMETER (KPREV_MAX = 5)
      PARAMETER (NWK = NMAX*(KPREV_MAX+2) +
     $                 KPREV_MAX*(KPREV_MAX+5)+1 )
C
      REAL*8 X(NMAX), XO(NMAX), RHS(NMAX)
      REAL*8 RPARAM(50), RWORK(NWK), DUM
      REAL*8 A_UDIA(NDIM,5), P_ILU(NDIM,5), TEMP
      INTEGER IP_ILU(5), INDEX_UDIA(5)
C
      INTEGER IPARAM(50), IDUM
      INTEGER I, NX, NY, NXNY
C
      EXTERNAL MATVEC, PCONDL, PCONDR
C
C ***** SET UP PROBLEM SIZE
C
      NX = 10
      NY = 10
      NXNY = NX*NY
      NZEROS = 5
C
C ***** SET THE PARAMETERS (INTEGER AND REAL)
C
      CALL DITSOL_DEFAULTS (IPARAM, RPARAM)
C
C ***** CHANGE ANY VALUES THAT ARE DIFFERENT FROM THE DEFAULT
C       ASSIGN VALUES TO PARAMETERS NOT SET BY DITSOL_DEFAULTS
C              CHANGE IOUNIT TO 7
C              CHANGE IPCOND TO 3 (FOR SPLIT PRECONDITIONING)
C              CHANGE IOLEVEL TO 3
C              CHANGE ISTOP TO 3 (ONLY ISTOP=3 AND 4 ALLOWED)
C
      IPARAM(3) = 0
      IPARAM(4) = NWK
C
      IPARAM(5) = 7
      IPARAM(6) = 3
      IPARAM(7) = 3
      IPARAM(8) = 3
C
C ***** ASSIGN VALUE TO KPREV (NUMBER OF PREVIOUS RESIDUALS STORED)
```

(continued on next page)

Example 12–2 (Cont.) Iterative Solver with DXML Routines (Fortran Code)

```
C       NOTE THAT THE SIZE OF RWORK ALLOWS A MAXIMUM VALUE OF
C       KPREV = 5
C
      IPARAM(34) = 3
C
C ***** SETUP OUTPUT FILE
C
      IOUNIT = IPARAM(5)
      OPEN (UNIT=IOUNIT,FILE='OUTPUT.DATA',STATUS='UNKNOWN')
      REWIND IOUNIT
C
      WRITE (IOUNIT,101)
101   FORMAT (/,2X,'SOLVING EXAMPLE PROBLEM WITH SPLIT
     $ PRECONDITIONED GMRES',/,2X,'ILU PRECONDITIONING USED ',/
     $ 2X,'MATRIX STORED IN UNSYMMETRIC DIAGONAL FORMAT',/)
C
C ***** GENERATE THE MATRIX
C
      CALL GENMAT(NX, NY, NXNY, A_UDIA, INDEX_UDIA, NDIM, NZEROS)
      IPARAM(31) = NZEROS
      IPARAM(32) = NDIM
C
C ***** GENERATE XO, THE TRUE SOLUTION
C
      DO I = 1, NXNY
          XO(I) = 1.0D0
      END DO
C
C ***** OBTAIN THE RIGHT HAND SIDE
C
      CALL MATVEC (0, IPARAM, RPARAM, A_UDIA, INDEX_UDIA,
     $             DUM, XO, RHS, NXNY)
C
C ***** OBTAIN INITIAL GUESS (ALL ZEROS)
C
      DO I = 1, NXNY
         X(I) = 0.0D0
      END DO
C
C ***** GENERATE THE ILU PRECONDITIONER
C
      CALL DCREATE_ILU_UDIA (A_UDIA, INDEX_UDIA, NDIM, NZEROS,
     $                       P_ILU, IP_ILU, NXNY)
C
C ***** CALL THE SOLVER
C
       CALL DITSOL_PGMRES ( MATVEC, PCONDL, PCONDR, DUM,
     $               A_UDIA, INDEX_UDIA,
     $               X, RHS, NXNY,
     $               P_ILU, IP_ILU, P_ILU, IP_ILU,
     $               IPARAM, RPARAM, IDUM, RWORK, IERROR)
C
C ***** PRINT OUT THE SOLUTION
C
```

(continued on next page)

Example 12–2 (Cont.) Iterative Solver with DXML Routines (Fortran Code)

```
      WRITE (IOUNIT,102)
102   FORMAT (/,5X,'TRUE SOLUTION',5X,'SOLUTION FROM SOLVER',
     $5X,'ABS. DIFFERENCE'/)
      WRITE (IOUNIT,103) (XO(I),X(I),ABS(XO(I)-X(I)),I=1,NXNY)
103   FORMAT (/,3(5X,E15.8))
C
C ***** FIND MAX ERROR IN SOLUTION
C
      TEMP = ABS(XO(1) - X(1))
      DO I =2,NXNY
         TEMP = MAX( TEMP, ABS(XO(I) - X(I)) )
      END DO
      WRITE(IOUNIT,104) TEMP
104   FORMAT(/,2X,'MAX ERROR IN SOLUTION = ', E15.8,/)
C
      STOP
      END
C
C
C
      SUBROUTINE MATVEC(JOB, IPARAM, RPARAM, A, IA, DUM, X, Y, N)
C
C  MULTIPLY N VECTOR X BY MATRIX TO OBTAIN Y
C
      IMPLICIT REAL*8 (A-H,O-Z)
C
      REAL*8 X(*), Y(*), A(*), RPARAM(*)
      INTEGER IA(*), JOB, IPARAM(*)
C
      NZEROS = IPARAM(31)
      NDIM = IPARAM(32)
C
      CALL DMATVEC_UDIA (JOB, A, IA, NDIM, NZEROS, DUM,
     $                   X, Y, N)
C
      RETURN
      END
C
C
C
      SUBROUTINE PCONDL(JOB, IPARAM, RPARAM, QL, IQL, A, IA,
     $                  W, X, Y, N)
C
C ***** CALL THE LEFT PRECONDITIONER
C
      IMPLICIT REAL*8 (A-H,O-Z)
      PARAMETER (NMAX = 100)
      REAL*8 X(*), Y(*), RPARAM(*), QL(*), A(*), TMP(NMAX)
      INTEGER JOB, IPARAM(*), IQL(*), IA(*)
C
C ***** ILU PRECONDITIONING
C
      IPCOND = IPARAM(7)
      NZEROS = IPARAM(31)
      NDIM = IPARAM(32)
```

(continued on next page)

Example 12–2 (Cont.) Iterative Solver with DXML Routines (Fortran Code)

```
C
      CALL DAPPLY_ILU_UDIA_L (JOB, QL, IQL, NDIM, NZEROS,
     $                        X, Y, N)
C
      RETURN
      END
C
C
C
      SUBROUTINE PCONDR(JOB, IPARAM, RPARAM, QR, IQR, A, IA,
     $                  W, X, Y, N)
C
C ***** CALL THE RIGHT PRECONDITIONER
C
      IMPLICIT REAL*8 (A-H,O-Z)
      REAL*8 X(*), Y(*), RPARAM(*), QR(*), A(*)
      INTEGER JOB, IPARAM(*), IQR(*), IA(*)
C
C ***** ILU PRECONDITIONING
C
      IPCOND = IPARAM(7)
      NZEROS = IPARAM(31)
      NDIM = IPARAM(32)
C
      CALL DAPPLY_ILU_UDIA_U (JOB, QR, IQR, NDIM, NZEROS,
     $                        X, Y, N)
C
      RETURN
      END
C
C
C
      SUBROUTINE GENMAT (NX, NY, NXNY, A, IA, NDIM, NZEROS)
C
C ***** GENERATE THE MATRIX FOR THE EXAMPLE IN THE UNSYMMETRIC
C       DIAGONAL FORM
C
      IMPLICIT REAL*8 (A-H,O-Z)
      REAL*8 A(NDIM,*)
      INTEGER IA(*)
C
      DO J = 2, NZEROS
          DO I = 1, NXNY
              A(I,J) = 0.0D0
          END DO
      END DO
C
      DO I = 1, NXNY
          A(I,1) = 4.0D0
      END DO
C
      DO J = 1, NY-1
          DO I = 1, NX
              K = (J-1)*NX+I
              A(K,2) = -1.0D0
          END DO
      END DO
```

(continued on next page)

Example 12–2 (Cont.) Iterative Solver with DXML Routines (Fortran Code)

```
C
      DO J = 1, NY
          DO I = 2, NX
              K = (J-1)*NX+I
              A(K,3) = -1.0D0
          END DO
      END DO
C
      DO J = 1, NY
          DO I = 1, NX-1
              K = (J-1)*NX+I
              A(K,4) = -1.0D0
          END DO
      END DO
C
      DO J = 2, NY
          DO I = 1, NX
              K = (J-1)*NX+I
              A(K,5) = -1.0D0
          END DO
      END DO
C
      IA(1) = 0
      IA(2) = NX
      IA(3) = -1
      IA(4) = 1
      IA(5) = -NX
C
      RETURN
      END
```

Output from Example 2

```
SOLVING EXAMPLE PROBLEM WITH SPLIT PRECONDITIONED GMRES
ILU PRECONDITIONING USED
MATRIX STORED IN UNSYMMETRIC DIAGONAL FORMAT
 METHOD USED : GMRES WITH SPLIT PRECONDITIONING
          3  PREVIOUS RESIDUAL VECTORS ARE STORED
 ORDER OF SYSTEM =    100
 STOPPING CRITERION USED :     3
 MAXIMUM ITERATIONS ALLOWED:   100
 TOLERANCE FOR CONVERGENCE :   0.10000000e-05
    ITERATION =          0    STOPPING TEST = 0.27403910e+01
    ITERATION =          1    STOPPING TEST = 0.83454209e+00
    ITERATION =          2    STOPPING TEST = 0.44870733e+00
    ITERATION =          3    STOPPING TEST = 0.17433646e+00
    ITERATION =          4    STOPPING TEST = 0.72856695e-01
    ITERATION =          5    STOPPING TEST = 0.38408003e-01
    ITERATION =          6    STOPPING TEST = 0.14164437e-01
    ITERATION =          7    STOPPING TEST = 0.56991076e-02
    ITERATION =          8    STOPPING TEST = 0.28790502e-02
    ITERATION =          9    STOPPING TEST = 0.13411353e-02
    ITERATION =         10    STOPPING TEST = 0.63725219e-03
    ITERATION =         11    STOPPING TEST = 0.32213502e-03
    ITERATION =         12    STOPPING TEST = 0.13084728e-03
    ITERATION =         13    STOPPING TEST = 0.54223092e-04
    ITERATION =         14    STOPPING TEST = 0.27502204e-04
    ITERATION =         15    STOPPING TEST = 0.13133042e-04
    ITERATION =         16    STOPPING TEST = 0.63404914e-05
    ITERATION =         17    STOPPING TEST = 0.32280646e-05
    ITERATION =         18    STOPPING TEST = 0.13311315e-05
    ITERATION =         19    STOPPING TEST = 0.55695871e-06
 SOLUTION OBTAINED AFTER          19  ITERATIONS
 NORMAL EXIT FROM SOLVER
 FINAL VALUE OF STOPPING TEST = 0.55695871e-06
   TRUE SOLUTION     SOLUTION FROM SOLVER     ABS. DIFFERENCE
    0.10000000e+01      0.99999997e+00      0.32325871e-07
    0.10000000e+01      0.99999987e+00      0.13396025e-06
                               :
                               : (edited for brevity)
                               :
    0.10000000e+01      0.99999983e+00      0.16690401e-06
    0.10000000e+01      0.99999995e+00      0.48797753e-07
MAX ERROR IN SOLUTION =  0.78315035e-06
```

Example 12–3 Iterative Solver with DXML Routines (C Code)

```
/*
 ****************************************************************************
 *                                                                          *
 *  Copyright Digital Equipment Corporation 1993 - 1995. All rights reserved.*
 *                                                                          *
 *  Restricted Rights: Use, duplication, or disclosure by the U.S.          *
 *  Government is subject to restrictions as set forth in subparagraph      *
 *  (c) (1) (ii) of DFARS 252.227-7013, or in FAR 52.227-19, or in FAR      *
 *  52.227-14 Alt. III, as applicable.                                      *
 *                                                                          *
 *  This software is proprietary to and embodies the confidential           *
 *  technology of Digital Equipment Corporation. Possession, use, or        *
 *  copying of this software and media is authorized only pursuant to a     *
 *  valid written license from Digital or an authorized sublicensor.        *
 *                                                                          *
 ****************************************************************************
*/
/*
    This is an example program to illustrate the use of the iterative
    solver ditsol_pcg from a C application program. The program generates
    the matrix and the preconditioner, calls the solver and prints the
    maximum error in the solution. The right hand side of the problem is
    generated assuming a known solution. The problem used is identical to
    the one in the example section of the chapter on iterative solvers
    in the DXML Reference Guide.

    This program illustrates the following:
        - routine naming convention for Digital Unix and VMS
        - differences in array limits between Fortran and C:
              C default: x[n] -> 0 to (n-1)
              Fortran default:   x(n) -> 1 to n
        - how to store two dimensional arrays in C for use in a
          Fortran library routine
        - how to use the matrix-free formulation from a C program

    For more detailed explanation of the routines used, please
    check the Reference Manual or manpage or the Fortran example
    programs in this directory.

    Note: the code used in this example works on both Digital Unix and
    VMS. Conditional compilation is used to select the statements appropriate
    to each operating system.

    All output from this program is sent to the screen.

*/


#include <stdio.h>
#include <stdlib.h>

#define ABS(x) (((x) < 0) ? -(x) : (x))
#define MAX(x,y) (((x) < (y)) ? (y) : (x))

/*
    Add trailing underscores to Fortran routines on Digital Unix.
*/
```

(continued on next page)

Example 12–3 (Cont.) Iterative Solver with DXML Routines (C Code)

```
#if !defined(vms) && !defined(__vms)
#define ditsol_defaults ditsol_defaults_
#define dcreate_ilu_sdia dcreate_ilu_sdia_
#define ditsol_pcg ditsol_pcg_
#define dmatvec_sdia dmatvec_sdia_
#define dapply_ilu_sdia dapply_ilu_sdia_
#endif

extern void  pcondl1();
extern void  matvec1();
extern void  genmat1();
extern void ditsol_defaults();
extern void dcreate_ilu_sdia();
extern void ditsol_pcg();
extern void dmatvec_sdia();
extern void dapply_ilu_sdia();

/*
    illustrating the use of the iterative solver:
    preconditioned conjugate gradient method, with incomplete
    cholesky preconditioning. the matrix is stored using the
    symmetric diagonal format
*/

int main()
{
    double *a_sdia;
    double *a_ilu;
    double *rwork1;
    double *rhs;
    double *x;
    double *xo;

    double rparam[50];

    double dum, max1, tmp1;

    int *index_sdia;
    int *index_ilu;

    int iparam[50];

    int  nx, ny, nxny, length, ndim, nzeros;
    int  i, j, idum, ierror;
    int  job;

/*
    define the size of the problem
*/

    nx = 10;
    ny = 10;
    nzeros = 3;
    nxny = nx * ny;
    ndim = nxny;

/*
   get the memory for the 2-dimensional arrays a_sdia and a_ilu
*/

    a_sdia = (double *)malloc (nzeros*ndim*sizeof(double));
    if (a_sdia == 0) perror("malloc");

    a_ilu = (double *)malloc (nzeros*ndim*sizeof(double));
    if (a_ilu == 0) perror("malloc");
```

(continued on next page)

Example 12–3 (Cont.) Iterative Solver with DXML Routines (C Code)

```
/*
    get the memory for the 1-dimensional arrays
*/

    rwork1 = (double *)malloc(4*nxny*sizeof(double));
    if (rwork1 == 0) perror("malloc");

    rhs = (double *)malloc(nxny*sizeof(double));
    if (rhs == 0) perror("malloc");

    x = (double *)malloc(nxny*sizeof(double));
    if (x == 0) perror("malloc");

    xo = (double *)malloc(nxny*sizeof(double));
    if (xo == 0) perror("malloc");

    index_sdia = (int *)malloc(nzeros*sizeof(int));
    if (index_sdia == 0) perror("malloc");

    index_ilu = (int *)malloc(nzeros*sizeof(int));
    if (index_ilu == 0) perror("malloc");

/*
    set the parameters (integer and real)
*/

    ditsol_defaults(iparam, rparam);

    iparam[2] = 0;
    iparam[3] = 4 * nxny;

/*
    direct all output to the screen
*/

    iparam[4] = 6;
    iparam[5] = 3;
    iparam[6] = 4;

/*
    generate the matrix
*/

    genmat1(nx, ny, nxny, a_sdia, index_sdia, ndim, nzeros);

    iparam[30] = nzeros;
    iparam[31] = ndim;

/*
    generate xo, the true solution
*/

    for (i=0; i<nxny; i++)
        xo[i] = 1.0;

/*
    obtain the right hand side
*/

    job = 0;
    matvec1(&job, iparam, rparam, a_sdia, index_sdia,
            &dum, xo, rhs, &nxny);

/*
    obtain initial guess (all zeros)
*/
```

(continued on next page)

Example 12–3 (Cont.) Iterative Solver with DXML Routines (C Code)

```
    for (i=0; i<nxny; i++)
        x[i] = 0.0;

/*
    generate the preconditioner
*/

    dcreate_ilu_sdia(a_sdia, index_sdia, &ndim, &nzeros,
                     a_ilu, index_ilu, &nxny);

/*
    call the solver
*/

    ditsol_pcg(matvec1, pcondl1, &dum, &dum,
               a_sdia, index_sdia,
               x, rhs, &nxny,
               a_ilu, index_ilu, &dum, &idum,
               iparam, rparam, &idum, rwork1, &ierror);

    if (ierror != 0)
        printf("ditsol_pcg returned with error flag: %d\n",ierror);

/*
    find the maximum absolute error in the solution
*/

    max1 = ABS((x[0]-xo[0]));

    for (i=1; i<nxny; i++)
        {
            tmp1 = ABS((x[i]-xo[i]));
            max1 = MAX((max1),(tmp1));
        }

/*
    print the maximum absolute error
*/

    printf("maximum error in the solution: %.10e\n",max1);

/*
    release the memory
*/

    free(a_sdia);
    free(a_ilu);
    free(rwork1);
    free(rhs);
    free(x);
    free(xo);
    free(index_sdia);
    free(index_ilu);

}  /* end of main() */


/*
    generate the matrix for the problem described in the chapter on
    iterative solvers in the DXML Reference Guide
*/

void genmat1(int nx, int ny, int nxny, double a[],
             int index[], int ndim, int nzeros)
{
    int  i, j, k;
```

(continued on next page)

Example 12–3 (Cont.) Iterative Solver with DXML Routines (C Code)

```
    for (j=0; j<nxny; j++)
        for (i=1; i<nzeros; i++)
            a[i*ndim+j] = 0.0;

    for (j=0; j<nxny; j++)
        a[0*ndim+j] = 4.0;

    for (j=0; j<ny; j++)
        for (i=1; i<nx; i++)
            {
                k = j * nx + i;
                a[2*ndim+k] = -1.0;
            }

    for (j=1; j<ny; j++)
        for (i=0; i<nx; i++)
            {
                k = j * nx + i;
                a[1*ndim+k] = -1.0;
            }
    index[0] = 0;
    index[2] = -1;
    index[1] = -nx;

}  /* end of genmat1() */


/*
    provide the matrix-vector routine using the standard parameter list
    as described in the DXML Refernce Guide
*/

void matvec1(int *job, int *iparam, double *rparam, double *a, int *ia,
             double *w, double *x, double *y, int *n)
{
    int  nzeros, ndim;
    double dum;

    nzeros = iparam[30];
    ndim = iparam[31];

    dmatvec_sdia(job, a, ia, &ndim, &nzeros, &dum, x, y, n);

}  /* end of matvec1() */


/*
    provide the left preconditioning routine using the standard parameter
    list as described in the DXML Refernce Guide
*/

void pcondl1(int *job, int *iparam, double *rparam, double *ql, int *iql,
             double *a, int *ia, double *w, double *x, double *y, int *n)
{
    int job1;
    int  nzeros, ndim;

    double *tmp;

/*
    ilu preconditioning
*/

    nzeros = iparam[30];
    ndim = iparam[31];
```

(continued on next page)

Example 12–3 (Cont.) Iterative Solver with DXML Routines (C Code)

```
/*
    get memory for temporary vector
*/

    tmp = (double *)malloc((*n)*sizeof(double));

    job1 = 0;
    dapply_ilu_sdia(&job1, ql, iql, &ndim, &nzeros, x, tmp, n);
    job1 = 1;
    dapply_ilu_sdia(&job1, ql, iql, &ndim, &nzeros, tmp, y, n);

/*
    release memory for temporary vector
*/

    free(tmp);

}  /* end of pcondl1() */
```

Output from Example 3

```
   method used : cg with spd split preconditioning
   order of system =       100
   stopping criterion used =     1
   maximum iterations allowed =         100
   tolerance for convergence =   0.10000000E-05
      iteration =          0    stopping test = 0.69282032E+01
      iteration =          1    stopping test = 0.19194193E+01
      iteration =          2    stopping test = 0.12100937E+01
      iteration =          3    stopping test = 0.52439623E+00
      iteration =          4    stopping test = 0.84029860E-01
      iteration =          5    stopping test = 0.20539881E-01
      iteration =          6    stopping test = 0.34309306E-02
      iteration =          7    stopping test = 0.47063334E-03
      iteration =          8    stopping test = 0.16605002E-03
      iteration =          9    stopping test = 0.45072557E-04
      iteration =         10    stopping test = 0.72087304E-05
      iteration =         11    stopping test = 0.88047573E-06
   solution obtained after          11  iterations
   normal exit from solver
   final value of stopping test = 0.88047573E-06

maximum error in the solution: 5.6260082260e-08
```

Example 12–4 Iterative Solver with DXML Routines (C++ Code)

```
//
// ************************************************************************
// *
// *  Copyright Digital Equipment Corporation 1993 - 1995. All rights reserved.
// *
// *  Restricted Rights: Use, duplication, or disclosure by the U.S.
// *  Government is subject to restrictions as set forth in subparagraph
// *  (c) (1) (ii) of DFARS 252.227-7013, or in FAR 52.227-19, or in FAR
// *  52.227-14 Alt. III, as applicable.
// *
// *  This software is proprietary to and embodies the confidential
// *  technology of Digital Equipment Corporation. Possession, use, or
// *  copying of this software and media is authorized only pursuant to a
// *  valid written license from Digital or an authorized sublicensor.
// *
// ************************************************************************
//
//
//   This is an example program to illustrate the use of the iterative
//   solver ditsol_pcg from a C application program. The program generates
//   the matrix and the preconditioner, calls the solver and prints the
//   maximum error in the solution. The right hand side of the problem is
//   generated assuming a known solution. The problem used is identical to
//   the one in the example section of the chapter on iterative solvers
//   in the DXML Reference Guide.
//
//   This program illustrates the following:
//        - routine naming convention for Digital Unix and VMS
//        - Differences in endexing arrays:
//             C default: x[n] -> 0 to (n-1)
//             Fortran default: x(n) -> 1 to n
//        - how to use two dimensional arrays in C to interface with a
//          Fortran library routine
//        - how to use the matrix-free formulation from a C program
//
//    For more detailed explanation of the routines used, please
//    check the DXML Reference Manual.
//
//    Note: the code used in this example works on both Digital Unix and
//    VMS. Conditional compilation is used to select the statements
//    appropriate to each operating system.
//
//    All output from this program is sent to the screen.
//

#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <new.h>

//
//   Add trailing underscores to Fortran routines on Digital Unix.
//

#if !defined(vms) && !defined(__vms)
#define ditsol_defaults ditsol_defaults_
#define dcreate_ilu_sdia dcreate_ilu_sdia_
#define ditsol_pcg ditsol_pcg_
#define dmatvec_sdia dmatvec_sdia_
#define dapply_ilu_sdia dapply_ilu_sdia_
#endif
```

(continued on next page)

Example 12–4 (Cont.) Iterative Solver with DXML Routines (C++ Code)

```
inline double ABS(double x)
{
    return(((x) < 0) ? -(x) : (x));
}

inline double MAX(double x, double y)
{
    return(((x) < (y)) ? (y) : (x));
}

extern void (*set_new_handler(void (*memory_err)()))();
void memory_err()
{
    cout << "memory allocation error\n";
    exit(1); // quit program
}

extern void pcondl1(int &, int [], double [],
                    double [], int [],
                    double [], int [],
                    double [], double [], double [], int &);
extern void  matvec1(int &, int [], double [],
                     double [], int [],
                     double [], double [], double [], int &);
extern void  genmat1(int, int, int,
                     double [], int [],
                     int, int);

//
//   Declare the Fortran routines
//

extern "C"
{
void ditsol_defaults(int [], double []);
void dcreate_ilu_sdia(double [], int [],
                      int &, int &,
                      double [], int [],
                      int &);
void ditsol_pcg(void (*)(int &, int [], double [],
                         double [], int [],
                         double [], double [], double [],
                         int &),
                void (*)(int &, int [], double [],
                          double [], int [],
                          double [], int [],
                          double [], double [], double [], int &),
                double &, double &,
                double [], int [],
                double [], double [], int &,
                double [], int [],
                double &, int &,
                int [], double [],
                int &, double [],
                int &);
void dmatvec_sdia(int &, double [], int [],
                  int &, int &, double [], double [], double [],
                  int &);
void dapply_ilu_sdia(int &,
                     double [], int [],
                     int [], int [],
                     double [], double [], int &);
```

(continued on next page)

Example 12–4 (Cont.) Iterative Solver with DXML Routines (C++ Code)

```
}

//
//     illustrating the use of the iterative solver:
//     preconditioned conjugate gradient method, with incomplete
//     cholesky preconditioning. the matrix is stored using the
//     symmetric diagonal format
//

void main()
{
    double *a_sdia;
    double *a_ilu;
    double *rwork1;
    double *rhs;
    double *x;
    double *xo;

    double rparam[50];

    double dum, max1, tmp1;

    int *index_sdia;
    int *index_ilu;

    int iparam[50];

    int nxny, length, ndim, nzeros;
    int i, j, idum, ierror;
    int job;

// set up exception handler

    set_new_handler(memory_err);

// define the problem size

    const int nx = 10;
    const int ny = 10;

    nxny = nx * ny;
    ndim = nxny;
    nzeros = 3;

// allocate memory for the 2-dimensional arrays

    a_sdia = new double [nzeros*ndim];
    a_ilu = new double [nzeros*ndim];

// allocate memory for the 1-dimensional arrays

    rwork1 = new double [4*nxny];
    rhs = new double [nxny];
    x = new double [nxny];
    xo = new double [nxny];

    index_sdia = new int [nzeros];
    index_ilu = new int [nzeros];

// set the parameters (integer and real)

    ditsol_defaults(iparam, rparam);

    iparam[2] = 0;
    iparam[3] = 4 * nxny;

// direct all output to the screen
```

(continued on next page)

Example 12–4 (Cont.) Iterative Solver with DXML Routines (C++ Code)

```
    iparam[4] = 6;
    iparam[5] = 3;
    iparam[6] = 4;

// generate the matrix

    genmat1(nx, ny, nxny,
            a_sdia, index_sdia,
            ndim, nzeros);

    iparam[30] = nzeros;
    iparam[31] = ndim;

// generate xo, the true solution

    for (i=0; i<nxny; i++)
        xo[i] = 1.0;

// obtain the right hand side

    job = 0;
    matvec1(job, iparam, rparam,
            a_sdia, index_sdia,
            &dum, xo, rhs, nxny);

// obtain initial guess (all zeros)

    for (i=0; i<nxny; i++)
        x[i] = 0.0;

// generate the preconditioner

    dcreate_ilu_sdia(a_sdia, index_sdia,
                     ndim, nzeros,
                     a_ilu, index_ilu,
                     nxny);

// call the solver

    ditsol_pcg(matvec1, pcondl1, dum, dum,
               a_sdia, index_sdia,
               x, rhs, nxny,
               a_ilu, index_ilu,
               dum, idum,
               iparam, rparam,
               idum, rwork1,
               ierror);

    if (ierror != 0)
        cout << "ditsol_pcg returned with error flag: " << ierror
            << endl;

// find the maximum absolute error in the solution

    max1 = ABS((x[0]-xo[0]));

    for (i=1; i<nxny; i++)
        {
            tmp1 = ABS((x[i]-xo[i]));
            max1 = MAX((max1),(tmp1));
        }
// print the maximum absolute error

    cout << "maximum error in the solution: " << max1 << endl;

// deallocate the memory
```

(continued on next page)

Example 12–4 (Cont.) Iterative Solver with DXML Routines (C++ Code)

```
    delete a_sdia;
    delete a_ilu;
    delete rwork1;
    delete xo;
    delete x;
    delete rhs;

    delete index_sdia;
    delete index_ilu;

}  // end of main()


//
//    generate the matrix for the problem described in the chapter on
//    iterative solvers in the DXML Reference Guide
//

void genmat1(int nx, int ny, int nxny,
             double a[], int index[],
             int ndim, int nzeros)
{
    int  i, j, k;

    for (j=0; j<nxny; j++)
        for (i=1; i<nzeros; i++)
            a[i*ndim+j] = 0.0;

    for (j=0; j<nxny; j++)
        a[0*ndim+j] = 4.0;

    for (j=0; j<ny; j++)
        for (i=1; i<nx; i++)
            {
                k = j * nx + i;
                a[2*ndim+k] = -1.0;
            }

    for (j=1; j<ny; j++)
        for (i=0; i<nx; i++)
            {
                k = j * nx + i;
                a[1*ndim+k] = -1.0;
            }

    index[0] = 0;
    index[2] = -1;
    index[1] = -nx;

}  // end of genmat1()


//
//    provide the matrix-vector routine using the standard parameter list
//    as described in the DXML Reference Guide
//

void matvec1(int &job, int iparam[], double rparam[],
             double a[], int ia[],
             double w[], double x[], double y[], int &n)
{
    int  nzeros, ndim;
    double dum;
```

(continued on next page)

Example 12–4 (Cont.) Iterative Solver with DXML Routines (C++ Code)

```
    nzeros = iparam[30];
    ndim = iparam[31];

    dmatvec_sdia(job, a, ia, ndim, nzeros, &dum, x, y, n);

}  // end of matvec1()


//
//    provide the left preconditioning routine using the standard parameter
//    list as described in the DXML Refernce Guide
//

void pcondl1(int &job, int iparam[], double rparam[],
             double ql[], int iql[],
             double a[], int ia[],
             double w[], double x[], double y[], int &n)
{
    int  nzeros, ndim, job1;
    double *tmp;

// ilu preconditioning

    nzeros = iparam[30];
    ndim = iparam[31];

// allocate temporary storage

    tmp = new double [n];

    job1 = 0;
    dapply_ilu_sdia(job1, ql, iql, &ndim, &nzeros, x, tmp, n);
    job1 = 1;
    dapply_ilu_sdia(job1, ql, iql, &ndim, &nzeros, tmp, y, n);

// deallocate temporary storage

    delete tmp;

}  // end of pcondl1()
```

Output from Example 4

```
   method used : cg with spd split preconditioning
   order of system =       100
   stopping criterion used =     1
   maximum iterations allowed =         100
   tolerance for convergence =   0.10000000E-05
      iteration =          0    stopping test = 0.69282032E+01
      iteration =          1    stopping test = 0.19194193E+01
      iteration =          2    stopping test = 0.12100937E+01
      iteration =          3    stopping test = 0.52439623E+00
      iteration =          4    stopping test = 0.84029860E-01
      iteration =          5    stopping test = 0.20539881E-01
      iteration =          6    stopping test = 0.34309306E-02
      iteration =          7    stopping test = 0.47063334E-03
      iteration =          8    stopping test = 0.16605002E-03
      iteration =          9    stopping test = 0.45072557E-04
      iteration =         10    stopping test = 0.72087304E-05
      iteration =         11    stopping test = 0.88047573E-06

   solution obtained after          11  iterations
   normal exit from solver
   final value of stopping test = 0.88047573E-06

maximum error in the solution: 5.6260082260e-08
```

Sparse Iterative Solver Subprograms

This section provides descriptions of the iterative solver subprograms for real double-precision operations. The subprograms are grouped by functionality starting with the iterative solvers, followed by the routines for matrix-vector product, the routines for the creation of the preconditioners, and finally the routines for the application of the preconditioners.

DITSOL_DEFAULTS
Set Default Values

Format

DITSOL_DEFAULTS (iparam, rparam)

Arguments

iparam
integer*4
On entry, a one-dimensional array of length at least 50.
On exit, the variables in the IPARAM array are assigned the default values, listed in Table 12–4. Of the first 50 elements, the variables that are not assigned a default value, are set equal to zero.

rparam
real*8
On entry, a one-dimensional array of length at least 50.
On exit, the variables in the RPARAM array are assigned the default values, listed in Table 12–5. Of the first 50 elements, the variables that are not assigned a default value, are set equal to zero.

Description

DITSOL_DEFAULTS sets the default values for the variables in the arrays IPARAM and RPARAM. Of the first 50 elements, the variables that are not assigned a default value, are set equal to zero. It is your responsibility to ensure that any variables in IPARAM and RPARAM that are required by the iterative solver are set to an appropriate value before the call to the solver routine.

DITSOL_DRIVER
Driver for Sparse Iterative Solvers (Serial and Parallel Versions)

Format

DITSOL_DRIVER (dmatvec_driver, dpcondl_driver, dpcondr_driver, mstop, a, ia, x, b, n, ql, iql , qr, iqr, iparam, rparam, iwork, rwork, ierror)

Arguments

DITSOL_DRIVER has the standard parameter list for an iterative solver, with the exception of the first three arguments which must be DMATVEC_DRIVER, DPCONDL_DRIVER, and DPCONDR_DRIVER. These must be declared external in your calling (sub)program.

Description

DITSOL_DRIVER solves the system of linear equations:

$$A * x = b$$

using one of the five iterative methods provided in DXML. By a suitable choice of the variables $isolve$, $istore$ and $iprec$ in the array IPARAM, an appropriate solver, storage scheme and preconditioner are selected. The preconditioner must be created in the appropriate storage scheme, prior to the call to the driver routine.

The following table shows the preconditioning options and the preconditioners that are permitted:

Preconditioner	Left	Right	Split	SPD Split
Diagonal	X	X		X
Polynomial	X	X		X
ILU	X	X	X	X

The preconditioning options applicable to the various iterative solvers are summarized in Table 12–8.

The following table shows the real workspace requirements ($nrwk$) for each method and the corresponding preconditioning option:

Method	None	Left	Right	Split
DITSOL_PCG	$3n$			$4n$ (SPD split)
DITSOL_PLSCG	$4n$	$5n$	$5n$	$6n$
DITSOL_PBCG	$5n$	$7n$	$6n$	$7n$
DITSOL_PCGS	$6n$	$7n$	$6n$	$7n$
DITSOL_PGMRES	$nrwk1$	$nrwk1 + n$	$nrwk1 + n$	$nrwk1 + n$
DITSOL_PTFQMR	$7n$	$8n$	$8n$	$9n$

In DITSOL_PGMRES, $nrwk1 = n * (kprev + 1) + kprev * (kprev + 5) + 1$ where $kprev$ is the number of previous vectors stored. If ILU preconditioning is used, then an additional real workspace of length n is required.

If you use the option of defining your own MSTOP routine, see the reference description of each solver for the definition of the vector z. format.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DITSOL_PCG
Preconditioned Conjugate Gradient Method (Serial and Parallel Versions)

Format

DITSOL_PCG (matvec, pcondl, pcondr, mstop, a, ia, x, b, n, ql, iql , qr, iqr, iparam, rparam, iwork, rwork, ierror)

Arguments

DITSOL_PCG has the standard parameter list for an iterative solver.

Description

DITSOL_PCG implements the conjugate gradient method [Hestenes and Stiefel 1952, Reid 1971] for the solution of a linear system of equations where the coefficient matrix A is symmetric positive definite or mildly nonsymmetric. This method requires the routine MATVEC to provide operations for **job** = 0. The routines MATVEC, PCONDL (if used) and MSTOP (if used) should be declared external in your calling (sub)program. PCONDR is not used by DITSOL_PCG and is therefore a dummy input parameter.

DXML provides the following two forms of the method:

- Unpreconditioned conjugate gradient method:
  This is the conjugate gradient method applied to:

  $$A * x = b$$

  where A is a symmetric positive definite or a mildly nonsymmetric matrix. As no preconditioning is used, both PCONDL and PCONDR are dummy input parameters.
  For the unpreconditioned conjugate gradient method, the length of the real work space array, defined by the variable $nrwk$ (IPARAM(4)), should be at least $3 * n$, where n is the order of the matrix A.

  The vector z, passed as an input argument to the routine MSTOP, is not defined.

- Conjugate gradient method with symmetric positive definite split preconditioning:
  This is the conjugate gradient method applied to:

  $$(Q_L^{-1} * A * Q_L^{-T}) * (Q_L^T * x) = (Q_L^{-1} * b)$$

  where:

  $$Q = Q_L * Q_L^T$$

  is the symmetric positive definite preconditioning matrix. The routine PCONDL, with **job**= 0 should evaluate:

  $$v = Q^{-1} * u$$

  The routine PCONDR is not used and an explicit split of the preconditioner Q into Q_L and Q_R is not required.

For the conjugate gradient method, with symmetric positive definite split preconditioning, the length of the real work space array, defined by the variable $nrwk$ (IPARAM(4)), should be at least $4 * n$, where n is the order of the matrix A. This does not include the memory requirements of the preconditioner.

The vector z, passed as an input argument to the routine MSTOP, is defined as:

$$z = Q^{-1} * r$$

where r is the residual at the i-th iteration.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DITSOL_PLSCG
Preconditioned Least Square Conjugate Gradient Method (Serial and Parallel Versions)

Format

DITSOL_PLSCG (matvec, pcondl, pcondr, mstop, a, ia, x, b, n, ql, iql , qr, iqr, iparam, rparam, iwork, rwork, ierror)

Arguments

DITSOL_PLSCG has the standard parameter list for an iterative solver.

Description

The least squares conjugate gradient is a robust method for the solution of general linear systems. It is equivalent to applying the conjugate gradient method to the normal equations:

$$A^T * A * x = A^T * b$$

This method requires the evaluation of two matrix products, involving matrix A and A^T. It suffers from the drawback that the condition number of $A^T * A$ is the square of the condition number of A, and therefore the convergence of the method is slow. To alleviate the numerical instability resulting from a straightforward application of the conjugate gradient method to the normal equations, DXML adopts the implementation proposed in [Björck and Elfving 1979].

The implementation of the least squares conjugate gradient method requires the routine MATVEC to provide operations for both **job**= 0 and **job**= 1. The routines MATVEC, PCONDL (if used), PCONDR (if used) and MSTOP (if used) should be declared external in your calling (sub)program.

DXML provides the following four forms of the method:

- Unpreconditioned least squares conjugate gradient method:
  This is the conjugate gradient method applied to:

  $$A^T * A * x = A^T * b$$

  where A is a general matrix. As no preconditioning is used, both PCONDL and PCONDR are dummy input parameters.

  For the unpreconditioned least squares conjugate gradient method, the length of the real work space array, defined by the variable $nrwk$ (IPARAM(4)), should be at least $4 * n$, where n is the order of the matrix A.

  The vector z, passed as an input argument to the routine MSTOP, is not defined.

- Least squares conjugate gradient method with left preconditioning:
  This is the conjugate gradient method applied to:

  $$(A^T * Q_L^{-T} * Q_L^{-1} * A) * x = (A^T * Q_L^{-T} * Q_L^{-1} * b)$$

  The routine PCONDL, with **job**= 0 should evaluate:

  $$v = Q_L^{-1} * u$$

  and with **job**= 1 should evaluate:

  $$v = Q_L^{-T} * u$$

  The routine PCONDR is not used and is therefore a dummy input parameter.

For the least squares conjugate gradient method, with left preconditioning, the length of the real work space array, defined by the variable *nrwk* (IPARAM(4)), should be at least 5*n, where n is the order of the matrix *A*. This does not include the memory requirements of the preconditioner.

The vector *z*, passed as an input argument to the routine MSTOP, is defined as:

$$z = Q_L^{-1} * r$$

where *r* is the residual at the *i*-th iteration.

- Least squares conjugate gradient method with right preconditioning:
  This is the conjugate gradient method applied to:

$$(Q_R^{-T} * A^T * A * Q_R^{-1}) * y = (Q_R^{-T} * A^T * b)$$

where:

$$y = Q_R * x$$

The routine PCONDR, with **job**= 0 should evaluate:

$$v = Q_R^{-1} * u$$

and with **job**= 1 should evaluate:

$$v = Q_R^{-T} * u$$

The routine PCONDL is not used and is therefore a dummy input parameter.

For the least squares conjugate gradient method, with right preconditioning, the length of the real work space array, defined by the variable *nrwk* (IPARAM(4)), should be at least 5*n, where *n* is the order of the matrix *A*. This does not include the memory requirements of the preconditioner.

The vector *z*, passed as an input argument to the routine MSTOP, is not defined.

- Least squares conjugate gradient method with split preconditioning:
  This is the conjugate gradient method applied to:

$$(Q_R^{-T} * A^T * Q_L^{-T} * Q_L^{-1} * A * Q_R^{-1}) * y = (Q_R^{-T} * A^T * Q_L^{-T} * Q_L^{-1} * b)$$

where:

$$y = Q_R * x$$

The routine PCONDL, with **job**= 0 should evaluate:

$$v = Q_L^{-1} * u$$

and with **job**= 1 should evaluate:

$$v = Q_L^{-T} * u$$

The routine PCONDR, with **job**= 0 should evaluate:

$$v = Q_R^{-1} * u$$

and with **job**= 1 should evaluate:

$$v = Q_R^{-T} * u$$

For the least squares conjugate gradient method, with split preconditioning, the length of the real work space array, defined by the variable $nrwk$ (IPARAM(4)), should be at least $6 * n$, where n is the order of the matrix A. This does not include the memory requirements of the preconditioner.

The vector z, passed as an input argument to the routine MSTOP, is defined as:

$$z = Q_L^{-1} * r$$

where r is the residual at the i-th iteration.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DITSOL_PBCG
Preconditioned Biconjugate Gradient Method (Serial and Parallel Versions)

Format

DITSOL_PBCG (matvec, pcondl, pcondr, mstop, a, ia, x, b, n, ql, iql , qr, iqr, iparam, rparam, iwork, rwork, ierror)

Arguments

DITSOL_PBCG has the standard parameter list for an iterative solver.

Description

The biconjugate gradient method is a method for the solution of nonsymmetric linear systems of equations. It is similar to the conjugate gradient method but generates two sequences of mutually orthogonal residuals [Fletcher 1976]. While there is no solid theoretical basis for the convergence behavior of the biconjugate gradient method, it can be efficient for some classes of problems. This method requires two matrix products involving the matrix A and A^T, but there is no squaring of the condition number.

The implementation of the biconjugate gradient method requires the routine MATVEC to provide operations for both **job**= 0 and **job**= 1. The routines MATVEC, PCONDL (if used), PCONDR (if used), and MSTOP (if used) should be declared external in your calling (sub)program.

DXML provides the following four forms of the method:

- Unpreconditioned biconjugate gradient method:
  This is the biconjugate gradient method applied to:

  $$A * x = b$$

  where A is a general matrix. As no preconditioning is used, both PCONDL and PCONDR are dummy input parameters.

  For the unpreconditioned bi-conjugate gradient method, the length of the real work space array, defined by the variable $nrwk$ (IPARAM(4)), should be at least $5 * n$, where n is the order of the matrix A.

  The vector z, passed as an input argument to the routine MSTOP, is not defined.

- Bi-conjugate gradient method with left preconditioning:
  This is the bi-conjugate gradient method applied to:

  $$(Q_L^{-1} * A) * x = (Q_L^{-1} * b)$$

  The routine PCONDL, with **job**= 0 should evaluate:

  $$v = Q_L^{-1} * u$$

  and with **job**= 1 should evaluate:

  $$v = Q_L^{-T} * u$$

  The routine PCONDR is not used and is therefore a dummy input parameter.

For the biconjugate gradient method, with left preconditioning, the length of the real work space array, defined by the variable $nrwk$ (IPARAM(4)), should be at least $7 * n$, where n is the order of the matrix A. This does not include the memory requirements of the preconditioner.

The vector z, passed as an input argument to the routine MSTOP, is defined as:

$$z = Q_L^{-1} * r$$

where r is the residual at the i-th iteration.

- Biconjugate gradient method with right preconditioning:
  This is the bi-conjugate gradient method applied to:

$$\left(A * Q_R^{-1}\right) * y = b$$

where:

$$y = Q_R * x$$

The routine PCONDR, with **job**= 0 should evaluate:

$$v = Q_R^{-1} * u$$

and with **job**= 1 should evaluate:

$$v = Q_R^{-T} * u$$

The routine PCONDL is not used and is therefore a dummy input parameter.

For the biconjugate gradient method, with right preconditioning, the length of the real work space array, defined by the variable nrwk (IPARAM(4)), should be at least 6*n, where n is the order of the matrix A. This does not include the memory requirements of the preconditioner.

The vector z, passed as an input argument to the routine MSTOP, is not defined.

- Biconjugate gradient method with split preconditioning:
  This is the biconjugate gradient method applied to:

$$\left(Q_L^{-1} * A * Q_R^{-1}\right) * y = \left(Q_L^{-1} * b\right)$$

where:

$$y = Q_R * x$$

The routine PCONDL, with **job**= 0 should evaluate:

$$v = Q_L^{-1} * u$$

and with **job**= 1 should evaluate:

$$v = Q_L^{-T} * u$$

The routine PCONDR, with **job**= 0 should evaluate:

$$v = Q_R^{-1} * u$$

and with **job**= 1 should evaluate:

$$v = Q_R^{-T} * u$$

For the biconjugate gradient method, with split preconditioning, the length of the real work space array, defined by the variable $nrwk$ (IPARAM(4)), should be at least $7 * n$, where n is the order of the matrix A. This does not include the memory requirements of the preconditioner.

The vector z, passed as an input argument to the routine MSTOP, is defined as:

$$z = Q_L^{-1} * r$$

where r is the residual at the i-th iteration.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DITSOL_PCGS
Preconditioned Conjugate Gradient Squared Method (Serial and Parallel Versions)

Format

DITSOL_PCGS (matvec, pcondl, pcondr, mstop, a, ia, x, b, n, ql, iql , qr, iqr, iparam, rparam, iwork, rwork, ierror)

Arguments

DITSOL_PCGS has the standard parameter list for an iterative solver.

Description

The conjugate gradient squared method [Sonneveld 1989] accelerates the convergence of the biconjugate gradient method by generating residuals which are related to the original residual by the square of a polynomial in A, instead of a polynomial in A, as in the case of the conjugate gradient and the bi-conjugate gradient methods. In practice, this results in the conjugate gradient squared method converging roughly twice as fast as the biconjugate gradient method. The additional advantage is that only the matrix A is involved and not A^T. The computational cost for both the biconjugate gradient method and the conjugate gradient squared method are about the same per iteration.

The implementation of the conjugate gradient squared method requires the routine MATVEC to provide operations for **job**= 0. The routines MATVEC, PCONDL (if used), PCONDR (if used), and MSTOP (if used) should be declared external in your calling (sub)program.

DXML provides the following four forms of the method:

- Unpreconditioned conjugate gradient squared method:
  This is the conjugate gradient squared method applied to:

  $$A * x = b$$

  where A is a general matrix. As no preconditioning is used, both PCONDL and PCONDR are dummy input parameters.

  For the unpreconditioned conjugate gradient squared method, the length of the real work space array, defined by the variable $nrwk$ (IPARAM(4)), should be at least $6 * n$, where n is the order of the matrix A.

  The vector z, passed as an input argument to the routine MSTOP, is not defined.

- Conjugate gradient squared method with left preconditioning:
  This is the conjugate gradient squared method applied to:

  $$(Q_L^{-1} * A) * x = (Q_L^{-1} * b)$$

  The routine PCONDL, with **job**= 0 should evaluate:

  $$v = Q_L^{-1} * u$$

  The routine PCONDR is not used and is therefore a dummy input parameter.

For the conjugate gradient squared method, with left preconditioning, the length of the real work space array, defined by the variable *nrwk* (IPARAM(4)), should be at least $7 * n$, where n is the order of the matrix A. This does not include the memory requirements of the preconditioner.

The vector z, passed as an input argument to the routine MSTOP, is defined as:

$$z = Q_L^{-1} * r$$

where r is the residual at the i-th iteration.

- Conjugate gradient squared method with right preconditioning:
  This is the conjugate gradient squared method applied to:

  $$(A * Q_R^{-1}) * y = b$$

  where:

  $$y = Q_R * x$$

  The routine PCONDR, with **job**= 0 should evaluate:

  $$v = Q_R^{-1} * u$$

  The routine PCONDL is not used and is therefore a dummy input parameter.

  For the conjugate gradient squared method, with right preconditioning, the length of the real work space array, defined by the variable *nrwk* (IPARAM(4)), should be at least $6 * n$, where n is the order of the matrix A. This does not include the memory requirements of the preconditioner.

  The vector z, passed as an input argument to the routine MSTOP, is not defined.

- Conjugate gradient squared method with split preconditioning:
  This is the conjugate gradient squared method applied to:

  $$(Q_L^{-1} * A * Q_R^{-1}) * y = (Q_L^{-1} * b)$$

  where:

  $$y = Q_R * x$$

  The routine PCONDL, with **job**= 0 should evaluate:

  $$v = Q_L^{-1} * u$$

  and the routine PCONDR, with **job**= 0 should evaluate:

  $$v = Q_R^{-1} * u$$

  For the conjugate gradient squared method, with split preconditioning, the length of the real work space array, defined by the variable *nrwk* (IPARAM(4)), should be at least $7 * n$, where n is the order of the matrix A. This does not include the memory requirements of the preconditioner.

  For split preconditioning, the vector z, passed as an input argument to the routine MSTOP, is defined as:

  $$z = Q_L^{-1} * r$$

  where r is the residual at the i-th iteration.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DITSOL_PGMRES
Preconditioned Generalized Minimum Residual Method (Serial and Parallel Versions)

Format

DITSOL_PGMRES (matvec, pcondl, pcondr, mstop, a, ia, x, b, n, ql, iql , qr, iqr, iparam, rparam, iwork, rwork, ierror)

Arguments

DITSOL_PGMRES has the standard parameter list for an iterative solver.

Description

The generalized minimum residual method [Saad and Schultz 1986] obtains a solution x of the form:

$$x = x_0 + z$$

where x_0 is the initial guess and z is a vector that minimizes the two norm of the residual:

$$r = b - A * (x_0 + z)$$

over the Krylov space:

$$K = \text{span}\left\{r_0, Ar_0, A^2 r_0, ..., A^{k-1} r_0\right\}$$

of dimension k, with the initial residual r_0 defined as:

$$r_0 = b - A * x_0$$

DXML implements the restarted generalized minimum residual method, where the method is restarted every *kprev* steps. This implies that only the *kprev* residuals need to be stored, instead of all the previous residuals as in the generalized minimum residual method, resulting in a substantial savings in memory.

The choice of *kprev* is crucial and requires some skill and experience — too small a value could result in poor convergence or no convergence at all, while too large a value could result in excessive memory requirements. *kprev* should be assigned a value prior to a call to DITSOL_PGMRES with the parameter IPARAM(34) in the array IPARAM. A suggested starting value for *kprev* is in the range of 3 to 6. If convergence is not obtained, the value should be increased.

While the generalized minimum residual method is applicable to a general problem and the residuals guaranteed not to increase, it is possible for the residuals to stagnate and for the convergence criterion never to be satisfied. Therefore, the convergence of the method should be monitored closely.

The two norm of the residual generated by the generalized minimum residual method is obtained during its implementation at no extra cost. However, this is the residual of the system to which the method is applied, which, in the left and split preconditioned case is the preconditioned residual $Q_L^{-1} * r$. To obtain the true residual, a non-negligible amount of extra computation would be required. Hence, for this method, only stopping criteria (12–5) and (12–6) are allowed. Additionally, a user-defined MSTOP is not allowed. In the unpreconditioned case, the stopping criteria default to (12–3) and (12–4), respectively. Thus only $istop$ = 3 and $istop$ = 4 are permitted for the preconditioned case.

The implementation of the generalized minimum residual method requires the routine MATVEC to provide operations for **job**= 0. The routines MATVEC, PCONDL (if used), PCONDR (if used) should be declared external in your calling (sub)program. A user-defined MSTOP is not allowed.

DXML provides the following four forms of the method:

- Unpreconditioned generalized minimum residual method:
  This is the generalized minimum residual method applied to:

  $$A * x = b$$

  where A is a general matrix. As no preconditioning is used, both PCONDL and PCONDR are dummy input parameters.

  For the unpreconditioned generalized minimum residual method, the length of the real work space array, defined by the variable *nrwk* (IPARAM(4)), should be at least:

  $$nrwk = n * (kprev + 1) + kprev * (kprev + 5) + 1$$

  where n is the order of the matrix A and $kprev$ is the number of previous residuals stored.
- Generalized minimum residual method with left preconditioning:
  This is the generalized minimum residual method applied to:

  $$(Q_L^{-1} * A) * x = (Q_L^{-1} * b)$$

  The routine PCONDL, with **job** = 0 should evaluate:

  $$v = Q_L^{-1} * u$$

  The routine PCONDR is not used and is therefore a dummy input parameter.

  For the generalized minimum residual method, with left preconditioning, the length of the real work space array, defined by the variable *nrwk* (IPARAM(4)), should be at least:

  $$nrwk = n * (kprev + 2) + kprev * (kprev + 5) + 1$$

  where n is the order of the matrix A and $kprev$ is the number of previous residuals stored. This does not include the memory requirements of the preconditioner.
- Generalized minimum residual method with right preconditioning:
  This is the generalized minimum residual method applied to:

  $$(A * Q_R^{-1}) * y = b$$

  where:

  $$y = Q_R * x$$

  The routine PCONDR, with **job** = 0 should evaluate:

  $$v = Q_R^{-1} * u$$

  The routine PCONDL is not used.

For the generalized minimum residual method, with right preconditioning, the length of the real work space array, defined by the variable *nrwk* (IPARAM(4)), should be at least:

$$nrwk = n * (kprev + 2) + kprev * (kprev + 5) + 1$$

where n is the order of the matrix A and $kprev$ is the number of previous residuals stored. This does not include the memory requirements of the preconditioner.

- GMRES with split preconditioning: This is the generalized minimum residual method applied to:

$$(Q_L^{-1} * A * Q_R^{-1}) * y = (Q_L^{-1} * b)$$

where:

$$y = Q_R * x$$

The routine PCONDL, with **job**= 0 should evaluate:

$$v = Q_L^{-1} * u$$

and the routine PCONDR, with **job**= 0 should evaluate:

$$v = Q_R^{-1} * u$$

For the generalized minimum residual method, with split preconditioning, the length of the real work space array, defined by the variable *nrwk* (IPARAM(4)), should be at least:

$$nrwk = n * (kprev + 2) + kprev * (kprev + 5) + 1$$

where n is the order of the matrix A and $kprev$ is the number of previous residuals stored. This does not include the memory requirements of the preconditioner.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DITSOL_PTFQMR
Preconditioned Transpose_free Quasiminimal Residual Method (Serial and Parallel Versions)

Format

DITSOL_PTFQMR (matvec, pcondl, pcondr, mstop, a, ia, x, b, n, ql, iql , qr, iqr, iparam, rparam, iwork, rwork, ierror)

Arguments

DITSOL_PTFQMR has the standard parameter list for an iterative solver.

Description

The quasiminimal residual (QMR) method [Freund and Nachtigal 1991] is one of the algorithms proposed as a remedy for the irregular convergence behavior of the bi-conjugate gradient and the conjugate gradient squared algorithms. Since these algorithms are not characterized by a minimization property, the residual norm often oscillates wildly. The QMR algorithm, by generating iterates that are defined by a quasiminimization of the residual norm, results in smooth convergence curves.

DXML includes TFQMR, the transpose-free variant of the QMR method, implemented without look-ahead [Freund 1993]. The implementation of the transpose-free quasiminimal residual method requires the routine MATVEC to provide operations for **job** = 0. The routines MATVEC, PCONDL (if used), PCONDR (if used) and MSTOP (if used) should be declared external in your calling (sub)program.

An upper bound for the two norm of the residual of the system being solved, is obtained during the implementation of the TFQMR method at no extra cost. This is the residual of the system to which the method is applied, which in the left and split preconditioned case is the preconditioned residual, $Q_L^{-1} * r$. To obtain the true residual, a non-negligible amount of extra computation would be required. Hence, for this method, only stopping criteria (12–5) and (12–6) are allowed. In the unpreconditioned case, the stopping criteria default to (12–3) and (12–4), respectively. Thus only $istop$ = 3 and $istop$ = 4 are permitted for both the preconditioned and unpreconditioned case. Additionally, a user-defined MSTOP is allowed, but the vectors r and z, corresponding to the real and preconditioned residuals, respectively, and passed as input parameters to the routine MSTOP, are undefined.

DXML provides the following four forms of the method:

- Unpreconditioned transpose-free, quasiminimal residual method:
  This is the transpose-free, quasiminimal residual method applied to:

  $$A * x = b$$

  where A is a general matrix. As no preconditioning is used, both PCONDL and PCONDR are dummy input parameters.

  For the unpreconditioned transpose-free, quasiminimal residual method, the length of the real work space array, defined by the variable $nrwk$ (IPARAM(4)), should be at least $7 * n$, where n is the order of the matrix A.

The vectors r and z, passed as input arguments to the routine MSTOP, are not defined.

- Transpose-free, quasiminimal residual method with left preconditioning: This is the transpose-free, quasiminimal residual method applied to:

  $$(Q_L^{-1} * A) * x = (Q_L^{-1} * b)$$

  The routine PCONDL, with **job** = 0 should evaluate:

  $$v = Q_L^{-1} * u$$

  The routine PCONDR is not used and is therefore a dummy input parameter.

  For the transpose-free, quasiminimal residual method, with left preconditioning, the length of the real work space array, defined by the variable $nrwk$ (IPARAM(4)), should be at least $8 * n$, where n is the order of the matrix A. This does not include the memory requirements of the preconditioner.

  The vectors r and z, passed as input arguments to the routine MSTOP, are undefined.

- Transpose-free, quasiminimal residual method with right preconditioning: This is the transpose-free, quasiminimal residual method applied to:

  $$(A * Q_R^{-1}) * y = b$$

  where:

  $$y = Q_R * x$$

  The routine PCONDR, with **job** = 0 should evaluate:

  $$v = Q_R^{-1} * u$$

  The routine PCONDL is not used and is therefore a dummy input parameter.

  For the transpose-free, quasiminimal residual method, with right preconditioning, the length of the real work space array, defined by the variable $nrwk$ (IPARAM(4)), should be at least $8 * n$, where n is the order of the matrix A. This does not include the memory requirements of the preconditioner.

  The vectors r and z, passed as input arguments to the routine MSTOP, are undefined.

- Transpose-free, quasiminimal residual method with split preconditioning: This is the transpose-free, quasiminimal residual method applied to:

  $$(Q_L^{-1} * A * Q_R^{-1}) * y = (Q_L^{-1} * b)$$

  where:

  $$y = Q_R * x$$

  The routine PCONDL, with **job** = 0 should evaluate:

  $$v = Q_L^{-1} * u$$

  and the routine PCONDR, with **job** = 0 should evaluate:

  $$v = Q_R^{-1} * u$$

  For the transpose-free, quasiminimal residual method, with split preconditioning, the length of the real work space array, defined by the variable $nrwk$ (IPARAM(4)), should be at least $9 * n$, where n is the order of the matrix A. This does not include the memory requirements of the preconditioner. The vectors r and z, passed as input arguments to the routine MSTOP, are undefined.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DMATVEC_SDIA
Matrix-Vector Product for Symmetric Diagonal Storage (Serial and Parallel Versions)

Format

DMATVEC_SDIA (job, a, ia, ndim, nz, w, x, y, n)

Arguments

job
integer*4
On entry, defines the operation to be performed:

job = 0 : $y = A * x$
job = 1 : $y = A^T * x$
job = 2 : $y = w - A * x$
job = 3 : $y = w - A^T * x$

On exit, **job** is unchanged.

a
real*8
On entry, a two-dimensional array with dimensions $ndim$ by nz containing the nonzero elements of the matrix A.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least nz, containing the distances of the diagonals from the main diagonal.
On exit, **ia** is unchanged.

ndim
integer*4
On entry, the leading dimension of array A, as declared in the calling subprogram; $ndim \geq n$.
On exit, **ndim** is unchanged.

nz
integer*4
On entry, the number of diagonals stored in array A.
On exit, **nz** is unchanged.

w
real*8
On entry, a one-dimensional array of length at least n containing the vector w when **job** = 2 or 3. The elements are accessed with unit increment. When **job** = 0 or 1, array W is not needed so **w** can be a dummy parameter.
On exit, **w** is unchanged.

x
real*8
On entry, a one-dimensional array of length at least n, containing the elements of vector x, accessed with unit increment.
On exit, **x** is unchanged.

y
real*8
On entry, a one-dimensional array of length at least n.
On exit, array Y is overwritten by the output vector y. The elements of array Y are accessed with unit increment.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DMATVEC_SDIA obtains the matrix-vector product for a sparse matrix stored using the symmetric diagonal storage scheme. Depending on the value of the input parameter **job**, either the matrix or its transpose is used in the operation.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DMATVEC_UDIA
Matrix-Vector Product for Unsymmetric Diagonal Storage (Serial and Parallel Versions)

Format

DMATVEC_UDIA (job, a, ia, ndim, nz, w, x, y, n)

Arguments

job
integer*4
On entry, defines the operation to be performed:

job = 0 : $y = A * x$
job = 1 : $y = A^T * x$
job = 2 : $y = w - A * x$
job = 3 : $y = w - A^T * x$

On exit, **job** is unchanged.

a
real*8
On entry, a two-dimensional array with dimensions $ndim$ by nz containing the nonzero elements of the matrix A.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least nz, containing the distances of the diagonals from the main diagonal.
On exit, **ia** is unchanged.

ndim
integer*4
On entry, the leading dimension of array A, as declared in the calling subprogram; $ndim \geq n$.
On exit, **ndim** is unchanged.

nz
integer*4
On entry, the number of diagonals stored in array A.
On exit, **nz** is unchanged.

w
real*8
On entry, a one-dimensional array of length at least n containing the vector w when **job** = 2 or 3. The elements are accessed with unit increment. When **job** = 0 or 1, array W is not needed so **w** can be a dummy parameter.
On exit, **w** is unchanged.

x
real*8
On entry, a one-dimensional array of length at least n, containing the elements of vector x, accessed with unit increment.
On exit, **x** is unchanged.

y
real*8
On entry, a one-dimensional array of length at least n.
On exit, array Y is overwritten by the output vector y. The elements of array Y are accessed with unit increment.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DMATVEC_UDIA obtains the matrix-vector product for a sparse matrix stored using the unsymmetric diagonal storage scheme. Depending on the value of the input parameter **job**, either the matrix or its transpose is used in the operation.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DMATVEC_GENR
Matrix-Vector Product for General Storage by Rows (Serial and Parallel Versions)

Format

DMATVEC_GENR (job, a, ia, ja, nz, w, x, y, n)

Arguments

job
integer*4
On entry, defines the operation to be performed:

job = 0 : $y = A * x$
job = 1 : $y = A^T * x$
job = 2 : $y = w - A * x$
job = 3 : $y = w - A^T * x$

On exit, **job** is unchanged.

a
real*8
On entry, a one-dimensional array of length at least nz containing the nonzero elements of the matrix A.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least $n+1$, containing the starting indices of each row in arrays JA and A.
On exit, **ia** is unchanged.

ja
integer*4
On entry, a one-dimensional array of length at least nz, containing the column values of each nonzero element of the matrix A, stored using the general storage by rows scheme.
On exit, **ja** is unchanged.

nz
integer*4
On entry, the number of nonzero elements stored in array A.
On exit, **nz** is unchanged.

w
real*8
On entry, a one-dimensional array of length at least n containing the vector w when **job** = 2 or 3. The elements are accessed with unit increment. When **job** = 0 or 1, array W is not needed so **w** can be a dummy parameter.
On exit, **w** is unchanged.

x
real*8
On entry, a one-dimensional array of length at least n, containing the elements of vector x, accessed with unit increment.
On exit, **x** is unchanged.

y
real*8
On entry, a one-dimensional array of length at least n.
On exit, array Y is overwritten by the output vector y. The elements of array Y are accessed with unit increment.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DMATVEC_GENR obtains the matrix-vector product for a sparse matrix stored using the general storage by rows scheme. Depending on the value of the input parameter **job**, either the matrix or its transpose is used in the operation.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DCREATE_DIAG_SDIA
Generate Diagonal Preconditioner for Symmetric Diagonal Storage (Serial and Parallel Versions)

Format

DCREATE_DIAG_SDIA (a, ia, ndim, nz, p, n)

Arguments

a
real*8
On entry, a two-dimensional array with dimensions $ndim$ by nz containing the nonzero elements of the matrix A.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least nz, containing the distances of the diagonals from the main diagonal.
On exit, **ia** is unchanged.

ndim
integer*4
On entry, the leading dimension of array A, as declared in the calling subprogram; $ndim \geq n$.
On exit, **ndim** is unchanged.

nz
integer*4
On entry, the number of diagonals stored in array A.
On exit, **nz** is unchanged.

p
real*8
On entry, a one-dimensional array of length at least n.
On exit, array P contains information for use by the diagonal preconditioner.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DCREATE_DIAG_SDIA computes the information required by the diagonal preconditioner for a sparse matrix stored using the symmetric diagonal storage scheme. The real part of this information is returned in the array P. There is no integer information returned for this preconditioner.

The routine DCREATE_DIAG_SDIA is called prior to a call to one of the iterative solver routines with diagonal preconditioning.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DCREATE_DIAG_UDIA
Generate Diagonal Preconditioner for Unsymmetric Diagonal Storage (Serial and Parallel Versions)

Format

DCREATE_DIAG_UDIA (a, ia, ndim, nz, p, n)

Arguments

a
real*8
On entry, a two-dimensional array with dimensions $ndim$ by nz containing the nonzero elements of the matrix A.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least nz, containing the distances of the diagonals from the main diagonal.
On exit, **ia** is unchanged.

ndim
integer*4
On entry, the leading dimension of array A, as declared in the calling subprogram; $ndim \geq n$.
On exit, **ndim** is unchanged.

nz
integer*4
On entry, the number of diagonals stored in array A.
On exit, **nz** is unchanged.

p
real*8
On entry, a one-dimensional array of length at least n.
On exit, array P contains information for use by the diagonal preconditioner.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DCREATE_DIAG_UDIA computes the information required by the diagonal preconditioner for a sparse matrix stored using the unsymmetric diagonal storage scheme. The real part of this information is returned in the array P. There is no integer information returned for this preconditioner.

The routine DCREATE_DIAG_UDIA is called prior to a call to one of the iterative solver routines with diagonal preconditioning.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DCREATE_DIAG_GENR
Generate Diagonal Preconditioner for General Storage by Rows (Serial and Parallel Versions)

Format

DCREATE_DIAG_GENR (a, ia, ja, nz, p, n)

Arguments

a
real*8
On entry, a one-dimensional array of length at least nz containing the nonzero elements of the matrix A.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least $n+1$, containing the starting indices of each row in arrays JA and A.
On exit, **ia** is unchanged.

ja
integer*4
On entry, a one-dimensional array of length at least nz, containing the column values of each nonzero element of the matrix A.
On exit, **ja** is unchanged.

nz
integer*4
On entry, the number of nonzero elements stored in array A.
On exit, **nz** is unchanged.

p
real*8
On entry, a one-dimensional array of length at least n.
On exit, array P contains information for use by the diagonal preconditioner.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DCREATE_DIAG_GENR computes the information required by the diagonal preconditioner for a sparse matrix stored using the general storage by rows scheme. The real part of this information is returned in the array P. There is no integer information returned for this preconditioner.

The routine DCREATE_DIAG_GENR is called prior to a call to one of the iterative solver routines with diagonal preconditioning.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DCREATE_POLY_SDIA
Generate Polynomial Preconditioner for Symmetric Diagonal Storage (Serial and Parallel Versions)

Format

DCREATE_POLY_SDIA (a, ia, ndim, nz, p, n)

Arguments

a
real*8
On entry, a two-dimensional array with dimensions $ndim$ by nz containing the nonzero elements of the matrix A.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least nz, containing the distances of the diagonals from the main diagonal.
On exit, **ia** is unchanged.

ndim
integer*4
On entry, the leading dimension of array A, as declared in the calling subprogram; $ndim \geq n$.
On exit, **ndim** is unchanged.

nz
integer*4
On entry, the number of diagonals stored in array A.
On exit, **nz** is unchanged.

p
real*8
On entry, a one-dimensional array of length at least $3 * n$.
On exit, array P contains information for use by the polynomial preconditioner.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DCREATE_POLY_SDIA computes the information required by the polynomial preconditioner for a sparse matrix stored using the symmetric diagonal storage scheme. The real part of this information is returned in the array P. There is no integer information returned for this preconditioner. Part of the array P is used as workspace during the application of the preconditioner.

The routine DCREATE_POLY_SDIA is called prior to a call to one of the iterative solver routines with polynomial preconditioning.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DCREATE_POLY_UDIA
Generate Polynomial Preconditioner for Unsymmetric Diagonal Storage (Serial and Parallel Versions)

Format

DCREATE_POLY_UDIA (a, ia, ndim, nz, p, n)

Arguments

a
real*8
On entry, a two-dimensional array with dimensions $ndim$ by nz containing the nonzero elements of the matrix A.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least nz, containing the distances of the diagonals from the main diagonal.
On exit, **ia** is unchanged.

ndim
integer*4
On entry, the leading dimension of array A, as declared in the calling subprogram; $ndim \geq n$.
On exit, **ndim** is unchanged.

nz
integer*4
On entry, the number of diagonals stored in array A.
On exit, **nz** is unchanged.

p
real*8
On entry, a one-dimensional array of length at least $3 * n$.
On exit, array P contains information for use by the polynomial preconditioner.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DCREATE_POLY_UDIA computes the information required by the polynomial preconditioner for a sparse matrix stored using the unsymmetric diagonal storage scheme. The real part of this information is returned in the array P. There is no integer information returned for this preconditioner. Part of the array P is used as workspace during the application of the preconditioner.

The routine DCREATE_POLY_UDIA is called prior to a call to one of the iterative solver routines with polynomial preconditioning.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DCREATE_POLY_GENR
Generate Polynomial Preconditioner for General Storage by Rows (Serial and Parallel Versions)

Format

DCREATE_POLY_GENR (a, ia, ja, nz, p, n)

Arguments

a
real*8
On entry, a one-dimensional array of length at least nz containing the nonzero elements of the matrix A.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least $n+1$, containing the starting indices of each row in arrays JA and A.
On exit, **ia** is unchanged.

ja
integer*4
On entry, a one-dimensional array of length at least nz, containing the column values of each nonzero element of the matrix A.
On exit, **ja** is unchanged.

nz
integer*4
On entry, the number of nonzero elements stored in array A.
On exit, **nz** is unchanged.

p
real*8
On entry, a one-dimensional array of length at least $3 * n$.
On exit, array P contains information for use by the diagonal preconditioner.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DCREATE_POLY_GENR computes the information required by the polynomial preconditioner for a sparse matrix stored using the general storage by rows scheme. The real part of this information is returned in the array P. There is no integer information returned for this preconditioner. Part of the array P is used as workspace during the application of the preconditioner.

The routine DCREATE_POLY_GENR is called prior to a call to one of the iterative solver routines with polynomial preconditioning.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DCREATE_ILU_SDIA
Generate Incomplete Cholesky Preconditioner for Symmetric Diagonal Storage

Format

DCREATE_ILU_SDIA (a, ia, ndim, nz, p, ip, n)

Arguments

a
real*8
On entry, a two-dimensional array with dimensions *ndim* by *nz* containing the nonzero elements of the matrix *A*.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least *nz*, containing the distances of the diagonals from the main diagonal.
On exit, **ia** is unchanged.

ndim
integer*4
On entry, the leading dimension of array A, as declared in the calling subprogram; $ndim \geq n$.
On exit, **ndim** is unchanged.

nz
integer*4
On entry, the number of diagonals stored in array A.
On exit, **nz** is unchanged.

p
real*8
On entry, a two-dimensional array with dimensions *ndim* by *nz*.
On exit, array P contains information used by the Incomplete Cholesky preconditioner.

ip
integer*4
On entry, a one-dimensional array of length at least *nz*.
On exit, IP contains information for the Incomplete Cholesky preconditioner.

n
integer*4
On entry, the order of the matrix *A*.
On exit, **n** is unchanged.

Description

DCREATE_ILU_SDIA computes the information required by the Incomplete Cholesky preconditioner for a sparse matrix stored using the symmetric diagonal storage scheme. The arrays P and IP contain the real and integer information, respectively, for use by the preconditioner.

If the lower triangular part of the matrix A is stored, the decomposition is LL^T, where L is a lower triangular matrix. If the upper triangular part is stored, the decomposition is $U^T * U$, where U is an upper triangular matrix.

The routine DCREATE_ILU_SDIA is called prior to a call to one of the iterative solver routines with Incomplete Cholesky preconditioning.

DCREATE_ILU_UDIA
Generate Incomplete LU Preconditioner for Unsymmetric Diagonal Storage

Format

DCREATE_ILU_UDIA (a, ia, ndim, nz, plu, iplu, n)

Arguments

a
real*8
On entry, a two-dimensional array with dimensions $ndim$ by nz containing the nonzero elements of the matrix A.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least nz, containing the distances of the diagonals from the main diagonal.
On exit, **ia** is unchanged.

ndim
integer*4
On entry, the leading dimension of array A, as declared in the calling subprogram; $ndim \geq n$.
On exit, **ndim** is unchanged.

nz
integer*4
On entry, the number of diagonals stored in array A.
On exit, **nz** is unchanged.

plu
real*8
On entry, a two-dimensional array with dimensions $ndim$ by nz.
On exit, array PLU contains information used by the Incomplete LU preconditioner.

iplu
integer*4
On entry, a one-dimensional array of length at least nz.
On exit, array IPLU contains information used by the Incomplete LU preconditioner.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DCREATE_ILU_UDIA computes the information required by the Incomplete LU preconditioner for a sparse matrix stored using the unsymmetric diagonal storage scheme. The arrays PLU and IPLU contain the real and integer information, respectively, for use by the preconditioner.

The routine DCREATE_ILU_UDIA is called prior to a call to one of the iterative solver routines with Incomplete LU preconditioning.

DCREATE_ILU_GENR
Generate Incomplete LU Preconditioner for General Storage by Rows

Format

DCREATE_ILU_GENR (a, ia, ja, nz, plu, n)

Arguments

a
real*8
On entry, a one-dimensional array of length at least nz containing the nonzero elements of the matrix A.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least $n+1$, containing the starting indices of each row in arrays JA and A.
On exit, **ia** is unchanged.

ja
integer*4
On entry, a one-dimensional array of length at least nz, containing the column values of each nonzero element of the matrix A.
On exit, **ja** is unchanged.

nz
integer*4
On entry, the number of nonzero elements in array A.
On exit, **nz** is unchanged.

plu
real*8
On entry, a one-dimensional array of length at least nz.
On exit, array PLU contains information used by the Incomplete LU preconditioner.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DCREATE_ILU_GENR computes the information required by the Incomplete LU preconditioner for a sparse matrix stored using the general storage by rows scheme. The array PLU contains the real information for use by the preconditioner. The integer information is identical to the information in arrays IA and JA and is therefore not generated.

The routine DCREATE_ILU_GENR is called prior to a call to one of the iterative solver routines with Incomplete LU preconditioning.

DAPPLY_DIAG_ALL
Apply Diagonal Preconditioner for Any Storage Scheme (Serial and Parallel Versions)

Format

DAPPLY_DIAG_ALL (p, x, y, n)

Arguments

p
real*8
On entry, a one-dimensional array of length at least n containing information for use by the polynomial preconditioner.
On exit, **p** is unchanged.

x
real*8
On entry, a one-dimensional array of length at least n, containing the elements of vector x, accessed with unit increment.
On exit, **x** is unchanged.

y
real*8
On entry, a one-dimensional array of length at least n.
On exit, array Y is overwritten by $Q^{-1} * x$, where Q is the diagonal preconditioner. The elements of Y are accessed with unit increment.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DAPPLY_DIAG_ALL applies the diagonal preconditioner for a sparse matrix stored using any one of the three storage schemes — symmetric diagonal, unsymmetric diagonal, or general storage by rows. The input vector, p, contains information for use by the routine. This vector is generated by a call to one of the routines DCREATE_DIAG_SDIA, DCREATE_DIAG_UDIA or DCREATE_DIAG_GENR prior to a call to one of the iterative solvers with diagonal preconditioning.

DAPPLY_DIAG_ALL applies the diagonal preconditioner, Q, using information stored in the vector p, to the vector x and returns the result in vector y:

$$y \leftarrow Q^{-1} * x$$

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DAPPLY_POLY_SDIA
Apply Polynomial Preconditioner for Symmetric Diagonal Storage (Serial and Parallel Versions)

Format

DAPPLY_POLY_SDIA (job, p, a, ia, ndim, nz, x, y, ndeg, n)

Arguments

job
integer*4
On entry, defines the operation to be performed:

job = 0 : $y = Q^{-1} * x$
job = 1 : $y = Q^{-T} * x$

where Q is the polynomial preconditioner.
On exit, **job** is unchanged.

p
real*8
On entry, a one-dimensional array of length at least $3*n$ that contains information for use by the polynomial preconditioner and workspace.
On exit, the part of array P that contains the information related to the polynomial preconditioner is unchanged. The part used as workspace is overwritten.

a
real*8
On entry, a two-dimensional array with dimensions $ndim$ by nz containing the nonzero elements of the matrix A.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least nz, containing the distances of the diagonals from the main diagonal.
On exit, **ia** is unchanged.

ndim
integer*4
On entry, the leading dimension of array A, as declared in the calling subprogram; $ndim \geq n$.
On exit, **ndim** is unchanged.

nz
integer*4
On entry, the number of diagonals stored in array A.
On exit, **nz** is unchanged.

x
real*8
On entry, a one-dimensional array of length at least n, containing the elements of vector x, accessed with unit increment.
On exit, **x** is unchanged.

y
real*8
On entry, a one-dimensional array of length at least n.
On exit, array Y is overwritten by the output vector y. The elements of array Y are accessed with unit increment.

ndeg
integer*4
On entry, the degree of the polynomial in the polynomial preconditioner.
On exit, **ndeg** is unchanged.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DAPPLY_POLY_SDIA applies the polynomial preconditioner for a sparse matrix stored using the symmetric diagonal storage scheme. The input vector, p, contains information for use by the routine. This vector is generated by a call to the routine DCREATE_POLY_SDIA prior to a call to one of the iterative solvers with polynomial preconditioning. Depending on the value of the input parameter **job**, DAPPLY_POLY_SDIA operates on either the preconditioning matrix or its transpose.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DAPPLY_POLY_UDIA
Apply Polynomial Preconditioner for Unsymmetric Diagonal Storage (Serial and Parallel Versions)

Format

DAPPLY_POLY_UDIA (job, p, a, ia, ndim, nz, x, y, ndeg, n)

Arguments

job
integer*4
On entry, defines the operation to be performed:

job = 0 : $y = Q^{-1} * x$
job = 1 : $y = Q^{-T} * x$

where Q is the polynomial preconditioner.
On exit, **job** is unchanged.

p
real*8
On entry, a one-dimensional array of length at least $3*n$ that contains information for use by the polynomial preconditioner and workspace.
On exit, the part of array P that contains the information related to the polynomial preconditioner is unchanged. The part used as workspace is overwritten.

a
real*8
On entry, a two-dimensional array with dimensions $ndim$ by nz containing the nonzero elements of the matrix A.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least nz, containing the distances of the diagonals from the main diagonal.
On exit, **ia** is unchanged.

ndim
integer*4
On entry, the leading dimension of array A, as declared in the calling subprogram; $ndim \geq n$.
On exit, **ndim** is unchanged.

nz
integer*4
On entry, the number of diagonals stored in array A.
On exit, **nz** is unchanged.

x
real*8
On entry, a one-dimensional array of length at least n, containing the elements of vector x, accessed with unit increment.
On exit, **x** is unchanged.

y
real*8
On entry, a one-dimensional array of length at least n.
On exit, array Y is overwritten by the output vector y. The elements of array Y are accessed with unit increment.

ndeg
integer*4
On entry, the degree of the polynomial in the polynomial preconditioner.
On exit, **ndeg** is unchanged.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DAPPLY_POLY_UDIA applies the polynomial preconditioner for a sparse matrix stored using the unsymmetric diagonal storage scheme. The input vector, p, contains information for use by the routine. This vector is generated by a call to the routine DCREATE_POLY_UDIA prior to a call to one of the iterative solvers with polynomial preconditioning. Depending on the value of the input parameter **job**, DAPPLY_POLY_UDIA operates on either the preconditioning matrix or its transpose.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DAPPLY_POLY_GENR
Apply Polynomial Preconditioner for General Storage by Rows (Serial and Parallel Versions)

Format

DAPPLY_POLY_GENR (job, p, a, ia, ja, nz, x, y, ndeg, n)

Arguments

job
integer*4
On entry, defines the operation to be performed:

job = 0 : $y = Q^{-1} * x$
job = 1 : $y = Q^{-T} * x$

where Q is the polynomial preconditioner.
On exit, **job** is unchanged.

p
real*8
On entry, a one-dimensional array of length at least $3*n$ that contains information for use by the polynomial preconditioner and workspace.
On exit, the part of array P that contains the information related to the polynomial preconditioner is unchanged. The part used as workspace is overwritten.

a
real*8
On entry, a one-dimensional array of length at least nz containing the nonzero elements of the matrix A.
On exit, **a** is unchanged.

ia
integer*4
On entry, a one-dimensional array of length at least $n+1$, containing the starting indices of each row in arrays JA and A.
On exit, **ia** is unchanged.

ja
integer*4
On entry, a one-dimensional array of length at least nz, containing the column values of each nonzero element of the matrix A.
On exit, **ja** is unchanged.

nz
integer*4
On entry, the number of nonzero elements stored in array A.
On exit, **nz** is unchanged.

x
real*8
On entry, a one-dimensional array of length at least n, containing the elements of vector x, accessed with unit increment.
On exit, **x** is unchanged.

y
real*8
On entry, a one-dimensional array of length at least n.
On exit, array Y is overwritten by the output vector y. The elements of array Y are accessed with unit increment.

ndeg
integer*4
On entry, the degree of the polynomial in the polynomial preconditioner.
On exit, **ndeg** is unchanged.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DAPPLY_POLY_GENR applies the polynomial preconditioner for a sparse matrix stored using the general storage by rows scheme. The input vector, p, contains information for use by the routine. This vector is generated by a call to the routine DCREATE_POLY_GENR prior to a call to one of the iterative solvers with polynomial preconditioning. Depending on the value of the input parameter **job**, DAPPLY_POLY_GENR operates on either the preconditioning matrix or its transpose.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DAPPLY_ILU_SDIA
Apply ILU Preconditioner for Symmetric Diagonal Storage

Format

DAPPLY_ILU_SDIA (job, p, ip, ndim, nz, x, y, n)

Arguments

job
integer*4
On entry, defines the operation to be performed. If the lower triangular part of the matrix A is stored, the preconditioner is of the form $L * L^T$, where L is a lower triangular matrix. In this case:

job = 0 : $y = L^{-1} * x$
job = 1 : $y = L^{-T} * x$

If the upper triangular part of the matrix A is stored, the preconditioner is of the form $U^T * U$, where U is an upper triangular matrix. In this case:

job = 0 : $y = U^{-1} * x$
job = 1 : $y = U^{-T} * x$

On exit, **job** is unchanged.

p
real*8
On entry, a two-dimensional array with dimensions $ndim$ by nz containing information for use by the Incomplete Cholesky preconditioner.
On exit, **p** is unchanged.

ip
integer*4
On entry, a one-dimensional array of length at least nz, containing the distances of the diagonals in array P from the main diagonal.
On exit, **ip** is unchanged.

ndim
integer*4
On entry, the leading dimension of array A, as declared in the calling subprogram; $ndim \geq n$.
On exit, **ndim** is unchanged.

nz
integer*4
On entry, the number of diagonals stored in array P.
On exit, **nz** is unchanged.

x
real*8
On entry, a one-dimensional array of length at least n, containing the elements of vector x, accessed with unit increment.
On exit, **x** is unchanged.

y
real*8
On entry, a one-dimensional array of length at least n.
On exit, array Y is overwritten by the output vector y. The elements of array Y are accessed with unit increment.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DAPPLY_ILU_SDIA applies the Incomplete Cholesky preconditioner for a sparse matrix stored using the symmetric diagonal storage scheme. The input arrays, P and IP, contain information for use by the routine. These arrays are generated by a call to the routine DCREATE_ILU_SDIA prior to a call to one of the iterative solvers with Incomplete Cholesky preconditioning.

Depending on the value of the input parameter **job**, DAPPLY_ILU_SDIA operates on either the matrix or its transpose. The symmetric diagonal storage scheme, SDIA, allows either the lower or upper triangular part of the matrix to be stored. If the lower triangular part is stored, the routine DCREATE_ILU_SDIA creates the incomplete Cholesky preconditioner in the form $L * L^T$, where L is a lower triangular matrix. In this case, **job** = 0 implies:

$$y = L^{-1} * x$$

and **job** = 1 implies:

$$y = L^{-T} * x$$

If the upper triangular part of the matrix A is stored, the routine DCREATE_ILU_SDIA creates the incomplete Cholesky preconditioner in the form $U^T * U$, where U is an upper triangular matrix. In this case, **job** = 0 implies:

$$y = U^{-1} * x$$

and **job** = 1 implies:

$$y = U^{-T} * x$$

DAPPLY_ILU_UDIA_L
Apply ILU Preconditioner for Unsymmetric Diagonal Storage

Format

DAPPLY_ILU_UDIA_L (job, plu, iplu, ndim, nz, x, y, n)

Arguments

job
integer*4
On entry, defines the operation to be performed:

job = 0 : $y = L^{-1} * x$
job = 1 : $y = L^{-T} * x$

where the incomplete factorization is calculated as $L * U$. L and U are lower and upper triangular matrices, respectively.
On exit, **job** is unchanged.

plu
real*8
On entry, a two-dimensional array with dimensions $ndim$ by nz, containing information for use by the ILU preconditioner.
On exit, **plu** is unchanged.

iplu
integer*4
On entry, a one-dimensional array of length at least nz, containing the distances of the diagonals of PLU from the main diagonal.
On exit, **iplu** is unchanged.

ndim
integer*4
On entry, the leading dimension of array A, as declared in the calling subprogram; $ndim \geq n$.
On exit, **ndim** is unchanged.

nz
integer*4
On entry, the number of diagonals stored in array PLU.
On exit, **nz** is unchanged.

x
real*8
On entry, a one-dimensional array of length at least n, containing the elements of vector x, accessed with unit increment.
On exit, **x** is unchanged.

y
real*8
On entry, a one-dimensional array of length at least n.
On exit, array Y is overwritten by the output vector y. The elements of array Y are accessed with unit increment.

n
integer*4
On entry, the order of the matrix ***A***.
On exit, **n** is unchanged.

Description

DAPPLY_ILU_UDIA_L applies the Incomplete LU preconditioner (lower triangular part) for a sparse matrix stored using the unsymmetric diagonal storage scheme. The input arrays, PLU and IPLU, contain information for use by the routine. These arrays are generated by a call to the routine DCREATE_ILU_UDIA prior to a call to one of the solvers with Incomplete LU preconditioning. Depending on the value of the input parameter **job**, DAPPLY_ILU_UDIA_L operates on either the matrix or its transpose.

DAPPLY_ILU_UDIA_U
Apply ILU Preconditioner for Unsymmetric Diagonal Storage

Format

DAPPLY_ILU_UDIA_U (job, plu, iplu, ndim, nz, x, y, n)

Arguments

job
integer*4
On entry, defines the operation to be performed:

job = 0 : $y = U^{-1} * x$
job = 1 : $y = U^{-T} * x$

where the incomplete factorization is calculated as $L * U$. L and U are lower and upper triangular matrices respectively.
On exit, **job** is unchanged.

plu
real*8
On entry, a two-dimensional array with dimensions $ndim$ by nz containing information used by the Incomplete LU preconditioner.
On exit, **plu** is unchanged.

iplu
integer*4
On entry, a one-dimensional array of length at least nz, containing the distances of the diagonals of PLU from the main diagonal.
On exit, **iplu** is unchanged.

ndim
integer*4
On entry, the leading dimension of array A, as declared in the calling subprogram; $ndim \geq n$.
On exit, **ndim** is unchanged.

nz
integer*4
On entry, the number of diagonals stored in arrays PLU and A.
On exit, **nz** is unchanged.

x
real*8
On entry, a one-dimensional array of length at least n, containing the elements of vector x, accessed with unit increment.
On exit, **x** is unchanged.

y
real*8
On entry, a one-dimensional array of length at least n.
On exit, array Y is overwritten by the output vector y. The elements of array Y are accessed with unit increment.

n
integer*4
On entry, the order of the matrix ***A***.
On exit, **n** is unchanged.

Description

DAPPLY_ILU_UDIA_U applies the Incomplete LU preconditioner (upper triangular part) for a sparse matrix stored using the unsymmetric diagonal storage scheme. The input arrays, PLU and IPLU, contain information for use by the routine. These arrays are generated by a call to the routine DCREATE_ILU_UDIA prior to a call to one of the solvers with Incomplete LU preconditioning. Depending on the value of the input parameter **job**, DAPPLY_ILU_UDIA_U operates on either the matrix or its transpose.

DAPPLY_ILU_GENR_L
Apply Incomplete LU Preconditioner for General Storage by Rows

Format

DAPPLY_ILU_GENR_L (job, plu, iplu, jplu, nz, x, y, n)

Arguments

job
integer*4
On entry, defines the operation to be performed:

job = 0 : $y = L^{-1} * x$
job = 1 : $y = L^{-T} * x$

where the incomplete factorization is calculated as $L * U$. L and U are lower and upper triangular matrices, respectively.
On exit, **job** is unchanged.

plu
real*8
On entry, a one-dimensional array of length at least nz containing information used by the Incomplete LU preconditioner.
On exit, **plu** is unchanged.

iplu
integer*4
On entry, a one-dimensional array of length at least $n+1$, containing the starting indices of each row in array PLU and JPLU. IPLU is identical to the array IA used in the storage of the matrix A.
On exit, **iplu** is unchanged.

jplu
integer*4
On entry, a one-dimensional array of length at least nz, containing the column values of each nonzero element of the matrix PLU, stored using the general storage by rows scheme. JPLU is identical to the array JA used in the storage of the matrix A.
On exit, **jplu** is unchanged.

nz
integer*4
On entry, the number of nonzero elements stored in array PLU.
On exit, **nz** is unchanged.

x
real*8
On entry, a one-dimensional array of length at least n, containing the elements of vector x, accessed with unit increment.
On exit, **x** is unchanged.

y
real*8
On entry, a one-dimensional array of length at least n.

On exit, array Y is overwritten by the output vector y. The elements of array Y are accessed with unit increment.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DAPPLY_ILU_GENR_L applies the Incomplete LU preconditioner (lower triangular part) for a sparse matrix stored using the general storage by rows scheme. The input arrays, PLU, IPLU and JPLU, contain information for use by the routine. These arrays are generated by a call to the routine DCREATE_ILU_GENR prior to a call to one of the iterative solvers with Incomplete LU preconditioning. The information in the arrays IPLU and JPLU is identical to the information in the arrays IA and JA. Therefore, routine DCREATE_ILU_GENR does not generate these arrays.

Depending on the value of the input parameter **job**, DAPPLY_ILU_GENR_L operates on either the matrix or its transpose.

DAPPLY_ILU_GENR_U
Apply Incomplete LU Preconditioner for General Storage by Rows

Format

DAPPLY_ILU_GENR_U (job, plu, iplu, jplu, nz, x, y, n)

Arguments

job
integer*4
On entry, defines the operation to be performed:

job = 0 : $y = U^{-1} * x$
job = 1 : $y = U^{-T} * x$

where the incomplete factorization is calculated as $L * U$. L and U are lower and upper triangular matrices, respectively.
On exit, **job** is unchanged.

plu
real*8
On entry, a one-dimensional array of length at least nz containing information used by the Incomplete LU preconditioner.
On exit, **plu** is unchanged.

iplu
integer*4
On entry, a one-dimensional array of length at least $n+1$, containing the starting indices of each row in array PLU and JPLU. IPLU is identical to the array IA used in the storage of the matrix A.
On exit, **iplu** is unchanged.

jplu
integer*4
On entry, a one-dimensional array of length at least nz, containing the column values of each nonzero element of the matrix PLU, stored using the general storage by rows scheme. JPLU is identical to the array JA used in the storage of the matrix A in the general storage by rows scheme.
On exit, **jplu** is unchanged.

nz
integer*4
On entry, the number of nonzero elements stored in array PLU.
On exit, **nz** is unchanged.

x
real*8
On entry, a one-dimensional array of length at least n, containing the elements of vector x, accessed with unit increment.
On exit, **x** is unchanged.

y
real*8
On entry, a one-dimensional array of length at least n.

On exit, array Y is overwritten by the output vector y. The elements of array Y are accessed with unit increment.

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

Description

DAPPLY_ILU_GENR_U applies the Incomplete LU preconditioner (upper triangular part) for a sparse matrix stored using the general storage by rows scheme. The input arrays, PLU, IPLU and JPLU, contain information for use by the routine. These arrays are generated by a call to the routine DCREATE_ILU_GENR prior to a call to one of the iterative solvers with Incomplete LU preconditioning. The information in the arrays IPLU and JPLU is identical to the information in the arrays IA and JA. Therefore, the routine DCREATE_ILU_GENR does not generate these arrays.

Depending on the value of the input parameter **job**, DAPPLY_ILU_GENR_U operates on either the matrix or its transpose.

13

Using the Direct Solvers for Sparse Linear Systems

DXML provides subprograms for the direct solution of sparse linear systems of equations. At present, the direct solvers are provided only for matrices stored using the skyline storage scheme.

This chapter provides information on the following topics:

- Introduction to linear system solvers (Section 13.1)
- Introduction to direct solvers (Section 13.2)
- Introduction to the skyline solver (Section 13.3)
- Storage schemes for sparse matrices stored in the skyline storage format — symmetric matrices and unsymmetric matrices (Section 13.4)
- Functionality provided by the skyline solvers (Section 13.5)
- Naming conventions (Section 13.6) and summary of the skyline solver subprograms (Section 13.7)
- Error handling for the skyline solvers (Section 13.8)
- Suggestions on the use of the skyline solvers (Section 13.9)
- A look at some skyline solvers (Section 13.10)

Descriptions of the sparse direct solver subprograms appear at the end of this chapter.

Two key skyline solver subprograms, DSSKYF and DUSKYF, have been parallelized for improved peformance on multiprocessor systems. For information about using the parallel library, see Chapter 4.

13.1 Introduction

Many applications in science and engineering require the solution of linear systems of equations as follows:

$$Ax = b \qquad (13\text{–}1)$$

In this equation, A is an n by n matrix and x and b are vectors of length n.

Often, these systems occur in the innermost loop of the application, and for good overall performance of the application, it is essential that the linear system solver be efficient. Depending on the application, the system may be solved either once, or many times with different right sides.

The linear systems of equations that arise from science and engineering applications are usually sparse; that is, the coefficient matrix A has a large number of zero elements. You can realize substantial savings in compute time and memory requirements by storing and operating on only the nonzero elements of A. Solution techniques that exploit this sparsity of the matrix A are referred to as sparse solvers.

Methods for the solution of linear systems of equations can be classified into two categories:

- Iterative Methods

  These methods start with an initial guess to the solution, and proceed to calculate solution vectors that approach the exact solution with each iteration. The process is stopped when a given convergence criterion is satisfied. The number of iteration steps required for convergence varies with the coefficient matrix, the initial guess and the convergence criterion — thus, an apriori estimate of the number of operations is not possible.

  See Chapter 12 for details about iterative methods.

- Direct Methods

  These methods first factor the coefficient matrix A into its triangular factors and then perform a forward and backward solve with the triangular factors to get the required solution. The solution is obtained in a finite number of operations, usually known apriori, and is guaranteed to be as accurate as the problem definition.

13.2 Describing the Direct Method

In a direct method, the matrix A is factored as follows:

$$A = LDU \tag{13–2}$$

In this equation, L is a unit lower triangular matrix, D is a diagonal matrix, and U is a unit upper triangular matrix. The system (13–1) is then solved for x by solving the following systems in order:

$$Lz = b$$

$$Dv = z$$

and:

$$Ux = v$$

In the previous equations, z and v are vectors of length n. In the case of a symmetric matrix A, the triangular factorization (13–2) has the following form:

$$A = U^T DU \tag{13–3}$$

The solution is obtained by solving for z, v and x as follows:

$$U^T z = b$$

$$Dv = z$$

and:

$$Ux = v$$

The process of triangular factorization of the matrix A introduces numerical errors due to roundoff or truncation. This can result in the growth of errors as the algorithm progresses. Unless excessive growth in these errors is checked, the accuracy of the solution may be seriously impaired, even resulting in a complete loss of significance. It is possible to minimize this growth by suitably scaling the original matrix A and by choosing large pivot elements in the course of the factorization. An appropriate choice of the pivot may require the interchange of the rows and columns of the matrix. This is acheived via the use of permutation matrices P and Q such that instead of solving the system (13–2), the following equivalent system is solved:

$$(PAQ)(Q^T x) = Pb$$

If A is a symmetric matrix, the equivalent system solved is as follows:

$$(PAP^T)(Px) = Pb$$

The matrices P and Q are obtained during the factorization process and are dependent on the numerical values of the elements of the matrix A.

In the case of a sparse matrix A, the triangular factorization (13–2) or (13–3) can give rise to L and U factors that are not sparse. The additional nonzero entries in the matrices L and U, in positions where the corresponding entry of the matrix A is zero, are referred to as fill-in.

As fill-in increases both the memory requirements as well as the compute time, it should be minimized. This can be achieved again by the use of permutation matrices that reorder the variables such that the factors L and U are appropriately sparse.

It is sometimes possible to perform this reordering based on the nonzero structure of the matrix, without explicit knowledge of the values of the nonzero elements. Therefore, ordering for preservation of sparsity can be done prior to the factorization of the matrix. Depending on the data structures used for storing the matrix, the reordering of the variables can be chosen to minimize specific attributes such as the fill-in, the profile of the matrix, the bandwidth of the matrix and so forth.

Thus, the permutation matrices P and Q are chosen to achieve either one, preferably both, of the following goals:

- Numerical stability and accuracy of the solution procedure
- Preservation of the sparsity of the original matrix A

Orderings that preserve the sparsity also provide an improvement in the accuracy of the solution by minimizing the number of operations performed on each element.

There are different versions of direct methods depending on the way the sparse matrix is stored and sparsity is exploited. These storage schemes are often reflective of the applications that give rise to the matrices. DXML currently provides solvers for matrices stored using the envelope or skyline data structure.

Further details about direct solvers can be found in George and Liu 1981, Pissanetzky 1984, and Duff, Erisman, and Reid, 1986.

13.3 Skyline Solvers

Skyline solvers [Jennings 1966, Felippa 1975] are also referred to as variable band, profile, or envelope solvers. They exploit the sparsity of the matrix by making use of the property that zero elements situated before the first nonzero element in any row or column always remain zero during the factorization, assuming no row or column interchanges are included for numerical stability.

Thus, only the elements to the right of the first nonzero element in a row and below the first nonzero element in a column need to be stored. Zero elements within this variable band are also stored explicitly as this part of the matrix usually fills in totally during factorization. As a result, it is possible to use a static data structure during the factorization and the solution of the linear system.

Skyline, or profile matrices, are a special case of banded matrices where the nonzero structure is exploited further by considering each row or column to have a variable bandwidth. For example, consider the symmetric matrix A with a lower triangular part:

$$\begin{bmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ 0 & 0 & a_{22} & & \\ 0 & a_{42} & 0 & a_{44} & \\ 0 & a_{52} & 0 & 0 & a_{55} \end{bmatrix} \tag{13–4}$$

Let $f_i(A)$ denote the column number of the first nonzero element in row i, as in the following:

$$f_i(A) = min\{j | a_{ij} \neq 0\}$$

The bandwidth of the row i is then defined as follows:

$$\beta_i(A) = i - f_i(A)$$

Thus, the following exists:

$$f_4(A) = 2 \text{ and } \beta_4(A) = 2$$

Skyline solvers take advantage of the variation in $\beta_4(A)$ across the rows and store only the profile or the envelope of A, $Env(A)$, as follows:

$$Env(A) = \{(i,j) | f_i(A) \leq j \leq i\}$$

Thus, in addition to the diagonal elements, the lower triangular part of A requires the storage of elements in locations (2,1), (4,2), (4,3), (5,2), (5,3) and (5,4).

These are essentially the elements in each row, starting with the first nonzero element and moving right to the diagonal element. Any zero elements in a row, between the first nonzero element and the diagonal, are stored explicitly. These elements (including the diagonal) define the width of the row.

In the case of a nonsymmetric matrix, the elements from the first nonzero entry in a column to the diagonal entry are also stored. This includes any zero elements in the column, between the first nonzero element and the diagonal. These elements (including the diagonal) define the height of the column. The number of elements within the envelope of A is called the profile or the envelope size.

13.4 Storage of Skyline Matrices

DXML provides skyline solvers for both symmetric and nonsymmetric matrices. In each case, the matrix can be stored in one of the following two storage modes:

- Profile-in storage mode
- Diagonal-out storage mode

Both modes have identical storage requirements and in addition to the elements in the envelope, also store pointers to the diagonal elements. Additionally, DXML provides a profile-in storage mode for matrices that are structurally symmetric, but numerically unsymmetric.

13.4.1 Symmetric Matrices

Symmetric matrices are stored in a skyline storage scheme using either the profile-in or the diagonal-out mode. The elements in a row (or column) of the lower (or upper) triangular part of the matrix are stored contiguously, starting either from the profile and moving towards the diagonal (profile-in mode) or starting from the diagonal and moving outward to the profile (diagonal-out mode).

13.4.1.1 Profile-in Storage Mode

The profile-in storage scheme stores the symmetric matrix A in two arrays, as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & & & \\ a_{12} & a_{22} & & a_{24} & a_{25} \\ & & a_{33} & 0 & 0 \\ & a_{24} & 0 & a_{44} & 0 \\ & a_{25} & 0 & 0 & a_{55} \end{bmatrix} \tag{13–5}$$

The two arrays are a real array AU that contains the variable banded columns of the upper triangular part of A and an integer array IAUDIAG that contains the pointers to the diagonal elements in the array AU.

As the scheme is a profile-in scheme, the elements in a column of the upper triangular part are stored starting from the first nonzero element in the column and moving down to the diagonal element. The data for each column are stored in consecutive locations, the columns are stored in order and there is no space between the columns.

As A is a symmetric matrix, this can also be interpreted as storing the elements of the lower triangular part starting with the first nonzero element in a row and moving across to the diagonal element. Thus, the matrix in (13–5) is stored in the following arrays:

$$\mathrm{AU} = (a_{11}, a_{12}, a_{22}, a_{33}, a_{24}, 0, a_{44}, a_{25}, 0, 0, a_{55})$$

and:

$$\mathrm{IAUDIAG} = (1, 3, 4, 7, 11)$$

IAUDIAG is of length at least n, where n is the order of the matrix A and AU is of length at least nau, where nau is the envelope size of the symmetric part of A. Thus, for $a_{ij} \in Env(A)$, the following exists:

$$a_{jj} = \mathrm{AU}(\mathrm{IAUDIAG}(j)) \text{ for } 1 \leq j \leq n$$
$$a_{ij} = \mathrm{AU}(\mathrm{IAUDIAG}(j) - j + i) \text{ for } i < j$$

and:

$$nau = 11$$

If all the elements in the column to be stored are zero, then the diagonal element having a value of zero is stored for that column.

13.4.1.2 Diagonal-out Storage Mode

The elements of the upper triangular part of the symmetric matrix A in (13–5) can also be stored by columns in order, starting with the diagonal element and moving up the column to the first nonzero element in the column.

Alternatively, the lower triangular part of the matrix can be considered as being stored by rows in order, with each row stored starting from the diagonal element and moving left to the first nonzero element in the row. The data for each column are stored in consecutive locations, the columns are stored in order and there is no space between columns. Thus, the matrix in (13–5) is stored as:

$$\mathrm{AU} = (a_{11}, a_{22}, a_{12}, a_{33}, a_{44}, 0, a_{24}, a_{55}, 0, 0, a_{25})$$

and:

$$\mathrm{IAUDIAG} = (1, 2, 4, 5, 8, 12)$$

Array AU is of length at least nau, where nau is the envelope size of the symmetric part of A. Array IAUDIAG is of length at least $n + 1$, where the $(n + 1)$-st element is the pointer to the location, in AU, of the diagonal entry of row $n + 1$, if there had been such a row. This allows the determination of the location of the first nonzero entry in the n-th column of A. Thus, for $a_{ij} \in Env(A)$, the following exists:

$$a_{jj} = \mathrm{AU}(\mathrm{IAUDIAG}(j)) \text{ for } 1 \leq j \leq n$$

$$a_{ij} = \mathrm{AU}(\mathrm{IAUDIAG}(j) + j - i) \text{ for } i < j$$

and:

$$nau = 11$$

If all the elements in the column to be stored are zero, then the diagonal element having a value of zero is stored for that column.

13.4.2 Unsymmetric Matrices

Unsymmetric matrices are stored in a skyline storage scheme using either the profile-in or the diagonal-out mode. The elements in a row (and column) of the lower (and upper) triangular part of the matrix are stored contiguously, starting either from the profile and moving towards the diagonal (profile-in mode) or starting from the diagonal and moving outward to the profile (diagonal-out mode). The lower and upper triangular parts are stored in separate arrays, with the diagonal stored in the upper triangular part.

DXML also provides a special storage scheme for matrices that are structurally symmetric, but numerically unsymmetric. Such matrices can also be stored using the general scheme, but at a cost of higher memory requirements.

13.4.2.1 Profile-in Storage Mode

The profile-in storage scheme stores the unsymmetric matrix A in four arrays, as follows:

$$A = \begin{bmatrix} a_{11} & & & & \\ a_{21} & a_{22} & a_{23} & & a_{25} \\ & & a_{33} & a_{34} & 0 \\ & a_{42} & 0 & a_{44} & 0 \\ & a_{52} & 0 & 0 & a_{55} \end{bmatrix} \tag{13–6}$$

The two real arrays AU and AL contain the columns of the upper triangular part and rows of the lower triangular part of A, respectively. The two integer arrays IAUDIAG and IALDIAG contain the pointers to the diagonal elements in the arrays AU and AL, respectively.

As the scheme is a profile-in scheme, the elements in a column of the upper triangular part are stored starting from the first nonzero element in the column and moving down to the diagonal element. Similarly, the elements in a row of the lower triangular part are stored starting with the first nonzero element in a row and moving across to the diagonal element. The data for each row and column are stored consecutively, the rows and columns are stored in order, and there is no space between successive rows or columns.

The diagonal of A is stored along with the upper triangular part in the array AU. The array AL also has storage for the diagonal elements, but these locations are not accessed and they can be used to store other information. Thus, the matrix A is stored as follows:

$$\mathrm{AU} = (a_{11}, a_{22}, a_{23}, a_{33}, a_{34}, a_{44}, a_{25}, 0, 0, a_{55})$$

$$\mathrm{AL} = (*, a_{21}, *, *, a_{42}, 0, *, a_{52}, 0, 0, *)$$

$$\mathrm{IAUDIAG} = (1, 2, 4, 6, 10)$$

and:

$$\mathrm{IALDIAG} = (1, 3, 4, 7, 11)$$

The diagonal elements in AL are indicated by * — the elements in the array IALDIAG are pointers to these locations. IAUDIAG and IALDIAG are of length at least n, where n is the order of the matrix A and AU and AL are of lengths at least nau and nal, respectively, where nau is the envelope size of the upper triangular part of A (including the diagonal) and nal is the envelope size of the lower triangular part of A (including the diagonal). Thus, for $a_{ij} \in Env(A)$, the following exists:

$$a_{jj} = \mathrm{AU}(\mathrm{IAUDIAG}(j)) \text{ for } 1 \leq j \leq n$$

$$a_{ij} = \mathrm{AU}(\mathrm{IAUDIAG}(j) - j + i) \text{ for } i < j$$

$$a_{ij} = \mathrm{AL}(\mathrm{IALDIAG}(i) - i + j) \text{ for } i > j$$

$$nau = 10 \text{ and } nal = 11$$

Note

The envelope size of the matrix A is $(nal + nau - n)$ as the diagonal elements are counted both in nal and nau. If all the elements in the row or column to be stored are zero, then the diagonal element having a value of zero is stored for that row or column.

13.4.2.2 Diagonal-out Storage Mode

The diagonal-out storage scheme stores the unsymmetric matrix A in (13–6) in four arrays — two real arrays AU and AL containing the columns of the upper triangular part and the rows of the lower triangular part of A respectively, and two integer arrays IAUDIAG and IALDIAG ontaining the pointers to the diagonal elements in the arrays AU and AL, respectively.

As the scheme is a diagonal-out scheme, the elements in a column of the upper triangular part are stored starting with the diagonal element and moving up the column to the first nonzero element in the column.

Similarly, the elements in a row of the lower triangular part are stored starting from the diagonal element and moving outward to the first nonzero element in the row. The data for each row and column are stored consecutively, the rows and columns are stored in order, and there is no space between successive rows and columns.

The diagonal of A is stored along with the upper triangular part in the array AU. The array AL also has storage for the diagonal elements, but these locations are not accessed and they can be used to store other information. Thus, the matrix A, in (13–6), is stored as follows:

$$\mathrm{AU} = (a_{11}, a_{22}, a_{33}, a_{23}, a_{44}, a_{34}, a_{55}, 0, 0, a_{25})$$

$$\mathrm{AL} = (*, *, a_{21}, *, *, 0, a_{42}, *, 0, 0, a_{52}))$$

$$\mathrm{IAUDIAG} = (1, 2, 3, 5, 7, 11)$$

and:

$$\mathrm{IALDIAG} = (1, 2, 4, 5, 8, 12)$$

The diagonal elements in AL are indicated by $*$ and the elements in the array IALDIAG are pointers to these locations. IAUDIAG and IALDIAG are of length at least $n + 1$, where n is the order of the matrix A.

The $n + 1$-st element is the pointer to the location, in AU and AL, of the diagonal entry of row $n + 1$, if there had been such a row. This allows the determination of the location of the first nonzero entry in the n-th row and column of A. AU and AL are of lengths at least nau and nal, respectively, where nau is the envelope size of the upper triangular part of A (including the diagonal) and nal is the envelope size of the lower triangular part of A (including the diagonal). Thus, for $a_{ij} \in Env(A)$, the following exists:

$$a_{jj} = \mathrm{AU}(\mathrm{IAUDIAG}(j)) \text{ for } 1 \leq j \leq n$$

$$a_{ij} = \mathrm{AU}(\mathrm{IAUDIAG}(j) + j - i) \text{ for } i < j$$

$$a_{ij} = \mathrm{AL}(\mathrm{IALDIAG}(i) + i - j) \text{ for } i > j$$

$$nau = 10 \text{ and } nal = 11$$

Note

The envelope size of the matrix A is $nal + nau - n$ as the diagonal elements are counted both in nal and nau. If all the elements in the row or column to be stored are zero, then the diagonal element having a value of zero is stored for that row or column.

13.4.2.3 Structurally Symmetric Profile-In Storage Mode

Unsymmetric matrices with structural symmetry can be stored using only one array for both the upper and lower triangular parts. The elements of the strict lower triangular part are stored first in a profile-in mode, followed by the elements of the strict upper triangular part, also in profile-in mode and finally the diagonal element.

As the matrix is structurally symmetric, it is possible to determine the start of each column, given a pointer to the diagonal entries. Thus the matrix A:

$$A = \begin{bmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & & a_{24} & a_{25} \\ & & a_{33} & 0 & 0 \\ & a_{42} & 0 & a_{44} & 0 \\ & a_{52} & 0 & 0 & a_{55} \end{bmatrix} \tag{13–7}$$

is stored using the real array AU and the integer array: IAUDIAG as

$$\mathrm{AU} = (a_{11}, a_{21}, a_{12}, a_{22}, a_{33}, a_{42}, 0, a_{24}, 0, a_{44}, a_{52}, 0, 0, a_{25}, 0, 0, a_{55})$$

and:

$$\mathrm{IAUDIAG} = (1, 4, 5, 10, 17)$$

IAUDIAG is of length at least n and AU is of length at least nau, where nau is the profile of the matrix A. The arrays AL and IALDIAG are not used. Thus, for $a_{ij} \in Env(A)$, the following exists:

$$a_{jj} = \mathrm{AU}(\mathrm{IAUDIAG}(j)) \text{ for } 1 \leq j \leq n$$

$$a_{ij} = \mathrm{AU}(\mathrm{IAUDIAG}(j) - j + i) \text{ for } i < j$$

$$a_{ij} = \mathrm{AU}\left(\frac{\mathrm{IAUDIAG}(i) + \mathrm{IAUDIAG}(i-1) + 1}{2} - i + j\right) \quad \text{for } i > j$$

and:

$$nau = 17$$

If all the elements in the row or column to be stored are zero, then the diagonal element having a value of zero is stored for that row or column.

13.5 DXML Skyline Solvers

DXML includes routines to solve systems of equations:

$$AX = B$$

or:

$$A^T X = B$$

where the matrix A is stored using the skyline storage scheme. A can be either a symmetric or an unsymmetric matrix. If A is a symmetric matrix, then it can be stored using either the profile-in or the diagonal-out storage mode. If A is unsymmetric, then it can be stored using either the profile-in storage mode, the diagonal-out storage mode, or the structurally symmetric, profile-in storage mode.

The skyline solvers in DXML include routines for both the LDU (or the $U^T DU$) factorization as well as the forward and back solves. As the routines for these two operations are separate, it allows the factorization to be evaluated once, followed by repeated use of the solve routine to obtain the solution for different right sides. The solve routines can also be called with multiple right sides. The factorize and solve routines must be called with the same storage scheme.

In addition to the LDU (or $U^T DU$) factorization and the triangular solve routines for the symmetric and unsymmetric matrices stored using the skyline storage scheme, DXML also provides the following functionality:

- **Evaluation of the determinant**

  The factorization routines include an option for the evaluation of the determinant of the matrix A. In order to prevent overflow or underflow, the determinant is returned in two parts, *det_base* and *det_pwr*, with the value of the determinant given as:

  $det\_base * 10^{det\_pwr}$

  where:

  $1.0 \leq det\_base < 10.0$

- **Evaluation of the inertia**

  The inertia of the symmetric matrix A is the triplet of integers $ipeigen, ineigen, izeigen$, consisting of the number of positive, negative and zero eigenvalues, respectively.

  The factorization routines allow the option of evaluating the inertia of the symmetric matrix A. In addition to the number of positive and negative eigenvalues, the routine returns an indication of the existence, or otherwise, of at least one zero eigenvalue.

- **Partial factorization**

  The factorization routines also allow a partial factorization of A, starting from row and column $ibeg+1$ where $ibeg > 0$. The factorization of rows and columns 1 through $ibeg$ is assumed to have been already obtained by a previous call to the routine.

- **Pivoting**

  No pivoting is done during the factorization process to ensure numerical stability. However, if a small pivot (in absolute value) is encountered, an option is provided to either stop the factorization, continue the factorization, or continue after setting the pivot equal to a predetermined value. The location of the first occurrence of a small pivot and its value are returned on exit from the factorization routine.

  Due to the lack of pivoting for numerical stability, caution is urged when using the skyline solvers for the solution of systems that are not positive (or negative) definite or diagonally dominant.

- **Statistics on the matrix**

  DXML provides the option to collect and print the statistics on the skyline matrix. By appropriately selecting the input parameters, the following information can be obtained from the factorization routines:

  - The envelope size of the matrix
  - The number of zeros in the envelope at the start of the factorization

- The percentage of the envelope that was sparse at the start of the factorization
- The maximum height of a column
- The average height of a column
- The root-mean-square height of a column
- The maximum width of a row (unsymmetric matrices only)
- The average width of a row (unsymmetric matrices only)
- The root-mean-square width of a row (unsymmetric matrices only)

These statistics also include the determinant of the matrix A, if evaluated, and in the case of symmetric matrices, the inertia of the matrix, if evaluated.

- **Evaluation of matrix norms**

DXML provides routines for the evaluation of the various norms of the matrix A as well as an estimate of the reciprocal of the condition number of A. By appropriately setting an input parameter, the following quantities can be evaluated:

- 1-norm of (A):

$$||A||_1 = \max_j \sum_i |a_{ij}|$$

- ∞-norm of (A):

$$||A||_\infty = \max_i \sum_j |a_{ij}|$$

- Frobenius-norm of (A):

$$||A||_F = \sqrt{\sum_i \sum_j |a_{ij}|^2}$$

- Largest absolute value of (A):

$$\max_{i,j} |a_{ij}|$$

Note

The 1-norm of A is the ∞-norm of A^T and the last quantity above is not a matrix norm.

- **Condition number estimator**

The condition number estimator included in DXML for matrices stored in the skyline storage format is based on the LAPACK routine DLACON [Anderson et.al. 1992], which estimates the 1-norm of a square, real matrix. The DXML routine returns the reciprocal of the condition number of A as:

$$rcond(A) = \frac{1}{||A|| \cdot ||A^{-1}||}$$

where either the 1-norm or the ∞-norm is used. The norm of A is evaluated by an appropriate call to the routine that evaluates the various norms, and the estimate of the norm of A^{-1} is evaluated using the routine DLACON.

- **Iterative refinement, error bounds, and backward error estimates**

  DXML provides routines to improve the computed solution via iterative refinement. This is done by obtaining the residual, r, corresponding to the calculated solution $\hat{x}$, and updating it to get a better solution vector x_{new} as follows:

  $$r = b - A\hat{x}$$

  $$\delta x = A^{-1}r$$

  and:

  $$x_{new} = \hat{x} + \delta x$$

  In the case of an unsymmetric matrix, if the system being solved is:

  $$A^T x = b$$

  the updated solution vector is obtained as:

  $$r = b - A^T\hat{x}$$

  $$\delta x = A^{-T}r$$

  and:

  $$x_{new} = \hat{x} + \delta x$$

  Therefore, both the original matrix A as well as the LDU (or $U^T DU$) factorization are required, in addition to the right hand side, b, and the calculated solution x. The iterative refinement routines evaluate all quantities in the same precision as the rest of the computation, that is, no extended precision is used [Skeel 1980].

  Additionally, the iterative refinement routines provide the component-wise relative backward error and the estimated forward error bound for each solution vector [Demmel et. al. 1988, Arioli, Demmel and Duff 1989, Anderson et. al. 1992]. These quantities can be used to provide an indication of the quality of the solution. The component-wise relative backward error, $berr$, of each solution vector is the smallest relative change in any entry of A or b that makes $\hat{x}$ an exact solution. The estimated forward error, $ferr$, bounds the magnitude of the largest entry in $\hat{x} - x_{true}$ divided by the magnitude of the largest entry in $\hat{x}$, where x_{true} is the true solution and $\hat{x}$, the calculated solution.

  The criterion for stopping iterative refinement is based on the discussion in [Arioli, Demmel, Duff 1989]. Iterations are continued as long as all of the following conditions are satisfied:

  - The number of iterations of the iterative refinement process is less than the maximum allowed.
  - $berr$ reduces by at least a factor of 2 during the previous iteration.
  - $berr$ is larger than the machine precision, ϵ (the greatest positive number such that the floating point representation of $1 + \epsilon$ equals 1).

- **Driver routines**

  DXML provides two types of driver routines for the solution of linear systems with the matrix stored in the skyline storage scheme:

  – A simple driver, which factorizes the matrix A and solves the system:

  $$AX = B$$

  or:

  $$A^T X = B$$

  with the solution x overwriting the right hand side and the factorization overwriting the matrix A.

  – An expert driver that can also perform condition number estimation, a check for singularity, iterative refinement of the solution and computation of the component-wise relative backward error and the estimated forward error bound for the solution vector.

  Both drivers are provided for symmetric and unsymmetric matrices and allow a choice of storage modes for the skyline matrix. The expert driver has higher memory requirements than the simple driver. It also allows the matrix to be input in either the factored or the unfactored form and provides more options in the factorization phase.

13.6 Naming Conventions for Direct Solver Subprograms

Table 13–1 shows the character groups and the character mnemonics and their meaning for each skyline solver routine name.

Table 13–1 Naming Conventions for Direct Solver Subprograms

Character Group	Mnemonic	Meaning
First group	D	Double-precision real data
Second group	S	Symmetric matrix
	U	Unsymmetric matrix
Third group	SKY	Matrix stored in skyline storage scheme
Fourth Group	N	Evaluate matrix norms
	F	Factorize
	S	Solve
	C	Estimate condition number
	R	Perform iterative refinement
	D	Simple driver
	X	Expert driver

Thus, the routine DUSKYF obtains the *LDU* factorization for an unsymmetric matrix consisting of real double-precision data stored in the skyline storage mode.

13.7 Summary of Skyline Solver Subprograms

Table 13–2 summarizes the skyline solver subprograms.

Table 13–2 Summary of Direct Solver Subprograms

Subprogram Name	Meaning
DSSKYN	Obtains, in double-precision arithmetic, the 1-norm, the ∞-norm, the Frobenius norm, or the maximum absolute value of a symmetric matrix stored in either the profile-in or the diagonal-out skyline storage mode.
DSSKYF	Obtains, in double-precision arithmetic, the U^TDU factorization of a symmetric matrix stored in either the profile-in or the diagonal-out skyline storage mode.
DSSKYS	Obtains, in double-precision arithmetic, the solution to the system $AX = B$, where A has been factored using the routine DSSKYF.
DSSKYC	Obtains, in double-precision arithmetic, the reciprocal of the estimate of the condition number of a symmetric matrix stored in either the profile-in or the diagonal-out skyline storage mode.
DSSKYR	Obtains, in double-precision arithmetic, an improvement to the solution via iterative refinement, the component-wise relative backward error and the estimated forward error bounds for the solution vector. The symmetric matrix is stored in either the profile-in or the diagonal-out skyline storage mode.
DSSKYD	Obtains, in double-precision arithmetic, the U^TDU factorization of the matrix A, followed by the solution of the system $AX = B$, where the symmetric matrix A is stored in either the profile-in or the diagonal-out skyline storage mode.
DSSKYX	Obtains, in double-precision arithmetic, the U^TDU factorization and the condition number estimate of the matrix A. If the matrix is non-singular, the solution of the system $AX = B$ is obtained, followed by iterative refinement and the calculation of the component-wise relative backward error and the estimated forward error bounds for the solution vector. The symmetric matrix A is stored in either the profile-in or the diagonal-out skyline storage mode.
DUSKYN	Obtains, in double-precision arithmetic, the 1-norm, the ∞-norm, the Frobenius norm or the maximum absolute value of an unsymmetric matrix stored in either the profile-in, the diagonal-out or the structurally symmetric profile-in skyline storage mode.
DUSKYF	Obtains, in double-precision arithmetic, the LDU factorization of an unsymmetric matrix stored in either the profile-in, the diagonal-out or the structurally symmetric profile-in skyline storage mode.
DUSKYS	Obtains, in double-precision arithmetic, the solution to the system $AX = B$ or $A^TX = B$, where A has been factored using the routine DUSKYF.
DUSKYC	Obtains, in double-precision arithmetic, the reciprocal of the estimate of the condition number of an unsymmetric matrix stored in either the profile-in, the diagonal-out or the structurally symmetric profile-in skyline storage mode. Either the 1-norm or the ∞-norm can be used.

(continued on next page)

Table 13–2 (Cont.) Summary of Direct Solver Subprograms

Subprogram Name	Meaning
DUSKYR	Obtains, in double-precision arithmetic, an improvement to the solution via iterative refinement, the component-wise relative backward error and the estimated forward error bounds for the solution vector. The unsymmetric matrix is stored in either the profile-in, the diagonal-out or the structurally symmetric profile-in skyline storage mode.
DUSKYD	Obtains, in double-precision arithmetic, the LDU factorization of the matrix A, followed by the solution of the system $AX = B$ or $A^TX = B$, where the unsymmetric matrix A is stored in either the profile-in, the diagonal-out or the structurally symmetric profile-in skyline storage mode.
DUSKYX	Obtains, in double-precision arithmetic, the LDU factorization and the condition number estimate of the matrix A. If the matrix is non-singular, the solution of the system $AX = B$ or $A^TX = B$ is obtained, followed by iterative refinement and the calculation of the component-wise relative backward error and the estimated forward error bounds for the solution vector. The unsymmetric matrix A is stored in either the profile-in, the diagonal-out or the structurally symmetric profile-in skyline storage mode.

13.8 Error Handling

Errors during the execution of one of the skyline solver routines are indicated by an appropriate value of the error flag, *ierror*. The routine sets the error flag, prints out any error message and returns control to the calling program. It is your responsibility to ensure that the routines completed successfully, as indicated by *ierror = 0*.

Table 13–3 provides a list of the error flags and their meaning.

Negative values indicate a fatal error such as invalid input data, while positive values indicate a warning, such as a small pivot during factorization.

The error flags in the -2000 range are the result of invalid input data and those in the -3000 range are caused by an error during computation such as a small pivot causing the factorization process to stop.

Table 13–3 Error Flags for Direct Solver Subprograms

Error Flag	Meaning
-2001	Value of n is invalid
-2002	Value of nau is invalid
-2003	Value of nal is invalid
-2004	Value of $niparam$ is invalid
-2005	Value of $nrparam$ is invalid
-2006	Value of $niwrk$ is invalid
-2007	Value of $nrwrk$ is invalid

(continued on next page)

Table 13–3 (Cont.) Error Flags for Direct Solver Subprograms

Error Flag	Meaning
-2008	Value of *iolevel* is invalid
-2009	Value of *idefault* is invalid
-2010	Value of *istore* is invalid
-2011	Value of *inorm* is invalid
-2012	Value of *ibeg* is invalid
-2013	Value of *idet* is invalid
-2014	Value of *ipvt* is invalid
-2015	Value of *inertia* is invalid
-2016	Value of *pvt_sml* is invalid
-2017	Value of *pvt_new* is invalid
-2018	Value of *ldb* or *ldbx* is invalid
-2019	Value of *nbx* is invalid
-2020	Value of *itrans* is invalid
-2021	Value of *anorm* is invalid
-2022	Value of *itmax* is invalid
-2023	Value of *ldx* is invalid
-2024	Value of *ifactor* is invalid
-3001	Small pivot encountered; factorization stopped
-3002	Matrix singular to working precision
3001	Small pivot encountered; factorization continued
3002	Small pivot encountered; factorization continued after pivot replacement
-4001	Memory allocation routine in the parallel version failed

To recover from errors in the -2000 range, the invalid argument should be set to an appropriate value and the routine called again. Errors in the -3000 range indicate that the solution procedure used might not be applicable to the problem under consideration. An alternative solution procedure is recommended.

The parallel version of the factorization routines require a small amount of additional memory for temporary variables. This memory allocation is not expected to fail under normal conditions. However, if an error flag with value -4001 is returned, you can either increase allocated values of pagefile quota and virtual memory, or reduce the number of processors used, or use the serial version of the routine.

The amount of information printed as a result of an error can be controlled by setting the variable *iolevel* to an appropriate value. By a suitable choice of *iounit*, all information printed can be suppressed.

13.9 Suggestions on the Use of the Skyline Solvers

The skyline solvers included in DXML provide routines for the factorization and the solution of matrices stored in the skyline storage format. Additional functionality includes the evaluation of matrix norms, condition number estimation, iterative refinement, component-wise backward error, and estimated forward error bounds.

These routines are provided for both symmetric and unsymmetric matrices. The symmetric matrix can be stored in either the profile-in or the diagonal-out storage mode. The unsymmetric matrix can be stored in either the profile-in, the diagonal-out, or the structurally symmetric profile-in storage modes.

The following steps are suggested in the use of the skyline solvers. Further details are provided in the description of each routine.

- Select a storage scheme for the matrix.

  Once the matrix is stored using a particular storage scheme, the same storage scheme must be used in all the skyline routines that operate on the matrix.

- Select the order in which the routines are called.

  The simple and expert driver routines provide most of the functionality that is included in the DXML's skyline solvers via a simple call to a single routine. If the functionality provided by one of these routines is what is required by your application, then the driver routines are recommended. If not, then you need to call the required routines in the appropriate order. It is also possible to mix the two approaches and follow a driver routine by one of the other skyline routines or vice-versa. For example, the expert driver routines do not allow partial factorization. If this functionality is required, then a call to the factorization routine can be followed by a call to the expert driver, with the fully factored matrix as input. Care must be taken to ensure that data that must not change between calls to successive skyline routines, remains unchanged.

  The factorization routine for the skyline matrices overwrites the original matrix A with the LDU (or $U^T DU$) factors. As a result, routines that require the original matrix as an input must be called prior to the factorization routine, or a copy of the matrix made before the call to the factorization routine. For example, the evaluation of the norms requires the original matrix A. Also the condition number estimator requires both the 1-norm or the ∞-norm of A as well as its LDU (or $U^T DU$) factors. In this case, the norm evaluation routine must be called first, followed by the factorization routine and finally the routine for the condition number estimation. This order of routines will allow the use of only one copy of the matrix.

  The iterative refinement routines require both the original matrix and the original right sides, as well as the factored matrix and the solution vectors. Since the solve routines overwrite the right hand sides, a copy of both the original matrix and the right sides must be saved before a call to the factor and solve routines, respectively.

- Set up the integer and the real parameter arrays.

  The arrays IPARAM and RPARAM are used to pass integer and real parameters, respectively, to the skyline routines. These arrays must be set up with the appropriate values prior to a call to each routine. There is an option to select default values for some of the parameters. If this option is chosen,

the remaining parameters must be assigned values before the call to a skyline routine.

Some parameters are presently unused, but included for future use. These can be dummy parameters.

- Set up arrays for integer and real workspace.

  Each skyline routine requires integer and real workspace for the computation. The workspace arrays, IWRK and RWRK must be of sufficient length as indicated in the description of each routine. The size of this workspace is provided via the variables *niwrk* and *nrwrk* in the array IPARAM. Some parts of the workspace contain information generated during the factorization process and used in other routines. This information must remain unchanged between the call to the factorization routine and any subsequent routines.

- Select the options in factorization.

  The factorization process allows additional functionality such as the calculation of the determinant, statistics on the matrix and so forth. These can be useful in getting a better understanding of the properties of the matrix under consideration. However, their use is also expensive, and hence these options should not be used unless the information generated is important for the problem being solved.

  The factorization routines allow partial factorization, with the factorization starting at row and column $(ibeg + 1)$ instead of 1. In this case, the first $ibeg$ rows and columns are assumed to have already been factored by a previous call to the factor routine.

  No pivoting is done during the factorization process. Hence care must be taken in cases where the matrix is not symmetric positive (negative) definite or diagonally dominant and therefore might require pivoting to ensure a stable factorization. Options are provided to stop the factorization process when a small pivot is encountered, or to continue the factorization process. In the latter case, there is an option to either use the same pivot element or use a replacement pivot element.

- Check the value of the PARALLEL environment variable.

  If you are using the parallelized version of the factorization routines, you must set the value of the PARALLEL environment variable, even if you are using a single processor. See Parallel Execution Environment Variable.

- Check the accuracy of the solution.

  An estimate of the quality of the solution can be obtained from the iterative refinement routine which calculates the component-wise backward error and the estimated forward error bounds.

- Check the error flags.

  The error flag *ierror* must be checked on exit from each call to a skyline routine, especially if all messages from the routine have been suppressed.

13.10 A Look at Some Skyline Solvers

To illustrate the use of the sparse skyline solvers, consider the linear system of equations derived from the discretization of the Laplace's equation:

$$-u_{xx} - u_{yy} = 0$$

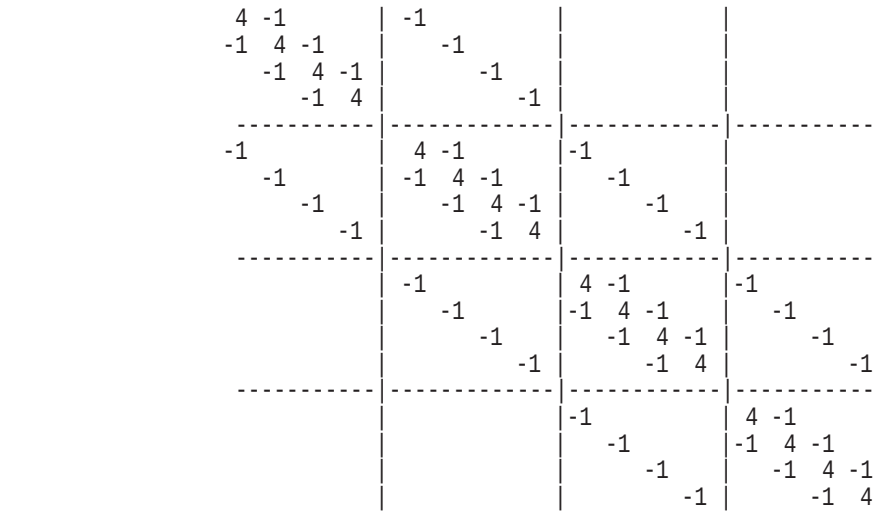
on the unit square:

$$0 \leq x \leq 1$$

and:

$$0 \leq y \leq 1$$

with Dirichlet boundary conditions, using the standard five point central differencing scheme. Assuming nx grid points in the x direction and ny grid points in the y direction, the coefficient matrix has the following form for $nx = ny = 4$:



The elements not defined in the matrix are zero. The exact solution is assumed to be all 1.0.

The examples in this section illustrate the use of the skyline solvers and the various options provided. Each example is self-contained, with comments indicating the operation being performed. The output files created by the example programs illustrate the output that can be obtained from the skyline solvers by setting the parameters appropriately.

These examples illustrate the use of the DXML skyline solver routines from Fortran, C, and C++ codes. Additional examples are included online in the `/usr/examples/dxml` directory.

Example 13–1 Skyline Solver with the Simple Driver Routine (Fortran Code)

```
      PROGRAM EXAMPLE_SKYSOL
C
C ***** TO DEMONSTRATE THE USE OF THE SPARSE SKYLINE SOLVER.
C
C ***** THIS PROGRAM ILLUSTRATES THE FOLLOWING:
C
C       (1) USE OF THE SIMPLE DRIVER ROUTINE TO SOLVE THE TEST
C           PROBLEM STORED AS A SYMMETRIC MATRIX IN THE PROFILE-IN
C           STORAGE MODE.
C
      IMPLICIT REAL*8 (A-H, O-Z)
C
      PARAMETER (NMAX = 100)
      PARAMETER (NMAX_SKY = 1100)
C
      REAL*8 AU(NMAX_SKY), XO(NMAX), BX(NMAX), RPARAM(100),
     $       DUM, TEMP
C
      INTEGER IAUDIAG(NMAX), IPARAM(100), IWRK(2*NMAX),
     $        I,NX, NY, NXNY, IDUM
C
C ***** SETUP OUTPUT FILE
C
      IOUNIT = 7
      OPEN (UNIT=IOUNIT, FILE='OUTPUT.DATA', STATUS='UNKNOWN')
      REWIND IOUNIT
C
      WRITE (IOUNIT, 101)
C
C ***** SET UP THE PROBLEM SIZE
C
      NX = 10
      NY = 10
      NXNY = NX*NY
      WRITE (IOUNIT, 102) NXNY
C
C ***** GENERATE THE MATRIX IN THE FIVE DIAGONAL FORM.
C
      CALL GENMAT(NX, NY, NXNY)
C
C ***** GENERATE XO, THE TRUE SOLUTION
C
      DO I = 1, NXNY
          XO(I) = 1.0D0
      END DO
C
C ***** OBTAIN THE CORRESPONDING RIGHT HAND SIDE (IN BX). THIS IS
C       OVERWRITTEN BY THE SOLUTION.
C
      CALL MATVEC (NX, NY, NXNY, XO, BX)
C
C ***** CONVERT THE MATRIX INTO SYMMETRIC PROFILE-IN SKYLINE STORAGE
C       MODE
C
      CALL CONVERT_TO_SKYLINE (NX, NY, NXNY, AU, IAUDIAG, NAU)
C
```

(continued on next page)

Example 13–1 (Cont.) Skyline Solver with the Simple Driver Routine (Fortran Code)

```
C ***** SET THE PARAMETERS (INTEGER AND REAL) FOR THE SIMPLE DRIVER
C
C       IWRK     = IPARAM(3)  = 2*NXNY (INTEGER WORKSPACE)
C       RWRK     = IPARAM(4)  = 0 (NO REAL WORKSPACE NEEDED)
C       IOLEVEL  = IPARAM(6)  = 2 (FOR DETAILED INFORMATION AND
C                               STATISTICS)
C       IDEFAULT = IPARAM(7)  = 1 (USER ASSIGNED VALUES)
C       ISTORE   = IPARAM(8)  = 1 (FOR PROFILE-IN STORAGE MODE)
C       IPVT     = IPARAM(9)  = 0 (STOP IF ABS(PIVOT) IS SMALLER THAN
C                               PVT_SML)
C       PVT_SML  = RPARAM(1)  = 1.0D-12 (STOP IF ABS(PIVOT) IS SMALLER
C                                THAN PVT_SML)
C
      IPARAM(1) = 100
      IPARAM(2) = 100
      IPARAM(3) = 2*NXNY
      IPARAM(4) = 0
      IPARAM(5) = IOUNIT
      IPARAM(6) = 2
      IPARAM(7) = 1
      IPARAM(8) = 1
      IPARAM(9) = 0
C
      RPARAM(1) = 1.0D-12
C
C ***** CALL THE SIMPLE DRIVER ROUTINE FOR FACTORIZATION AND SOLUTION,
C       WITH A SINGLE RIGHT HAND SIDE. RWRK IS A DUMMY ARGUMENT.
C
      LDBX = NMAX
      NBX = 1
      CALL DSSKYD ( NXNY, AU, IAUDIAG, NAU, BX, LDBX, NBX,
     $              IPARAM, RPARAM, IWRK, DUM, IERROR )
C
C ***** CHECK THAT THE SOLUTION COMPLETED WITHOUT ERROR
C
      IF (IERROR.NE.0) THEN
          WRITE (IOUNIT, 103) IERROR
          GO TO 999
      END IF
C
C ***** FIND MAX ERROR IN SOLUTION
C
      TEMP = ABS(XO(1) - BX(1))
      DO I = 2, NXNY
         TEMP = MAX( TEMP, ABS(XO(I) - BX(I)) )
      END DO
      WRITE (IOUNIT, 104) TEMP
C
  999 CONTINUE
C
  101 FORMAT (/,2X, 'SOLVING EXAMPLE PROBLEM WITH SYMMETRIC ',
     $              'PROFILE-IN SKYLINE STORAGE MODE',/)
  102 FORMAT (/,2X, 'ORDER OF THE MATRIX:',I5)
  103 FORMAT (/,2X, 'ERROR IN THE SIMPLE DRIVER ROUTINE:',I10)
  104 FORMAT (/,2X, 'MAXIMUM ERROR IN SOLUTION: ', E15.8,/)
C
      STOP
      END
```

(continued on next page)

Example 13–1 (Cont.) Skyline Solver with the Simple Driver Routine (Fortran Code)

```
C
C
C
      SUBROUTINE GENMAT (NX, NY, NXNY)
C
C ***** ROUTINE TO GENERATE THE MATRIX FOR THE EXAMPLE (IN THE FIVE
C       DIAGONAL FORM). A1 AND A2 ARE THE SUBDIAGONALS, A3 IS THE MAIN
C       DIAGONAL AND A4 AND A5 ARE THE SUPERDIAGONALS.
C
      IMPLICIT REAL*8 (A-H,O-Z)
C
      PARAMETER (NMAX = 100)
C
      COMMON /MATRIX/ A1(NMAX), A2(NMAX), A3(NMAX), A4(NMAX), A5(NMAX)
C
      DO I = 1, NXNY
          A3(I) = 4.0D0
          A1(I) = 0.0D0
          A2(I) = 0.0D0
          A4(I) = 0.0D0
          A5(I) = 0.0D0
      END DO
C
      DO J = 1, NY
          DO I = 2, NX
              K = (J-1)*NX+I
              A2(K) = -1.0D0
          END DO
          DO I = 1, NX-1
              K = (J-1)*NX+I
              A4(K) = -1.0D0
          END DO
      END DO
C
      DO J = 2, NY
          DO I = 1, NX
              K = (J-1)*NX+I
              A1(K) = -1.0D0
          END DO
      END DO
C
      DO J = 1, NY-1
          DO I = 1, NX
              K = (J-1)*NX+I
              A5(K) = -1.0D0
          END DO
      END DO
C
      RETURN
      END
C
C
C
      SUBROUTINE MATVEC (NX, NY, NXNY, TMP1, TMP2)
C
C ***** ROUTINE TO OBTAIN THE MATRIX VECTOR MULTIPLY, USING THE MATRIX
C       FROM THE FIVE-DIAGONAL FORM. TMP1 IS THE INPUT VECTOR; TMP2 IS
C       THE OUTPUT VECTOR.
C
```

(continued on next page)

Example 13–1 (Cont.) Skyline Solver with the Simple Driver Routine (Fortran Code)

```
      IMPLICIT REAL*8 (A-H,O-Z)
C
      PARAMETER (NMAX = 100)
C
      REAL*8 TMP1(*), TMP2(*)
C
      INTEGER NX,NY,NXNY
C
      COMMON /MATRIX/ A1(NMAX), A2(NMAX), A3(NMAX), A4(NMAX), A5(NMAX)
C
C
      DO I = 1, NXNY
          TMP2(I) = A3(I)*TMP1(I)
      END DO
C
      DO I = 1, NXNY-1
          TMP2(I) = TMP2(I) + A4(I)*TMP1(I+1)
      END DO
C
      DO I = 2, NXNY
          TMP2(I) = TMP2(I) + A2(I)*TMP1(I-1)
      END DO
C
      DO I = 1, NXNY-NX
          TMP2(I) = TMP2(I) + A5(I)*TMP1(I+NX)
      END DO
C
      DO I = NX+1, NXNY
          TMP2(I) = TMP2(I) + A1(I)*TMP1(I-NX)
      END DO
C
      RETURN
      END
C
C
C
      SUBROUTINE CONVERT_TO_SKYLINE (NX, NY, NXNY, AU, IAUDIAG, NAU)
C
C ***** ROUTINE FOR CONVERTING THE MATRIX FROM THE FIVE-DIAGONAL FORM
C       TO THE SYMMETRIC PROFILE-IN SKYLINE FORM. ONLY THE UPPER
C       TRIANGULAR PART IS STORED. THE MATRIX IS RETURNED IN THE REAL
C       AND INTEGER ARRAYS, AU AND IAUDIAG, RESPECTIVELY.
C
      IMPLICIT REAL*8 (A-H,O-Z)
C
      PARAMETER (NMAX = 100)
C
      REAL*8 AU(*)
C
      INTEGER IAUDIAG(*), NAU, NX, NY, NXNY
C
      COMMON /MATRIX/ A1(NMAX), A2(NMAX), A3(NMAX), A4(NMAX), A5(NMAX)
C
C
      INDEX = 1
      AU(INDEX) = A3(1)
      IAUDIAG(1) = 1
      INDEX = INDEX + 1
```

(continued on next page)

Example 13–1 (Cont.) Skyline Solver with the Simple Driver Routine (Fortran Code)

```
C
      DO I = 2, NX
          AU(INDEX) = A4(I-1)
          AU(INDEX+1) = A3(I)
          IAUDIAG(I) = INDEX+1
          INDEX= INDEX+2
      END DO
C
      DO I = NX+1, NXNY
          AU(INDEX) = A5(I-NX)
          INDEX = INDEX+1
          DO J = 1, NX-2
              AU(INDEX) = 0.0D0
              INDEX = INDEX+1
          END DO
          AU(INDEX) = A4(I-1)
          INDEX = INDEX+1
          AU(INDEX) = A3(I)
          IAUDIAG(I) = INDEX
          INDEX = INDEX+1
      END DO
C
      NAU = INDEX - 1

      RETURN
      END
```

(continued on next page)

Example 13–1 (Cont.) Skyline Solver with the Simple Driver Routine (Fortran Code)

Output from Example 1

```
 SOLVING EXAMPLE PROBLEM WITH SYMMETRIC PROFILE-IN SKYLINE STORAGE
 MODE ORDER OF THE MATRIX:  100
 ---------------------------------------------------------------
 simple driver for a symmetric matrix
 storage scheme: symmetric profile-in skyline
 order of matrix:        100
 ---------------------------------------------------------------
 u^ t-d-u factorization on a symmetric matrix
 storage scheme: symmetric profile-in skyline
 order of matrix:        100
 factorization starts at row/column:           1
 size of the envelope (upper triangle):        1009
 number of initial zeros in the envelope:      729
 percentage sparsity of the envelope:  72.25%
 maximum column height (including diagonal):   11
 average column height (including diagonal):   10.09
 root-mean-square column height (including diagonal): 10.45
 u^ t-d-u factorization completed without error
---------------------------------------------------------------
 u^ t-d-u solve for a symmetric matrix
 storage scheme: symmetric profile-in skyline
 order of matrix:        100
 number of right hand sides:         1
 u^ t-d-u solve completed without error
 simple driver completed without error
 MAXIMUM ERROR IN SOLUTION:  0.66613381E-15
```

Example 13–2 Skyline Solver with Iterative Refinement (Fortran Code)

```
      PROGRAM EXAMPLE_SKYSOL

C
C ***** TO DEMONSTRATE THE USE OF THE SPARSE SKYLINE SOLVER.
C
C ***** THIS PROGRAM ILLUSTRATES THE FOLLOWING:
C
C       (1) USE OF THE FACTOR AND SOLVE ROUTINES TO SOLVE THE TEST
C           PROBLEM WITH THE MATRIX STORED AS AN UNSYMMETRIC MATRIX
C           IN THE STRUCTURALLY SYMMETRIC PROFILE-IN STORAGE MODE.
C       (2) USE OF THE ITERATIVE REFINEMENT ROUTINES TO IMPROVE
C           THE SOLUTION AND OBTAIN THE ERROR BOUNDS.
C
      IMPLICIT REAL*8 (A-H, O-Z)
C
      PARAMETER (NMAX = 100)
      PARAMETER (NMAX_SKY = 2000)
C
      REAL*8 AU(NMAX_SKY), AU_ORIG(NMAX_SKY), XO(NMAX), BX(NMAX),
     $       BX_ORIG(NMAX), RWRK(3*NMAX), RPARAM(100), FERR(1),
     $       BERR(1), DUM, TEMP, ANORM
C
      INTEGER IAUDIAG(NMAX), IPARAM(100), IWRK(5*NMAX),
     $        NX, NY, NXNY, NAU, NAL, IDUM
C
C ***** SETUP OUTPUT FILE
C
      IOUNIT = 7
      OPEN (UNIT=IOUNIT, FILE='OUTPUT.DATA', STATUS='UNKNOWN')
      REWIND IOUNIT
C
      WRITE (IOUNIT, 101)
C
C ***** SET UP THE PROBLEM SIZE
C
      NX = 10
      NY = 10
      NXNY = NX*NY
      WRITE (IOUNIT, 102) NXNY
C
C ***** GENERATE THE MATRIX IN THE FIVE DIAGONAL FORM.
C
      CALL GENMAT(NX, NY, NXNY)
C
C ***** GENERATE XO, THE TRUE SOLUTION
C
      DO I = 1, NXNY
          XO(I) = 1.0D0
      END DO
C
C ***** OBTAIN THE CORRESPONDING RIGHT HAND SIDE (IN BX). THIS IS
C       OVERWRITTEN BY THE SOLUTION.
C
      CALL MATVEC (NX, NY, NXNY, XO, BX)
C
C ***** CONVERT THE MATRIX INTO STRUCTURALLY SYMMETRIC PROFILE-IN
C       SKYLINE STORAGE MODE.
C
      CALL CONVERT_TO_SKYLINE (NX, NY, NXNY, AU, IAUDIAG, NAU)
```

(continued on next page)

Example 13–2 (Cont.) Skyline Solver with Iterative Refinement (Fortran Code)

```
C
C ***** COPY THE MATRIX AND THE RIGHT HAND SIDE FOR USE IN ITERATIVE
C       REFINEMENT.
C
      DO I = 1, NAU
          AU_ORIG(I) = AU(I)
      END DO
C
      DO I = 1, NMAX
          BX_ORIG(I) = BX(I)
      END DO
C
C ***** SET THE PARAMETERS (INTEGER AND REAL) FOR THE FACTORIZATION
C
C       IWRK     = IPARAM(3)  = 4*NXNY (INTEGER WORKSPACE)
C       RWRK     = IPARAM(4)  = 0 (NO REAL WORKSPACE NEEDED)
C       IOLEVEL  = IPARAM(6)  = 1 (FOR MINIMAL INFORMATION)
C       IDEFAULT = IPARAM(7)  = 1 (USER ASSIGNED VALUES)
C       ISTORE   = IPARAM(8)  = 3 (FOR STRUCTURALLY SYMMETRIC
C                                  PROFILE-IN STORAGE MODE)
C       IBEG     = IPARAM(9)  = 0 (FOR FULL FACTORIZATION)
C       IDET     = IPARAM(10) = 0 (NO EVALUATION OF THE DETERMINANT)
C       IPVT     = IPARAM(11) = 0 (STOP IF ABS(PIVOT) IS SMALLER THAN
C                                  PVT_SML)
C       PVT_SML  = RPARAM(1)  = 1.0D-12 (STOP IF ABS(PIVOT) IS SMALLER
C                                        THAN PVT_SML).
C
      IPARAM(1) = 100
      IPARAM(2) = 100
      IPARAM(3) = 4*NXNY
      IPARAM(4) = 0
      IPARAM(5) = IOUNIT
      IPARAM(6) = 1
      IPARAM(7) = 1
      IPARAM(8) = 3
      IPARAM(9) = 0
      IPARAM(10) = 0
      IPARAM(11) = 0
C
      RPARAM(1) = 1.0D-12
C
C ***** CALL THE ROUTINE FOR FACTORIZATION (AL, ALDIAG, NAL, RWRK ARE
C       DUMMY ARGUMENTS).
C
      CALL DUSKYF ( NXNY, AU, IAUDIAG, NAU,
     $              DUM, IDUM, IDUM,
     $              IPARAM, RPARAM, IWRK, DUM, IERROR )
C
C ***** CHECK THAT FACTORIZATION COMPLETED WITHOUT ERROR.
C
      IF (IERROR.NE.0) THEN
          WRITE (IOUNIT, 103) IERROR
          GO TO 999
      END IF
```

(continued on next page)

Example 13–2 (Cont.) Skyline Solver with Iterative Refinement (Fortran Code)

```
C
C ***** SET THE PARAMETERS (INTEGER) FOR THE SOLUTION (NO REAL
C       PARAMETERS USED AT PRESENT).
C
C       IWRK     = IPARAM(3)  = 4*NXNY (INTEGER WORKSPACE)
C       RWRK     = IPARAM(4)  = 0 (NO REAL WORKSPACE NEEDED)
C       IOLEVEL  = IPARAM(6)  = 2 (FOR DETAILED INFORMATION)
C       IDEFAULT = IPARAM(7)  = 1 (USER ASSIGNED VALUES)
C       ISTORE   = IPARAM(8)  = 3 (FOR STRUCTURALLY SYMMETRIC
C                                  PROFILE-IN STORAGE MODE)
C       ITRANS   = IPARAM(9)  = 0 (TO SOLVE A * X = B)
C
      IPARAM(1) = 100
      IPARAM(2) = 100
      IPARAM(3) = 4*NXNY
      IPARAM(4) = 0
      IPARAM(5) = IOUNIT
      IPARAM(6) = 2
      IPARAM(7) = 1
      IPARAM(8) = 3
      IPARAM(9) = 0
C
C ***** CALL THE ROUTINE FOR SOLUTION (SINGLE RIGHT HAND SIDE). AL,
C       IALDIAG, NAL, RPARAM AND RWRK ARE DUMMY ARGUMENTS.
C
      LDBX = NMAX
      NBX = 1
      CALL DUSKYS ( NXNY, AU, IAUDIAG, NAU,
     $              DUM, IDUM, IDUM,
     $              BX, LDBX, NBX,
     $              IPARAM, DUM, IWRK, DUM, IERROR )
C
C ***** CHECK THAT THE SOLUTION COMPLETED WITHOUT ERROR
C
      IF (IERROR.NE.0) THEN
          WRITE (IOUNIT, 104) IERROR
          GO TO 999
      END IF
C
C ***** FIND MAX ERROR IN THE SOLUTION BEFORE REFINEMENT.
C
      TEMP = ABS(XO(1) - BX(1))
      DO I = 2, NXNY
         TEMP = MAX( TEMP, ABS(XO(I) - BX(I)) )
      END DO
      WRITE (IOUNIT, 105) TEMP
C
C ***** SET THE PARAMETERS (INTEGER) FOR ITERATIVE REFINEMENT. NO
C       REAL PARAMETERS USED AT PRESENT.
C
C       IWRK     = IPARAM(3)  = 5*NXNY (INTEGER WORKSPACE)
C       RWRK     = IPARAM(4)  = 3*NXNY (REAL WORKSPACE)
C       IOLEVEL  = IPARAM(6)  = 2 (FOR DETAILED INFORMATION)
C       IDEFAULT = IPARAM(7)  = 1 (USER ASSIGNED VALUES)
C       ISTORE   = IPARAM(8)  = 3 (FOR STRUCTURALLY SYMMETRIC
C                                  PROFILE-IN STORAGE MODE)
C       ITRANS    = IPARAM(9)  = 1 (SOLVING A * X = B)
C       ITMAX    = IPARAM(10) = 5 (MAXIMUM NUMBER OF ITERATIONS)
```

(continued on next page)

Example 13–2 (Cont.) Skyline Solver with Iterative Refinement (Fortran Code)

```
C
      IPARAM(1) = 100
      IPARAM(2) = 100
      IPARAM(3) = 5*NXNY
      IPARAM(4) = 3*NXNY
      IPARAM(5) = IOUNIT
      IPARAM(6) = 2
      IPARAM(7) = 1
      IPARAM(8) = 3
      IPARAM(9) = 1
      IPARAM(10) = 5
C
C ***** CALL THE ROUTINE FOR ITERATIVE REFINEMENT. AL_ORIG, AL,
C       IALDIAG, NAL AND RPARAM ARE DUMMY ARGUMENTS.
C
      CALL DUSKYR ( NXNY, AU_ORIG, AU, IAUDIAG, NAU,
     $              DUM, DUM, IDUM, IDUM,
     $              BX_ORIG, LDBX, BX, LDBX, FERR, BERR, NBX,
     $              IPARAM, DUM, IWRK, RWRK, IERROR )
C
C ***** CHECK THAT THE REFINEMENT COMPLETED WITHOUT ERROR.
C
      IF (IERROR.NE.0) THEN
          WRITE (IOUNIT, 106) IERROR
          GO TO 999
      END IF
C
C ***** FIND MAX ERROR IN THE SOLUTION AFTER REFINEMENT
C
      TEMP = ABS(XO(1) - BX(1))
      DO I = 2, NXNY
         TEMP = MAX( TEMP, ABS(XO(I) - BX(I)) )
      END DO
      WRITE (IOUNIT, 107) TEMP
C
  999 CONTINUE
C
  101 FORMAT (/,2X, 'SOLVING EXAMPLE PROBLEM WITH STRUCTURALLY ',
     $              'SYMMETRIC PROFILE-IN SKYLINE',
     $        /,2x, ' STORAGE MODE',/)
  102 FORMAT (/,2X, 'ORDER OF THE MATRIX:',I5)
  103 FORMAT (/,2X, 'ERROR IN THE FACTORIZATION ROUTINE:',I10)
  104 FORMAT (/,2X, 'ERROR IN THE SOLUTION ROUTINE:',I10)
  105 FORMAT (/,2X, 'MAXIMUM ERROR IN SOLUTION BEFORE REFINEMENT: ',
     $              E15.8,/)
  106 FORMAT (/,2X, 'ERROR IN THE ITERATIVE REFINEMENT ROUTINE:',I10)
  107 FORMAT (/,2X, 'MAXIMUM ERROR IN SOLUTION AFTER REFINEMENT: ',
     $              E15.8,/)
C
      STOP
      END
```

(continued on next page)

Example 13–2 (Cont.) Skyline Solver with Iterative Refinement (Fortran Code)

```
C
C
C
      SUBROUTINE GENMAT (NX, NY, NXNY)
C
C ***** ROUTINE TO GENERATE THE MATRIX FOR THE EXAMPLE (IN THE FIVE
C       DIAGONAL FORM). A1 AND A2 ARE THE SUBDIAGONALS, A3 IS THE
C       MAIN DIAGONAL, A4 AND A5 ARE THE SUPERDIAGONALS.
C
      IMPLICIT REAL*8 (A-H,O-Z)
C
      PARAMETER (NMAX = 100)
C
      COMMON /MATRIX/ A1(NMAX), A2(NMAX), A3(NMAX), A4(NMAX), A5(NMAX)
C
      DO I = 1, NXNY
          A3(I) = 4.0D0
          A1(I) = 0.0D0
          A2(I) = 0.0D0
          A4(I) = 0.0D0
          A5(I) = 0.0D0
      END DO
C
      DO J = 1, NY
          DO I = 2, NX
              K = (J-1)*NX+I
              A2(K) = -1.0D0
          END DO
          DO I = 1, NX-1
              K = (J-1)*NX+I
              A4(K) = -1.0D0
          END DO
      END DO
C
      DO J = 2, NY
          DO I = 1, NX
              K = (J-1)*NX+I
              A1(K) = -1.0D0
          END DO
      END DO
C
      DO J = 1, NY-1
          DO I = 1, NX
              K = (J-1)*NX+I
              A5(K) = -1.0D0
          END DO
      END DO
C
      RETURN
      END
```

(continued on next page)

Example 13–2 (Cont.) Skyline Solver with Iterative Refinement (Fortran Code)

```
C
C
C
      SUBROUTINE MATVEC (NX, NY, NXNY, TMP1, TMP2)
C
C ***** ROUTINE TO OBTAIN THE MATRIX VECTOR MULTIPLY, USING THE MATRIX
C       FROM THE FIVE-DIAGONAL FORM. TMP1 IS THE INPUT VECTOR; TMP2 IS
C       THE OUTPUT VECTOR.
C
      IMPLICIT REAL*8 (A-H,O-Z)
C
      PARAMETER (NMAX = 100)
C
      REAL*8 TMP1(*), TMP2(*)
C
      INTEGER NX,NY,NXNY
C
      COMMON /MATRIX/ A1(NMAX), A2(NMAX), A3(NMAX), A4(NMAX), A5(NMAX)
C
C
      DO I = 1, NXNY
          TMP2(I) = A3(I)*TMP1(I)
      END DO
C
      DO I = 1, NXNY-1
          TMP2(I) = TMP2(I) + A4(I)*TMP1(I+1)
      END DO
C
      DO I = 2, NXNY
          TMP2(I) = TMP2(I) + A2(I)*TMP1(I-1)
      END DO
C
      DO I = 1, NXNY-NX
          TMP2(I) = TMP2(I) + A5(I)*TMP1(I+NX)
      END DO
C
      DO I = NX+1, NXNY
          TMP2(I) = TMP2(I) + A1(I)*TMP1(I-NX)
      END DO
C
      RETURN
      END
C
C
C
      SUBROUTINE CONVERT_TO_SKYLINE (NX, NY, NXNY, AU, IAUDIAG, NAU)
C
C ***** ROUTINE FOR CONVERTING THE MATRIX FROM THE FIVE-DIAGONAL FORM
C       TO THE STRUCTURALLY SYMMETRIC PROFILE-IN SKYLINE FORM. THE
C       MATRIX IS RETURNED IN REAL AND INTEGER ARRAYS, AU AND IAUDIAG,
C       RESPECTIVELY.
```

(continued on next page)

Example 13–2 (Cont.) Skyline Solver with Iterative Refinement (Fortran Code)

```
C
      IMPLICIT REAL*8 (A-H,O-Z)
C
      PARAMETER (NMAX = 100)
C
      REAL*8 AU(*)
C
      INTEGER IAUDIAG(*), NAU, NX, NY, NXNY
C
      COMMON /MATRIX/ A1(NMAX), A2(NMAX), A3(NMAX), A4(NMAX), A5(NMAX)
C
C
      INDEX = 1
      AU(INDEX) = A3(1)
      IAUDIAG(1) = 1
      INDEX = INDEX + 1
C
      DO I = 2, NX
          AU(INDEX) = A2(I)
          AU(INDEX+1) = A4(I-1)
          AU(INDEX+2) = A3(I)
          IAUDIAG(I) = INDEX+2
          INDEX= INDEX+3
      END DO
C
      DO I = NX+1, NXNY
          AU(INDEX) = A1(I)
          INDEX = INDEX+1
          DO J = 1, NX-2
              AU(INDEX) = 0.0D0
              INDEX = INDEX+1
          END DO
          AU(INDEX) = A2(I)
          INDEX = INDEX+1
          AU(INDEX) = A5(I-NX)
          INDEX = INDEX+1
          DO J = 1, NX-2
              AU(INDEX) = 0.0D0
              INDEX = INDEX+1
          END DO
          AU(INDEX) = A4(I-1)
          INDEX = INDEX+1
          AU(INDEX) = A3(I)
          IAUDIAG(I) = INDEX
          INDEX = INDEX+1
      END DO
C
      NAU = INDEX - 1
C
      RETURN
C
      END
```

(continued on next page)

Example 13–2 (Cont.) Skyline Solver with Iterative Refinement (Fortran Code)

Output from Example 2

```
 SOLVING EXAMPLE PROBLEM WITH STRUCTURALLY SYMMETRIC PROFILE-IN
 SKYLINE STORAGE MODE ORDER OF THE MATRIX:  100
 ---------------------------------------------------------------------
  l-d-u factorization on an unsymmetric matrix
  storage scheme: structurally symmetric profile-in skyline
  order of matrix:        100
  partial factorization starts at row/column:         1
  l-d-u factorization completed without error
 ---------------------------------------------------------------------
  l-d-u solve for a unsymmetric matrix
  storage scheme: structurally symmetric profile-in skyline
  order of matrix:        100
  number of right hand sides:         1
  solving the system a * x = b
  l-d-u solve completed without error
 MAXIMUM ERROR IN SOLUTION BEFORE REFINEMENT:  0.66613381E-15
---------------------------------------------------------------------
  iterative refinement using an unsymmetric matrix
  storage scheme: structurally symmetric profile-in skyline
  order of matrix:        100
  number of right hand sides:         1
 for the right hand side:     1
     number of iterations of iterative refinement:    1
     componentwise relative backward error: 0.97144515E-16
     estimated forward error bound: 0.17274574E-12
  iterative refinement completed without error
 MAXIMUM ERROR IN SOLUTION AFTER REFINEMENT:  0.44408921E-15
```

Example 13–3 Skyline Solver Using Factorize and Solve Routines (C Code)

```
/*
 ******************************************************************************
 *                                                                            *
 *  Copyright Digital Equipment Corporation 1993 - 1996. All rights reserved.*
 *                                                                            *
 *  Restricted Rights: Use, duplication, or disclosure by the U.S.            *
 *  Government is subject to restrictions as set forth in subparagraph        *
 *  (c) (1) (ii) of DFARS 252.227-7013, or in FAR 52.227-19, or in FAR        *
 *  52.227-14 Alt. III, as applicable.                                        *
 *                                                                            *
 *  This software is proprietary to and embodies the confidential             *
 *  technology of Digital Equipment Corporation. Possession, use, or          *
 *  copying of this software and media is authorized only pursuant to a       *
 *  valid written license from Digital or an authorized sublicensor.          *
 *                                                                            *
 ******************************************************************************
*/
/*
    This is an example program to illustrate the use of the skyline
    solver routines dsskyf and dsskys from a C application program. The
    program generates the matrix, converts it into a profile-in symmetric
    skyline matrix, factorizes the matrix using dsskyf and solves the
    system using the routine dsskys. The right hand side of the problem
    is generated assuming a known solution. The maximum absolute error
    in the solution is printed out. The problem used is identical to the
    one in the example section of the chapter on skyline solvers in the DXML
    Reference Guide.

    This program illustrates the following:
         - routine naming convention for Digital Unix and VMS
         - Differences in array indexing between Fortran and C:
               C default x[n]: 0 to (n-1)
               Fortran default  x(n): 1 to n
         - implications for storing the index vector for the
           skyline storage scheme.

    For more detailed explanation of the routines used, please
    check the Reference Manual.

    Note: the code used in this example works on both Digital Unix and
    VMS. Conditional compilation is used to select the statements appropriate
    to each operating system.

    All output is directed to the screen.

*/

#include <stdio.h>
#include <stdlib.h>

/*
   Add trailing underscores to Fortran routines on Digital Unix.
*/

#if !defined(vms) && !defined(__vms)
#define dsskyf dsskyf_
#define dsskys dsskys_
#endif

#define ABS(x) (((x) < 0) ? -(x) : (x))
#define MAX(x,y) (((x) < (y)) ? (y) : (x))
```

(continued on next page)

Example 13–3 (Cont.) Skyline Solver Using Factorize and Solve Routines (C Code)

```
extern void genmat();
extern void matvec();
extern void convert_to_skyline1();

extern void dsskyf();
extern void dsskys();

int main()
{
    double *xo;
    double *bx;
    double *au;

    double *av1;
    double *av2;
    double *av3;
    double *av4;
    double *av5;

    double rparam[100];

    double dum, temp, max1, tmp1;

    int *iaudiag;
    int *iwrk;

    int iparam[100];

    int iounit,
        ierror,
        nx, ny, nxny,
        i, nau,
        nmax_sky, nmax_sky_au,
        ldbx, nbx;

/*
    define the size of the problem
*/

    nx = 10;
    ny = 10;
    nxny = nx * ny;
    nmax_sky_au = 1200;

/*
    obtain the memory for the 1-dimensional arrays
*/

    au = (double *)malloc(nmax_sky_au*sizeof(double));
    if (au == 0) perror("malloc");

    xo = (double *)malloc(nxny*sizeof(double));
    if (xo == 0) perror("malloc");

    bx = (double *)malloc(nxny*sizeof(double));
    if (bx == 0) perror("malloc");

    av1 = (double *)malloc(nxny*sizeof(double));
    if (av1 == 0) perror("malloc");

    av2 = (double *)malloc(nxny*sizeof(double));
    if (av2 == 0) perror("malloc");

    av3 = (double *)malloc(nxny*sizeof(double));
    if (av3 == 0) perror("malloc");
```

(continued on next page)

Example 13–3 (Cont.) Skyline Solver Using Factorize and Solve Routines (C Code)

```
    av4 = (double *)malloc(nxny*sizeof(double));
    if (av4 == 0) perror("malloc");

    av5 = (double *)malloc(nxny*sizeof(double));
    if (av5 == 0) perror("malloc");

    iaudiag = (int *)malloc(nxny*sizeof(int));
    if (iaudiag == 0) perror("malloc");

    iwrk = (int *)malloc(5*nxny*sizeof(int));
    if (iwrk == 0) perror("malloc");

/*
    generate the matrix in the five diagonal form.
*/

    genmat(nx, ny, nxny, av1, av2, av3, av4, av5);

/*
    generate xo, the true solution
*/

    for (i=0; i<nxny; i++)
        xo[i] = 1.0;

/*
    obtain the corresponding right hand side (in bx).
*/

    matvec(nx, ny, nxny, av1, av2, av3, av4, av5, xo, bx);

/*
    convert the matrix into symmetric profile-in skyline storage mode
*/

    convert_to_skyline1 (nx, ny, nxny, av1, av2, av3, av4, av5,
                         au, iaudiag, &nau);

/*
    set the parameters (integer and real) for the factorization
*/

    iparam[0] = 100;
    iparam[1] = 100;
    iparam[2] = 2*nxny;
    iparam[3] = 0;
    iparam[4] = 6;
    iparam[5] = 2;
    iparam[6] = 1;
    iparam[7] = 1;
    iparam[8] = 0;
    iparam[9] = 0;
    iparam[10] = 0;
    iparam[12] = 0;

    rparam[0] = 1.0e-12;

/*
    call the factorization routine
*/

    dsskyf(&nxny, au, iaudiag, &nau,
           iparam, rparam, iwrk, &dum, &ierror);

    if (ierror != 0)
        printf("exit from routine dsskyf with error: %d\n",ierror);
```

(continued on next page)

Example 13–3 (Cont.) Skyline Solver Using Factorize and Solve Routines (C Code)

```
/*
    call the solve routine
*/

    ldbx = nxny;
    nbx = 1;

    dsskys(&nxny, au, iaudiag, &nau,
            bx, &ldbx, &nbx,
            iparam, rparam, iwrk, &dum, &ierror);

    if (ierror != 0)
        printf("exit from routine dsskyf with error: %d\n",ierror);

/*
    find the maximum absolute error in the solution
*/

    max1 = ABS((bx[0]-xo[0]));

    for (i=1; i<nxny; i++)
        {
            tmp1 = ABS((bx[i]-xo[i]));
            max1 = MAX((max1),(tmp1));
        }

/*
    print the maximum absolute error
*/

    printf("maximum error in the solution: %.10e\n",max1);

/*
    release the memory
*/

    free(au);
    free(xo);
    free(bx);
    free(av1);
    free(av2);
    free(av3);
    free(av4);
    free(av5);

    free(iaudiag);
    free(iwrk);

}  /* end of main() */


/*
    routine to generate the matrix for the example (in the five
    diagonal form). av1 and av2 are the subdiagonals, av3 is the main
    diagonal and av4 and av5 are the superdiagonals.
*/

void genmat(int nx, int ny, int nxny, double av1[], double av2[],
            double av3[], double av4[], double av5[])
{
    int i, j, k;
```

(continued on next page)

Example 13–3 (Cont.) Skyline Solver Using Factorize and Solve Routines (C Code)

```
    for (i=0; i<nxny; i++)
    {
        av3[i] = 4.0;
        av1[i] = 0.0;
        av2[i] = 0.0;
        av4[i] = 0.0;
        av5[i] = 0.0;
    }

    for (j=0; j<ny; j++)
    {
        for (i=1; i<nx; i++)
        {
            k = j * nx + i;
            av2[k] = -1.0;
        }
        for (i=0; i<(nx-1); i++)
        {
            k = j * nx + i;
            av4[k] = -1.0;
        }
    }

    for (j=1; j<ny; j++)
    {
        for (i=0; i<nx; i++)
        {
            k = j * nx + i;
            av1[k] = -1.0;
        }
    }

    for (j=0; j<(ny-1); j++)
    {
        for (i=0; i<nx; i++)
        {
            k = j * nx + i;
            av5[k] = -1.0;
        }
    }
}  /* end of genmat() */


/*
    routine to obtain the matrix vector multiply, using the matrix
    from the five-diagonal form. tmp1 is the input vector; tmp2 is
    the output vector.
*/

void matvec(int nx, int ny, int nxny,
            double av1[], double av2[],
            double av3[], double av4[], double av5[],
            double *tmp1, double *tmp2)
{
    int  i;

    for (i=0; i<nxny; i++)
        tmp2[i] = av3[i] * tmp1[i];

    for (i=0; i<(nxny-1); i++)
        tmp2[i] = tmp2[i] + av4[i] * tmp1[i+1];
```

(continued on next page)

Example 13–3 (Cont.) Skyline Solver Using Factorize and Solve Routines (C Code)

```
    for (i=1; i<nxny; i++)
        tmp2[i] = tmp2[i] + av2[i] * tmp1[i-1];

    for (i=0; i<(nxny-nx); i++)
        tmp2[i] = tmp2[i] + av5[i] * tmp1[i+nx];

    for (i=nx; i<nxny; i++)
        tmp2[i] = tmp2[i] + av1[i] * tmp1[i-nx];

}  /* end of matvec() */


/*
    routine for converting the matrix from the five-diagonal form
    to the symmetric profile-in skyline form. only the upper
    triangular part is stored. the matrix is returned in the real
    and integer arrays, au and iaudiag, respectively.
*/

void convert_to_skyline1(int nx, int ny, int nxny,
                         double av1[], double av2[], double av3[],
                         double av4[], double av5[],
                         double *a, int *ia, int *nau)
{
    int  i, j, index;

    index = 0;
    a[index] = av3[0];
    ia[0] = 1;
    index++;

    for (i=1; i<nx; i++)
    {
        a[index] = av4[i-1];
        a[index+1] = av3[i];
        index += 2;
        ia[i] = index ;
    }

    for (i=nx; i<nxny; i++)
    {
        a[index] = av5[i-nx];
        index++;
        for (j=1; j<=(nx-2); j++)
        {
            a[index] = 0.0;
            index++;
        }
        a[index] = av4[i-1];
        index++;
        a[index] = av3[i];
        index++;
        ia[i] = index;
    }

    *nau = index - 1;

}  /* end of convert_to_skyline1() */
```

(continued on next page)

Example 13–3 (Cont.) Skyline Solver Using Factorize and Solve Routines (C Code)

Output from Example 3

```
   ------------------------------------------------------------------------
   u^t-d-u factorization on a symmetric matrix
   storage scheme: symmetric profile-in skyline
   order of matrix:        100
   factorization starts at row/column:         1
   size of the envelope (upper triangle):       1009
   number of initial zeros in the envelope:        729
   percentage sparsity of the envelope:  72.25 %
   maximum column height (including diagonal):         11
   average column height (including diagonal):           10.09
   root-mean-square column height (including diagonal):           10.45
   u^t-d-u factorization completed without error
   ------------------------------------------------------------------------
   u^t-d-u solve for a symmetric matrix
   storage scheme: symmetric profile-in skyline
   order of matrix:        100
   number of right hand sides:         1
   u^t-d-u solve completed without error
maximum error in the solution: 1.1102230246e-15
```

Example 13–4 Skyline Solver Using Factorize and Solve Routines (C++ Code)

```
//
// ************************************************************************
// *
// *  Copyright Digital Equipment Corporation 1993 - 1995. All rights reserved.
// *
// *  Restricted Rights: Use, duplication, or disclosure by the U.S.
// *  Government is subject to restrictions as set forth in subparagraph
// *  (c) (1) (ii) of DFARS 252.227-7013, or in FAR 52.227-19, or in FAR
// *  52.227-14 Alt. III, as applicable.
// *
// *  This software is proprietary to and embodies the confidential
// *  technology of Digital Equipment Corporation. Possession, use, or
// *  copying of this software and media is authorized only pursuant to a
// *  valid written license from Digital or an authorized sublicensor.
// *
// ************************************************************************
//
//
//   This is an example program to illustrate the use of the iterative
//   solver ditsol_pcg from a C application program. The program generates
//   the matrix and the preconditioner, calls the solver and prints the
//   maximum error in the solution. The right hand side of the problem is
//   generated assuming a known solution. The problem used is identical to
//   the one in the example section of the chapter on iterative solvers
//   in the DXML Reference Guide.
//
//   This program illustrates the following:
//        - routine naming convention for Digital Unix and VMS
//        - Differences in endexing arrays:
//             C default: x[n] -> 0 to (n-1)
//             Fortran default: x(n) -> 1 to n
//        - how to use two dimensional arrays in C to interface with a
//          Fortran library routine
//        - how to use the matrix-free formulation from a C program
//
//    For more detailed explanation of the routines used, please
//    check the DXML Reference Manual.
//
//    Note: the code used in this example works on both Digital Unix and
//    VMS. Conditional compilation is used to select the statements
//    appropriate to each operating system.
//
//    All output from this program is sent to the screen.
//
//

#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <new.h>

//
//   Add trailing underscores to Fortran routines on Digital Unix.
//
```

(continued on next page)

Example 13–4 (Cont.) Skyline Solver Using Factorize and Solve Routines (C++ Code)

```
#if !defined(vms) && !defined(__vms)
#define ditsol_defaults ditsol_defaults_
#define dcreate_ilu_sdia dcreate_ilu_sdia_
#define ditsol_pcg ditsol_pcg_
#define dmatvec_sdia dmatvec_sdia_
#define dapply_ilu_sdia dapply_ilu_sdia_
#endif

inline double ABS(double x)
{
    return(((x) < 0) ? -(x) : (x));
}

inline double MAX(double x, double y)
{
    return(((x) < (y)) ? (y) : (x));
}

extern void (*set_new_handler(void (*memory_err)()))();
void memory_err()
{
    cout << "memory allocation error\n";
    exit(1); // quit program
}

extern void pcondl1(int &, int [], double [],
                    double [], int [],
                    double [], int [],
                    double [], double [], double [], int &);
extern void  matvec1(int &, int [], double [],
                     double [], int [],
                     double [], double [], double [], int &);
extern void  genmat1(int, int, int,
                     double [], int [],
                     int, int);

//
//   Declare the Fortran routines
//
```

(continued on next page)

Example 13–4 (Cont.) Skyline Solver Using Factorize and Solve Routines (C++ Code)

```
extern "C"
{
void ditsol_defaults(int [], double []);
void dcreate_ilu_sdia(double [], int [],
                      int &, int &,
                      double [], int [],
                      int &);
void ditsol_pcg(void (*)(int &, int [], double [],
                         double [], int [],
                         double [], double [], double [],
                         int &),
                void (*)(int &, int [], double [],
                         double [], int [],
                         double [], int [],
                         double [], double [], double [], int &),
                double &, double &,
                double [], int [],
                double [], double [], int &,
                double [], int [],
                double &, int &,
                int [], double [],
                int &, double [],
                int &);
void dmatvec_sdia(int &, double [], int [],
                  int &, int &, double [], double [], double [],
                  int &);
void dapply_ilu_sdia(int &,
                     double [], int [],
                     int [], int [],
                     double [], double [], int &);
}

//
//     illustrating the use of the iterative solver:
//     preconditioned conjugate gradient method, with incomplete
//     cholesky preconditioning. the matrix is stored using the
//     symmetric diagonal format
//

void main()
{
    double *a_sdia;
    double *a_ilu;
    double *rwork1;
    double *rhs;
    double *x;
    double *xo;

    double rparam[50];

    double dum, max1, tmp1;

    int *index_sdia;
    int *index_ilu;

    int iparam[50];

    int nxny, length, ndim, nzeros;
    int i, j, idum, ierror;
    int job;

// set up exception handler
```

(continued on next page)

Example 13–4 (Cont.) Skyline Solver Using Factorize and Solve Routines (C++ Code)

```
    set_new_handler(memory_err);

// define the problem size

    const int nx = 10;
    const int ny = 10;

    nxny = nx * ny;
    ndim = nxny;
    nzeros = 3;

// allocate memory for the 2-dimensional arrays

    a_sdia = new double [nzeros*ndim];
    a_ilu = new double [nzeros*ndim];

// allocate memory for the 1-dimensional arrays

    rwork1 = new double [4*nxny];
    rhs = new double [nxny];
    x = new double [nxny];
    xo = new double [nxny];

    index_sdia = new int [nzeros];
    index_ilu = new int [nzeros];

// set the parameters (integer and real)

    ditsol_defaults(iparam, rparam);

    iparam[2] = 0;
    iparam[3] = 4 * nxny;

// direct all output to the screen

    iparam[4] = 6;
    iparam[5] = 3;
    iparam[6] = 4;

// generate the matrix

    genmat1(nx, ny, nxny,
            a_sdia, index_sdia,
            ndim, nzeros);

    iparam[30] = nzeros;
    iparam[31] = ndim;

// generate xo, the true solution

    for (i=0; i<nxny; i++)
        xo[i] = 1.0;

// obtain the right hand side

    job = 0;
    matvec1(job, iparam, rparam,
            a_sdia, index_sdia,
            &dum, xo, rhs, nxny);

// obtain initial guess (all zeros)

    for (i=0; i<nxny; i++)
        x[i] = 0.0;

// generate the preconditioner
```

(continued on next page)

Example 13–4 (Cont.) Skyline Solver Using Factorize and Solve Routines (C++ Code)

```
    dcreate_ilu_sdia(a_sdia, index_sdia,
                     ndim, nzeros,
                     a_ilu, index_ilu,
                     nxny);

// call the solver

    ditsol_pcg(matvec1, pcondl1, dum, dum,
               a_sdia, index_sdia,
               x, rhs, nxny,
               a_ilu, index_ilu,
               dum, idum,
               iparam, rparam,
               idum, rwork1,
               ierror);

    if (ierror != 0)
        cout << "ditsol_pcg returned with error flag: " << ierror
            << endl;

// find the maximum absolute error in the solution

    max1 = ABS((x[0]-xo[0]));

    for (i=1; i<nxny; i++)
        {
            tmp1 = ABS((x[i]-xo[i]));
            max1 = MAX((max1),(tmp1));
        }
// print the maximum absolute error

    cout << "maximum error in the solution: " << max1 << endl;

// deallocate the memory

    delete a_sdia;
    delete a_ilu;
    delete rwork1;
    delete xo;
    delete x;
    delete rhs;

    delete index_sdia;
    delete index_ilu;

}  // end of main()


//
//    generate the matrix for the problem described in the chapter on
//    iterative solvers in the DXML Reference Guide
//

void genmat1(int nx, int ny, int nxny,
             double a[], int index[],
             int ndim, int nzeros)
{
    int  i, j, k;

    for (j=0; j<nxny; j++)
        for (i=1; i<nzeros; i++)
            a[i*ndim+j] = 0.0;
```

(continued on next page)

Example 13–4 (Cont.) Skyline Solver Using Factorize and Solve Routines (C++ Code)

```
    for (j=0; j<nxny; j++)
        a[0*ndim+j] = 4.0;

    for (j=0; j<ny; j++)
        for (i=1; i<nx; i++)
            {
                k = j * nx + i;
                a[2*ndim+k] = -1.0;
            }

    for (j=1; j<ny; j++)
        for (i=0; i<nx; i++)
            {
                k = j * nx + i;
                a[1*ndim+k] = -1.0;
            }

    index[0] = 0;
    index[2] = -1;
    index[1] = -nx;

}  // end of genmat1()


//
//    provide the matrix-vector routine using the standard parameter list
//    as described in the DXML Reference Guide
//

void matvec1(int &job, int iparam[], double rparam[],
             double a[], int ia[],
             double w[], double x[], double y[], int &n)
{
    int  nzeros, ndim;
    double dum;

    nzeros = iparam[30];
    ndim = iparam[31];

    dmatvec_sdia(job, a, ia, ndim, nzeros, &dum, x, y, n);

}  // end of matvec1()


//
//    provide the left preconditioning routine using the standard parameter
//    list as described in the DXML Refernce Guide
//

void pcondl1(int &job, int iparam[], double rparam[],
             double ql[], int iql[],
             double a[], int ia[],
             double w[], double x[], double y[], int &n)
{
    int  nzeros, ndim, job1;
    double *tmp;

// ilu preconditioning

    nzeros = iparam[30];
    ndim = iparam[31];

// allocate temporary storage

    tmp = new double [n];
```

(continued on next page)

Example 13–4 (Cont.) Skyline Solver Using Factorize and Solve Routines (C++ Code)

```
    job1 = 0;
    dapply_ilu_sdia(job1, ql, iql, &ndim, &nzeros, x, tmp, n);
    job1 = 1;
    dapply_ilu_sdia(job1, ql, iql, &ndim, &nzeros, tmp, y, n);

// deallocate temporary storage

    delete tmp;

}  // end of pcondl1()
```

Output from Example 4

```
   method used : cg with spd split preconditioning
   order of system =       100
   stopping criterion used =     1
   maximum iterations allowed =         100
   tolerance for convergence =   0.10000000E-05
      iteration =          0    stopping test = 0.69282032E+01
      iteration =          1    stopping test = 0.19194193E+01
      iteration =          2    stopping test = 0.12100937E+01
      iteration =          3    stopping test = 0.52439623E+00
      iteration =          4    stopping test = 0.84029860E-01
      iteration =          5    stopping test = 0.20539881E-01
      iteration =          6    stopping test = 0.34309306E-02
      iteration =          7    stopping test = 0.47063334E-03
      iteration =          8    stopping test = 0.16605002E-03
      iteration =          9    stopping test = 0.45072557E-04
      iteration =         10    stopping test = 0.72087304E-05
      iteration =         11    stopping test = 0.88047573E-06
   solution obtained after          11  iterations
   normal exit from solver
   final value of stopping test = 0.88047573E-06

maximum error in the solution: 5.6260082260e-08
```

Sparse Direct Solver Subprograms

This section provides descriptions of the direct solver subprograms for real double precision operations. The operations are grouped by functionality, starting with the routines for the symmetric matrices, followed by the routines for the unsymmetric matrices.

DSSKYN
Symmetric Sparse Matrix Norm Evaluation Using Skyline Storage Scheme

Format

DSSKYN (n, au, iaudiag, nau, iparam, rparam, iwrk, rwrk, ierror)

Arguments

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

au
real*8
On entry, an array of length at least nau, containing the matrix A stored in the skyline storage scheme, using either the profile-in or the diagonal-out storage mode.
On exit, **au** is unchanged.

iaudiag
integer*4
On entry, an array of length at least n for the profile-in storage mode and $n+1$ for the diagonal-out storage mode, containing the pointers to the locations of the diagonal elements in array AU.
On exit, **iaudiag** is unchanged.

nau
integer*4
On entry, the number of elements in array AU. nau is also the envelope size of the symmetric part of the matrix A. For the profile-in storage mode, $nau =$ IAUDIAG(n). For the diagonal-out storage mode, $nau =$ IAUDIAG($n+1$) - 1.
On exit, **nau** is unchanged.

iparam
integer*4
An array of length at least 100, containing the integer parameters for the evaluation of the matrix norms.

iparam(1): niparam
On entry, defines the length of the array IPARAM. $niparam \geq 100$.
On exit, **iparam(1)** is unchanged.

iparam(2): nrparam
On entry, defines the length of the array RPARAM. $nrparam \geq 100$.
On exit, **iparam(2)** is unchanged.

iparam(3): niwrk
On entry, defines the size of the integer work array, IWRK. $niwrk \geq n$.
On exit, **iparam(3)** is unchanged.

iparam(4): nrwrk
On entry, defines the size of the real work array, RWRK. $nrwrk \geq n$
On exit, **iparam(4)** is unchanged.

iparam(5): iounit
On entry, defines the I/O unit number for printing error messages and information from the routine DSSKYN. The I/O unit must be opened in the calling subprogram. If *iounit* ≤ 0, no output is generated.
On exit, **iparam(5)** is unchanged.

iparam(6): iolevel
On entry, defines the message level that determines the amount of information printed out to *iounit*, when *iounit* > 0:

iolevel = 0 : fatal error messages only
iolevel = 1 : error messages and minimal information
iolevel = 2 : error messages and detailed information

On exit, **iparam(6)** is unchanged.

iparam(7): idefault
On entry, defines if the default values should be used in arrays IPARAM and RPARAM. If *idefault* = 0, then the following default values are assigned:

IPARAM(1) = *niparam* = 100
IPARAM(2) = *nrparam* = 100
IPARAM(6) = *iolevel* = 0
IPARAM(8) = *istore* = 1
IPARAM(9) = *inorm* = 1

If *idefault* = 1, then you must assign values to the above variables before the call to the DSSKYN routine.
On exit, **iparam(7)** is unchanged.

iparam(8): istore
On entry, defines the type of storage scheme used for the skyline matrix. If *istore* = 1, the matrix A is stored using the profile-in storage mode; if *istore* = 2, the matrix A is stored using the diagonal-out storage mode. Default: *istore* = 1.
On exit, **iparam(8)** is unchanged.

iparam(9): inorm
On entry, defines the matrix quantity to be evaluated:

inorm = 1 : 1-norm of A
inorm = 2 : ∞-norm of A
inorm = 3 : Frobenius norm of A
inorm = 4 : Maximum absolute value of A

Default: *inorm* = 1.
On exit, **iparam(9)** is unchanged.

rparam
real*8
An array of length at least 100, containing the real parameters for the norm evaluation.

rparam(1): anorm
On entry, an unspecified variable.
On exit, **rparam(1)** contains the matrix quantity evaluated, as defined by the value of IPARAM(9) = *inorm*.

iwrk
integer*4
On entry, an array of length at least n used for integer workspace.
On exit, **iwrk** contains information used by the routine DSSKYN. This information is not used by any other routine and therefore can be overwritten.

rwrk
real*8
On entry, an array of length at least n used for real workspace.
On exit, **rwrk** contains information used by the routine DSSKYN. This information is not used by any other routine and therefore can be overwritten.

ierror
integer*4
On entry, an unspecified variable.
On exit, **ierror** contains the error flag. A value of zero indicates a normal exit from the routine DSSKYN.

Description

DSSKYN evaluates the following quantities for the symmetric matrix A:

- 1-norm of A:
$$||A||_1 = \max_j \sum_i |a_{ij}|$$
- ∞-norm of A:
$$||A||_\infty = \max_i \sum_j |a_{ij}|$$
- Frobenius-norm of A:
$$||A||_F = \sqrt{\sum_i \sum_j |a_{ij}|^2}$$
- Largest absolute value of A:
$$\max_{i,j} |a_{ij}|$$

The last quantity in the above list is not a matrix norm. The quantity evaluated is determined by the value of *inorm*, with the 1-norm and ∞-norm being identical for symmetric matrices.

The real and integer workspace used by the routine DSSKYN do not contain information for use by any other routines, and can therefore be overwritten.

As the routine DSSKYN requires the matrix A, it should be called prior to a call to the factorization routine DSSKYF, which overwrites the elements of A by its $U^T DU$ factorization.

DSSKYF
Symmetric Sparse Matrix Factorization Using Skyline Storage Scheme (Serial and Parallel Versions)

Format

DSSKYF (n, au, iaudiag, nau, iparam, rparam, iwrk, rwrk, ierror)

Arguments

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

au
real*8
On entry, an array of length at least nau, containing the matrix A stored in the skyline storage scheme, using either the profile-in or the diagonal-out storage mode.
On exit, **au** contains the $U^T DU$ factorization of the matrix A. **au** must remain unchanged between the call to the routine DSSKYF and any routine that uses the factors such as DSSKYS, DSSKYC, DSSKYR and DSSKYX.

iaudiag
integer*4
On entry, an array of length at least n for the profile-in storage mode and $n+1$ for the diagonal-out storage mode, containing the pointers to the locations of the diagonal elements in array AU.
On exit, **iaudiag** is unchanged.

nau
integer*4
On entry, the number of elements in array AU. nau is also the envelope size of the symmetric part of the matrix A. For the profile-in storage mode, $nau =$ IAUDIAG(n). For the diagonal-out storage mode, $nau =$ IAUDIAG(n+1) - 1.
On exit, **nau** is unchanged.

iparam
integer*4
An array of length at least 100, containing the integer parameters for the $U^T DU$ factorization.

iparam(1): niparam
On entry, defines the length of the array IPARAM. $niparam \geq 100$.
On exit, **iparam(1)** is unchanged.

iparam(2): nrparam
On entry, defines the length of the array RPARAM. $nrparam \geq 100$.
On exit, **iparam(2)** is unchanged.

iparam(3): niwrk
On entry, defines the size of the integer work array, IWRK. $niwrk \geq 2n$.
On exit, **iparam(3)** is unchanged.

iparam(4): nrwrk
On entry, defines the size of the real work array, RWRK. As the real work array is not used at present, *nrwrk* can be unspecified.
On exit, **iparam(4)** is unchanged.

iparam(5): iounit
On entry, defines the I/O unit number for printing error messages and information from the routine DSSKYF. The I/O unit must be opened in the calling subprogram. If *iounit* $\leq$0, no output is generated.
On exit, **iparam(5)** is unchanged.

iparam(6): iolevel
On entry, defines the message level that determines the amount of information printed out to *iounit*, when *iounit* > 0:

iolevel = 0 : fatal error messages only
iolevel = 1 : error messages and minimal information
iolevel = 2 : error messages and detailed information

On exit, **iparam(6)** is unchanged.

iparam(7): idefault
On entry, defines if the default values should be used in arrays IPARAM and RPARAM. If *idefault* = 0, then the following default values are assigned:

IPARAM(1) = *niparam* = 100
IPARAM(2) = *nrparam* = 100
IPARAM(6) = *iolevel* = 0
IPARAM(8) = *istore* = 1
IPARAM(9) = *ibeg* = 0
IPARAM(10) = *idet* = 0
IPARAM(11) = *ipvt* = 0
IPARAM(13) = *inertia* = 0
RPARAM(1) = *pvt_sml* = 10^{-12}

If *idefault* = 1, then you must assign values to the above variables before the call to the DSSKYF routine.
On exit, **iparam(7)** is unchanged.

iparam(8): istore
On entry, defines the type of storage scheme used for the skyline matrix. If *istore* = 1, the matrix A is stored using the profile-in storage mode; if *istore* = 2, the matrix A is stored using the diagonal-out storage mode. Default: *istore* = 1.
On exit, **iparam(8)** is unchanged.

iparam(9): ibeg
On entry, defines if full or partial factorization is to be performed. If *ibeg* = 0, then a full factorization is performed for rows and columns 1 through n. If *ibeg* > 0, then a partial factorization is performed starting from rows and columns $ibeg + 1$ through n, that is, rows and columns from 1 through $ibeg$ have already been factorized. Default: *ibeg* = 0.
On exit, **iparam(9)** is unchanged.

iparam(10): idet
On entry, defines if the determinant of the matrix A is to be calculated. If *idet* = 0, then the determinant is not calculated; if *idet* = 1, the determinant is

calculated as *det_base* * $10^{\text{det\_pwr}}$. See RPARAM(4) and RPARAM(5). Default: *idet* = 0.
On exit, **iparam(10)** is unchanged.

iparam(11): ipvt
On entry, defines if the factorization should continue when a small pivot (defined by RPARAM(1) is encountered. If *ipvt* = 0 and the absolute value of the pivot element is smaller than *pvt_sml* = RPARAM(1), then the factorization process is stopped and control returned to the calling subprogram. If *ipvt* = 1 and a pivot smaller than RPARAM(1) in absolute value is encountered in the factorization, the process continues. If *ipvt* = 2 and a pivot smaller than RPARAM(1) in absolute value is encountered in the factorization, it is replaced by a predetermined value *pvt_new* = RPARAM(2), and the factorization is continued. Default: *ipvt* = 0.
On exit, **iparam(11)** is unchanged.

iparam(12): ipvt_loc
On entry, an unspecified variable.
On exit, **iparam(12)** contains the location of the first pivot element smaller in absolute value than *pvt_sml*. The pivot element is returned in *pvt_val* = RPARAM(3). If **iparam(12)** = 0, then no such pivot element exists.

iparam(13): inertia
On entry, defines if the inertia of the matrix A should be calculated. The inertia of the symmetric matrix A is the triplet of integers (*ipeigen*, *ineigen*, *izeigen*), consisting of the number of positive, negative, and zero eigenvalues, respectively. If *inertia* = 0, then the inertia is not calculated; if *inertia* = 1, then the number of positive and negative eigenvalues are returned in *ipeigen* = IPARAM(14) and *ineigen* = IPARAM(15), respectively. An indication of the existence of zero eigenvalues is returned in *izeigen* = IPARAM(16). Default: *inertia* = 0.
On exit, **iparam(13)** is unchanged.

iparam(14): ipeigen
On entry, an unspecified variable.
On exit, if **inertia** = 1, **iparam(14)** contains the number of positive eigenvalues of the matrix A.

iparam(15): ineigen
On entry, an unspecified variable.
On exit, if *inertia* = 1, **iparam(15)** contains the number of negative eigenvalues of the matrix A.

iparam(16): izeigen
On entry, an unspecified variable.
On exit, if *inertia* = 1, **iparam(16)** indicates if the matrix A has any zero eigenvalues. If *izeigen* = 0, then the matrix A does not have a zero eigenvalue; if *izeigen* = 1, then the matrix A has at least one zero eigenvalue.

rparam
real*8
An array of length at least 100, containing the real parameters for the $U^T DU$ factorization.

rparam(1): pvt_sml
On entry, defines the value of the pivot element which is considered to be small. If a pivot element smaller than *pvt_sml*, in absolute value, is encountered in the factorization process, then, depending on the value of *ipvt* = IPARAM(11), the process either stops, continues or continues after the pivot is set equal to *pvt_new* = RPARAM(2). *pvt_sml* > 0. Recommended value: $10^{-15} \leq$ *pvt_sml* ≤ 1. Default: *pvt_sml* = 10^{-12}.
On exit, **rparam(1)** is unchanged.

rparam(2): pvt_new
On entry, defines the value to which the pivot element must be set if *ipvt* = 2 and the pivot element is less than *pvt_sml* in absolute value. *pvt_new* should be large enough to avoid overflow when calculating the reciprocal of the pivot element.
On exit, **rparam(2)** is unchanged.

rparam(3): pvt_val
On entry, an unspecified variable.
On exit, **rparam(3)** contains the value of the first pivot element smaller than *pvt_sml* in absolute value. This element occurs at the location returned in IPARAM(12). If no such pivot element is found, the value of *pvt_val* is unspecified.

rparam(4): det_base
On entry, an unspecified variable.
On exit, defines the base for the determinant of the matrix ***A***. If *idet* = 1, the determinant is calculated as *det_base* * $10^{\text{det\_pwr}}$. $1.0 \leq$ *det_base* < *10.0*.

rparam(5): det_pwr
On entry, an unspecified variable.
On exit, defines the power for the determinant of the matrix ***A***. If *idet* = 1, the determinant is calculated as *det_base* * $10^{\text{det\_pwr}}$.

iwrk
integer*4
On entry, an array of length at least $2n$ used for integer workspace.
On exit, **iwrk** contains information for use by routines that use the factorization such as DSSKYS, DSSKYC, DSSKYR, and DSSKYX. The first $2n$ elements of **iwrk** should therefore be passed unchanged to these routines.

rwrk
real*8
On entry, an array used for real workspace.
On exit, **rwrk** is unchanged. Presently, **rwrk** is not used by the routine DSSKYF. It can be a dummy variable.

ierror
integer*4
On entry, an unspecified variable.
On exit, **ierror** contains the error flag. A value of zero indicates a normal exit from the routine DSSKYF.

Description

DSSKYF obtains the factorization of the symmetric matrix A as:

$$A = U^T DU$$

where D is a diagonal matrix, and U is a unit upper triangular matrix. The matrix A is stored in a skyline form, using either the profile-in storage mode or the diagonal-out storage mode.

The routine DSSKYF does not perform any pivoting to preserve the numerical stability of the $U^T DU$ factorization. It is therefore primarily intended for the solution of symmetric positive (or negative) definite systems as they do not require pivoting for numerical stability. Caution is urged when using this routine for symmetric indefinite systems.

If a small pivot, in absolute value, *pvt_sml*, is encountered in the process of factorization, you have the option of either stopping the factorization process and returning to the calling sub-program, continuing the factorization process with the small value of the pivot, or continuing after setting the pivot equal to some predetermined value, *pvt_new*. The location of the first occurrence of a small pivot is returned in *ipvt_loc* and its value in *pvt_val*.

In addition to the $U^T DU$ factorization, you can also obtain the determinant of A, the number of positive and negative eigenvalues of the matrix A and an indication of the existence of zero eigenvalues. A partial factorization can also be obtained by appropriately setting the value of *ibeg*. If $ibeg > 0$, then factorization begins at row and column $ibeg + 1$; the rows and columns from 1 to *ibeg* are assumed to have been already factorized. When $ibeg > 0$, the determinant of A, the inertia of A and the statistics on the matrix are calculated from rows and columns *ibeg* + 1 through n. If the factorization process is stopped at row i due to a small pivot, then the inertia, determinant and statistics on the matrix are evaluated for rows $ibeg + 1$ through $(i - 1)$.

The data in the first $2n$ elements of the integer workspace, IWRK, are used in routines that use the $U^T DU$ factorization, such as DSSKYS, DSSKYC and DSSKYR. This data must therefore remain unchanged between the call to DSSKYF and any one of these routines. The real workspace array, RWRK, is not used at present.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DSSKYS
Symmetric Sparse Matrix Solve Using Skyline Storage Scheme

Format

DSSKYS (n, au, iaudiag, nau, bx, ldbx, nbx, iparam, rparam, iwrk, rwrk, ierror)

Arguments

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

au
real*8
On entry, an array of length at least nau, containing the $U^T DU$ factorization of the matrix A stored in the skyline storage scheme, using either the profile-in or the diagonal-out storage mode. The factorization has been obtained by a prior call to the routine DSSKYF. **au** must remain unchanged between calls to the routines DSSKYF and DSSKYS.
On exit, **au** is unchanged.

iaudiag
integer*4
On entry, an array of length at least n for the profile-in storage mode and $(n+1)$ for the diagonal-out storage mode, containing the pointers to the locations of the diagonal elements in array AU.
On exit, **iaudiag** is unchanged.

nau
integer*4
On entry, the number of elements in array AU. nau is also the envelope size of the symmetric part of the matrix A. For the profile-in storage mode, nau = IAUDIAG(n). For the diagonal-out storage mode, nau = IAUDIAG($n+1$) - 1.
On exit, **nau** is unchanged.

bx
real*8
On entry, a two-dimensional array BX of order $ldbx$ by at least nbx, containing the nbx right sides.
On exit, **bx** contains the solutions for the nbx systems.

ldbx
integer*4
On entry, the leading dimensional of array BX. $ldbx \geq n$.
On exit, **ldbx** is unchanged.

nbx
integer*4
On entry, the number of right sides.
On exit, **nbx** is unchanged.

iparam
integer*4
An array of length at least 100, containing the integer parameters for the matrix solve operation.

iparam(1): niparam
On entry, defines the length of the array IPARAM. $niparam \geq 100$.
On exit, **iparam(1)** is unchanged.

iparam(2): nrparam
On entry, defines the length of the array RPARAM. As the real parameter array is not used at present, *nrparam* can be unspecified.
On exit, **iparam(2)** is unchanged.

iparam(3): niwrk
On entry, defines the size of the integer work array, IWRK. $niwrk \geq 2n$.
On exit, **iparam(3)** is unchanged.

iparam(4): nrwrk
On entry, defines the size of the real work array, RWRK. As the real work array is not used at present, *nrwrk* can be unspecified.
On exit, **iparam(4)** is unchanged.

iparam(5): iounit
On entry, defines the I/O unit number for printing error messages and information from the routine DSSKYS. The I/O unit must be opened in the calling subprogram. If $iounit \leq 0$, no output is generated.
On exit, **iparam(5)** is unchanged.

iparam(6): iolevel
On entry, defines the message level that determines the amount of information printed out to *iounit*, when *iounit* > 0:

iolevel = 0 : fatal error messages only
iolevel = 1 : error messages and minimal information
iolevel = 2 : error messages and detailed information

On exit, **iparam(6)** is unchanged.

iparam(7): idefault
On entry, defines if the default values should be used in arrays IPARAM and RPARAM. If *idefault* = 0, then the following default values are assigned:

IPARAM(1) = *niparam* = 100
IPARAM(6) = *iolevel* = 0
IPARAM(8) = *istore* = 1

If *idefault* = 1, then you must assign values to the above variables before the call to the DSSKYS routine.
On exit, **iparam(7)** is unchanged.

iparam(8): istore
On entry, defines the type of storage scheme used for the skyline matrix. If *istore* = 1, the matrix A is stored using the profile-in storage mode; if *istore* = 2, the matrix A is stored using the diagonal-out storage mode. The storage scheme used in the routines DSSKYF and DSSKYS must be identical. Default: *istore* = 1.
On exit, **iparam(8)** is unchanged.

rparam
real*8
An array of length at least 100, containing the real parameters for the solution. On exit, **rparam** is unchanged. **rparam** is not used by the routine DSSKYS at present, but is reserved for future use. It can be a dummy variable.

iwrk
integer*4
On entry, an array of length at least $2n$ used for integer workspace. The first $2n$ elements of the array IWRK, generated by the routine DSSKYF, should be passed unchanged to the routine DSSKYS.
On exit, the first $2n$ elements of **iwrk** are unchanged.

rwrk
real*8
On entry, an array used for real workspace.
On exit, **rwrk** is unchanged. Presently, **rwrk** is not used by the routine DSSKYS, but is reserved for future use. It can be a dummy variable.

ierror
integer*4
On entry, an unspecified variable.
On exit, **ierror** contains the error flag. A value of zero indicates a normal exit from the routine DSSKYS.

Description

DSSKYS solves the system:

$$AX = B$$

where A is a symmetric matrix stored in a skyline form, using either the profile-in storage mode or the diagonal-out storage mode; B is a matrix of nbx right sides and X is the matrix of the corresponding nbx solution vectors. On entry to the routine DSSKYS, the array BX contains the nbx right sides; on exit, these are overwritten by the solution vectors.

The matrix A has been factorized as:

$$A = U^T DU$$

by a prior call to the routine DSSKYF. U is a unit upper triangular matrix and D is a diagonal matrix. The first $2n$ elements of the integer workspace array IWRK, generated by DSSKYF, contain information for use by DSSKYS and therefore must remain unchanged between the calls to the routines DSSKYF and DSSKYS.

The real work array, RWRK, is not used at present. The storage scheme used in the routines DSSKYF and DSSKYS must be identical.

Once the factorization has been obtained, the routine DSSKYS can be used to solve a system with multiple right hand sides, by setting $nbx > 1$. The routine can also be called repeatedly, provided the first $2n$ elements of the work array IWRK remain unchanged between calls.

DSSKYC
Symmetric Sparse Matrix Condition Number Estimator Using Skyline Storage Scheme

Format

DSSKYC (n, au, iaudiag, nau, iparam, rparam, iwrk, rwrk, ierror)

Arguments

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

au
real*8
On entry, an array of length at least nau, containing the $U^T DU$ factorization of the matrix A stored in the skyline storage scheme, using either the profile-in, or the diagonal-out storage mode. The factorization has been obtained by a prior call to the routine DSSKYF. **au** must remain unchanged between calls to the routines DSSKYF and DSSKYC.
On exit, **au** is unchanged.

iaudiag
integer*4
On entry, an array of length at least n for the profile-in storage mode and $(n+1)$ for the diagonal-out storage mode, containing the pointers to the locations of the diagonal elements in array AU.
On exit, **iaudiag** is unchanged.

nau
integer*4
On entry, the number of elements in array AU. **nau** is also the envelope size of the symmetric part of the matrix A. For the profile-in storage mode, $nau =$ IAUDIAG(n). For the diagonal-out storage mode, $nau =$ IAUDIAG($n+1$) - 1.
On exit, **nau** is unchanged.

iparam
integer*4
An array of length at least 100, containing the integer parameters for the condition number estimator.

iparam(1): niparam
On entry, defines the length of the array IPARAM. $niparam \geq 100$.
On exit, **iparam(1)** is unchanged.

iparam(2): nrparam
On entry, defines the length of the array RPARAM. $nrparam \geq 100$.
On exit, **iparam(2)** is unchanged.

iparam(3): niwrk
On entry, defines the size of the integer work array, IWRK. $niwrk \geq 3n$.
On exit, **iparam(3)** is unchanged.

iparam(4): nrwrk
On entry, defines the size of the real work array, RWRK. $nrwrk \geq 2n$.
On exit, **iparam(4)** is unchanged.

iparam(5): iounit
On entry, defines the I/O unit number for printing error messages and information from the routine DSSKYC. The I/O unit must be opened in the calling subprogram. If *iounit* ≤ 0, no output is generated.
On exit, **iparam(5)** is unchanged.

iparam(6): iolevel
On entry, defines the message level that determines the amount of information printed out to *iounit*, when *iounit* > 0:

iolevel = 0 : fatal error messages only
iolevel = 1 : error messages and minimal information
iolevel = 2 : error messages and detailed information

On exit, **iparam(6)** is unchanged.

iparam(7): idefault
On entry, defines if the default values should be used in arrays IPARAM and RPARAM. If *idefault* = 0, then the following default values are assigned:

IPARAM(1) = *niparam* = 100
IPARAM(2) = *nrparam* = 100
IPARAM(6) = *iolevel* = 0
IPARAM(8) = *istore* = 1

If *idefault* = 1, then you must assign values to the above variables before the call to the DSSKYC routine.
On exit, **iparam(7)** is unchanged.

iparam(8): istore
On entry, defines the type of storage scheme used for the skyline matrix. If *istore* = 1, the matrix A is stored using the profile-in storage mode; if *istore* = 2, the matrix A is stored using the diagonal-out storage mode. The storage scheme used in the routines DSSKYF and DSSKYC must be identical. Default: *istore* = 1.
On exit, **iparam(8)** is unchanged.

rparam
real*8
An array of length at least 100, containing the real parameters for the condition number estimator.

rparam(1): anorm
On entry, the 1-norm of the matric A, which has been obtained by a prior call to routine DSSKYN.
On exit, **rparam(1)** is unchanged.

rparam(2): ainorm
On entry, an unspecified variable.
On exit, **rparam(2)** contains the estimate of the 1-norm of A^{-1}.

rparam(3): rcond
On entry, an unspecified variable.
On exit, **rparam(3)** contains the reciprocal of the estimate of the 1-norm condition number of the matrix A.

iwrk
integer*4
On entry, an array of length at least $3n$ used for integer workspace. The first $2n$ elements of **iwrk** generated by the routine DSSKYF must be passed unchanged to the routine DSSKYC.
On exit, the first $2n$ elements of **iwrk** are unchanged. The next n elements are used as integer workspace by DSSKYC.

rwrk
real*8
On entry, an array of length at least $2n$ used for integer workspace.
On exit, the first $2n$ elements of **rwrk** are overwritten.

ierror
integer*4
On entry, an unspecified variable.
On exit, **ierror** contains the error flag. A value of zero indicates a normal exit from the routine DSSKYC.

Description

DSSKYC obtains the reciprocal of the estimate of the 1-norm condition number of the symmetric matrix A as:

$$rcond(A) = \frac{1}{||A|| \cdot ||A^{-1}||}$$

The 1-norm of A^{-1} is obtained using the LAPACK routine DLACON, which uses Higham's modification [Higham 1988] of Hager's method [Hager 1984]. This routine uses reverse communication for the evaluation of matrix-vector products. As the matrix under consideration is A^{-1}, routine DSSKYC requires calls to the routine DSSKYS. Hence the first $2n$ elements of the integer work array, IWRK, which are generated by the routine DSSKYF, and used by the routine DSSKYS, must remain unchanged between the calls to the routines DSSKYF and DSSKYC. The storage scheme used in the routines DSSKYF and DSSKYC must be identical.

DSSKYR
Symmetric Sparse Iterative Refinement Using Skyline Storage Scheme

Format

DSSKYR (n, au, auf, iaudiag, nau, b, ldb, x, ldx, ferr, berr, nbx, iparam, rparam, iwrk, rwrk, ierror)

Arguments

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

au
real*8
On entry, an array of length at least *nau*, containing the matrix A stored in the skyline storage scheme, using either the profile-in or the diagonal-out storage mode.
On exit, **au** is unchanged.

auf
real*8
On entry, an array of length at least *nau*, containing $U^T DU$ factorization of the matrix A stored in the skyline storage scheme, using either the profile-in or the diagonal-out storage mode. The factorization has been obtained by a prior call to the routine DSSKYF. **auf** must remain unchanged between calls to the routines DSSKYF and DSSKYR.
On exit, **auf** is unchanged.

iaudiag
integer*4
On entry, an array of length at least n for the profile-in storage mode and $(n + 1)$ for the diagonal-out storage mode, containing the pointers to the locations of the diagonal elements in arrays AU and AUF.
On exit, **iaudiag** is unchanged.

nau
integer*4
On entry, the number of elements in array AU. *nau* is also the envelope size of the symmetric part of the matrix A. For the profile-in storage mode, $nau =$ IAUDIAG(n). For the diagonal-out storage mode, $nau =$ IAUDIAG($n+1$) - 1.
On exit, **nau** is unchanged.

b
real*8
On entry, a two-dimensional array B of order *ldb* by at least *nbx*, containing the *nbx* right sides.
On exit, **b** is unchanged.

ldb
integer*4
On entry, the leading dimension of array B. $ldb \geq n$.
On exit, **ldb** is unchanged.

x
real*8
On entry, a two-dimensional array X of order ldx by at least nbx, containing the nbx solution vectors obtained after a call to the routine DSSKYS.
On exit, **x** contains the improved solutions obtained after iterative refinement.

ldx
integer*4
On entry, the leading dimension of array X. $ldx \geq n$.
On exit, **ldx** is unchanged.

ferr
real*8
On entry, an array FERR of length at least nbx, whose elements are unspecified variables.
On exit, **ferr** contains the estimated error bounds for each of the nbx solution vectors.

berr
real*8
On entry, an array BERR of length at least nbz, whose elements are unspecified variables.
On exit, **berr** contains the component-wise relative backward error for each of the nbx solution vectors.

nbx
integer*4
On entry, the number of right sides.
On exit, **nbx** is unchanged.

iparam
integer*4
An array of length at least 100, containing the integer parameters for the iterative refinement and error bounds calculation.

iparam(1): niparam
On entry, defines the length of the array IPARAM. $niparam \geq 100$.
On exit, **iparam(1)** is unchanged.

iparam(2): nrparam
On entry, defines the length of the array RPARAM. As the real parameter array is not used at present, *nrparam* can be unspecified.
On exit, **iparam(2)** is unchanged.

iparam(3): niwrk
On entry, defines the size of the integer work array, IWRK. $niwrk \geq 3n$.
On exit, **iparam(3)** is unchanged.

iparam(4): nrwrk
On entry, defines the size of the real work array, RWRK. $nrwrk \geq 3n$.
On exit, **iparam(4)** is unchanged.

iparam(5): iounit
On entry, defines the I/O unit number for printing error messages and information from the routine DSSKYR. The I/O unit must be opened in the calling subprogram. If *iounit* ≤ 0, no output is generated.

On exit, **iparam(5)** is unchanged.

iparam(6): iolevel
On entry, defines the message level that determines the amount of information printed out to *iounit*, when *iounit* > 0:

iolevel = 0 : fatal error messages only
iolevel = 1 : error messages and minimal information
iolevel = 2 : error messages and detailed information

On exit, **iparam(6)** is unchanged.

iparam(7): idefault
On entry, defines if the default values should be used in arrays IPARAM and RPARAM. If *idefault* = 0, then the following default values are assigned:

IPARAM(1) = *niparam* = 100
IPARAM(6) = *iolevel* = 0
IPARAM(8) = *istore* = 1
IPARAM(9) = *itmax* = 5

If *idefault* = 1, then you must assign values to the above variables before the call to the DSSKYR routine.
On exit, **iparam(7)** is unchanged.

iparam(8): istore
On entry, defines the type of storage scheme used for the skyline matrix. If *istore* = 1, the matrix A is stored using the profile-in storage mode; if *istore* = 2, the matrix A is stored using the diagonal-out storage mode. Default: *istore* = 1.
On exit, **iparam(8)** is unchanged.

iparam(9): itmax
On entry, defines the maximum number of iterations for the iterative refinement process. Default: *itmax* = 5.
On exit, **iparam(9)** is unchanged.

rparam
real*8
An array of length at least 100, containing the real parameters for the iterative refinement and error bounds calculation.
On exit, **rparam** is unchanged. **rparam** is not used by the routine DSSKYR at present, but is reserved for future use. It can be a dummy variable.

iwrk
integer*4
On entry, an array of length at least $3n$ used for integer workspace. The first $2n$ elements of the array IWRK, generated by the routine DSSKYF, should be passed unchanged to the routine DSSKYR.
On exit, the first $2n$ elements of **iwrk** are unchanged.

rwrk
real*8
On entry, an array of length at least $3n$ used for real workspace.
On exit, the first $3n$ elements of **rwrk** are overwritten.

ierror
integer*4
On entry, an unspecified variable.
On exit, **ierror** contains the error flag. A value of zero indicates a normal exit from the routine DSSKYR.

Description

DSSKYR obtains an improved solution to the system:

$$AX = B$$

via iterative refinement. This is done by calculating the matrix of residuals R using the matrix of solutions $\hat{X}$ obtained from DSSKYS, and obtaining a new matrix of solutions X_{new} as follows:

$$R = B - A\hat{X}$$

$$\delta X = A^{-1}R$$

and:

$$X_{new} = \hat{X} + \delta X$$

The process of iterative refinement therefore requires both the original matrix A as well as the $U^T DU$ factorization obtained via the routine DSSKYF. Since this routine overwrites the matrix A by the factorization, a copy of the matrix must be made prior to the call to DSSKYF. Further, both the right sides B and the solution vectors x are required during iterative refinement. Since the solution process in the routine DSSKYS overwrites the right sides with the solution vectors, a copy of the right sides must be made prior to the call to the routine DSSKYS.

In addition to the iterative refinement of the solution vectors, the routine DSSKYR also provides the component-wise relative backward error, $berr$ and the estimated forward error bound, $ferr$, for each solution vector [Arioli, Demmel, Duff 1989, Anderson et. al. 1992]. $berr$ is the smallest relative change in any entry of A or b that makes $\hat{x}$ an exact solution. $ferr$ bounds the magnitude of the largest entry in:

$$\hat{x} - x_{true}$$

divided by the magnitude of the largest entry in: $\hat{x}$.

The process of iterative refinement is continued as long as all of the following conditions are satisfied [Arioli, Demmel, Duff 1989]:

- The number of iterations of the iterative refinement process is less than IPARAM(9) = $itmax$.
- $berr$ reduces by at least a factor of 2 during the previous iteration.
- $berr$ is larger than the machine precision.

The routine DSSKYR is called after a call to the routine DSSKYF to obtain the $U^T DU$ factorization and a call to the routine DSSKYS to obtain the solution x. The first $2n$ elements of the integer workspace array IWRK, generated by DSSKYF, contain information for use by DSSKYR and therefore must remain unchanged between the calls to the routines DSSKYF and DSSKYR. The real work array, RWRK, is not used at present. The storage scheme used in the routines DSSKYF, DSSKYS, and DSSKYR must be identical.

DSSKYD
Symmetric Sparse Simple Driver Using Skyline Storage Scheme

Format

DSSKYD (n, au, iaudiag, nau, bx, ldbx, nbx, iparam, rparam, iwrk, rwrk, ierror)

Arguments

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

au
real*8
On entry, an array of length at least nau, containing the matrix A stored in the skyline storage scheme, using either the profile-in, or the diagonal-out storage mode.
On exit, **au** contains the $U^T DU$ factorization of the matrix A. **au** must remain unchanged between the call to the routine DSSKYD and any subsequent calls to the routines DSSKYS, DSSKYC, and DSSKYR.

iaudiag
integer*4
On entry, an array of length at least n for the profile-in storage mode and $(n + 1)$ for the diagonal-out storage mode, containing the pointers to the locations of the diagonal elements in array AU.
On exit, **iaudiag** is unchanged.

nau
integer*4
On entry, the number of elements in array AU. nau is also the envelope size of the symmetric part of the matrix A. For the profile-in storage mode, $nau =$ IAUDIAG(n). For the diagonal-out storage mode, $nau =$ IAUDIAG($n+1$) - 1.
On exit, **nau** is unchanged.

bx
real*8
On entry, a two-dimensional array BX of order $ldbx$ by at least nbx, containing the nbx right sides.
On exit, **bx** contains the solutions for the nbx systems.

ldbx
integer*4
On entry, the leading dimensional of array BX. $ldbx \geq n$.
On exit, **ldbx** is unchanged.

nbx
integer*4
On entry, the number of right sides.
On exit, **nbx** is unchanged.

iparam
integer*4
An array of length at least 100, containing the integer parameters for the simple driver.

iparam(1): niparam
On entry, defines the length of the array IPARAM. $niparam \geq 100$.
On exit, **iparam(1)** is unchanged.

iparam(2): nrparam
On entry, defines the length of the array RPARAM. $nrparam \geq 100$.
On exit, **iparam(2)** is unchanged.

iparam(3): niwrk
On entry, defines the size of the integer work array, IWRK. $niwrk \geq 2n$.
On exit, **iparam(3)** is unchanged.

iparam(4): nrwrk
On entry, defines the size of the real work array, RWRK. As the real work array is not used at present, *nrwrk* can be unspecified.
On exit, **iparam(4)** is unchanged.

iparam(5): iounit
On entry, defines the I/O unit number for printing error messages and information from the routine DSSKYD. The I/O unit must be opened in the calling subprogram. If *iounit* ≤ 0, no output is generated.
On exit, **iparam(5)** is unchanged.

iparam(6): iolevel
On entry, defines the message level that determines the amount of information printed out to *iounit*, when *iounit* > 0:

iolevel = 0 : fatal error messages only
iolevel = 1 : error messages and minimal information
iolevel = 2 : error messages and detailed information

On exit, **iparam(6)** is unchanged.

iparam(7): idefault
On entry, defines if the default values should be used in arrays IPARAM and RPARAM. If *idefault* = 0, then the following default values are assigned:

IPARAM(1) = *niparam* = 100
IPARAM(2) = *nrparam* = 100
IPARAM(6) = *iolevel* = 0
IPARAM(8) = *istore* = 1
IPARAM(9) = *ipvt* = 0
RPARAM(1) = *pvt_sml* = 10^{-12}

If *idefault* = 1, then you must assign values to the above variables before the call to the DSSKYD routine.
On exit, **iparam(7)** is unchanged.

iparam(8): istore
On entry, defines the type of storage scheme used for the skyline matrix. If *istore* = 1, the matrix A is stored using the profile-in storage mode; if *istore* = 2, the matrix A is stored using the diagonal-out storage mode. Default: *istore* = 1.
On exit, **iparam(8)** is unchanged.

iparam(9): ipvt
On entry, defines if the factorization should continue when a small pivot, (defined by RPARAM(1), is encountered. If *ipvt = 0* and the absolute value of the pivot element is smaller than *pvt_sml* = RPARAM(1), then the factorization process is stopped and control returned to the calling subprogram. If *ipvt=1* and a pivot smaller than RPARAM(1) in absolute value is encountered in the factorization, the process continues. If *ipvt=2* and a pivot smaller than RPARAM(1) in absolute value, is encountered in the factorization, it is replaced by a predetermined value *pvt_new* = RPARAM(2), and the factorization is continued. Default: *pvt_new = 0*.
On exit, **iparam(9)** is unchanged.

iparam(10): ipvt_loc
On entry, an unspecified variable.
On exit, **iparam(10)** contains the location of the first pivot element smaller in absolute value than *pvt_sml*. The pivot element is returned in *pvt_val* = RPARAM(3). If **iparam(10)**= 0, then no such pivot element exits.

rparam
real*8
An array of length at least 100, containing the real parameters for the simple driver.

rparam(1): pvt_sml
On entry, defines the value of the pivot element which is considered to be small. If a pivot element smaller than *pvt_sml* in absolute value is encountered in the factorization process, then, depending on the value of *ipvt* = IPARAM(9), the process either stops, continues or continues after the pivot is set equal to *pvt_new* = RPARAM(2). *pvt_sml* > 0. Recommended value: $10^{-15} \leq pvt\_sml \leq 1$. Default: $pvt\_sml = 10^{-12}$.
On exit, **rparam(1)** is unchanged.

rparam(2): pvt_new
On entry, defines the value to which the pivot element must be set if *ipvt* = 2 and the pivot element is less than *pvt_sml* in absolute value. *pvt_sml* should be large enough to avoid overflow when calculating the reciprocal of the pivot element.
On exit, **rparam(2)** is unchanged.

rparam(3): pvt_val
On entry, an unspecified variable.
On exit, *rparam(3)* contains the value of the first pivot element smaller than *pvt_sml* in absolute value. This element occurs at the location returned in IPARAM(10). If no such pivot element is found, the value of *pvt_val* is unspecified.

iwrk
integer*4
On entry, an array of length at least $2n$ used for integer workspace.
On exit, the first $2n$ elements of **iwrk** contain information generated by the factorization routine DSSKYF. This information is required by routines that use the factorization such as DSSKYS, DSSKYC, DSSKYR and remain unchanged between the call to DSSKYD and any subsequent calls to one of these routines.

rwrk
real*8
On entry, an array used for real workspace.
On exit, **rwrk** is unchanged. Presently, **rwrk** is not used by the routine DSSKYD. It can be a dummy variable.

ierror
integer*4
On entry, an unspecified variable.
On exit, **ierror** contains the error flag. A value of zero indicates a normal exit from the routine DSSKYD.

Description

DSSKYD is a simple driver routine that factors and solves the system:

$$AX = B$$

where A is a symmetric matrix stored in a skyline form, using either the profile-in storage mode or the diagonal-out storage mode; B is a matrix of nbx right sides and x is the matrix of the corresponding nbx solution vectors. On entry to the routine DSSKYD, the array BX contains the *nbx* right sides; on exit, these are overwritten by the solution vectors.

The matrix A is first factorized as:

$$A = U^T DU$$

by a call to the routine DSSKYF. U is a unit upper triangular matrix and D is a diagonal matrix. The routine DSSKYF does not perform any pivoting to preserve the numerical stability of the $U^T DU$ factorization. It is therefore primarily intended for the solution of symmetric positive (or negative) definite systems as they do not require pivoting for numerical stability. Caution is urged when using this routine for symmetric indefinite systems.

If a small pivot, in absolute value, *pvt_sml*, is encountered in the process of factorization, you have the option of either stopping the factorization process and returning to the calling subprogram, continuing the factorization process with the small value of the pivot, or continuing after setting the pivot equal to some predetermined value, *pvt_new*. The location of the first occurrence of a small pivot is returned in *ipvt_loc* and its value in *pvt_val*.

After the factorization has been obtained without any error, the routine DSSKYD calls the solve routine, DSSKYS, to solve the system. The call to the routine DSSKYD can be followed by a call to the routines DSSKYS, DSSKYC and DSSKYR, provided the first $2n$ elements of the work array IWRK remain unchanged between calls. The real workspace array, RWRK, is not used at present.

DSSKYX
Symmetric Sparse Expert Driver Using Skyline Storage Scheme

Format

DSSKYX (n, au, auf, iaudiag, nau, b, ldb, x, ldx, ferr, berr, nbx, iparam, rparam, iwrk, rwrk, ierror)

Arguments

n
integer*4
On entry, the order of the matrix A.
On exit, **n** is unchanged.

au
real*8
On entry, an array of length at least *nau*, containing the matrix A stored in the skyline storage scheme, using either the profile-in or the diagonal-out storage mode.
On exit, **au** is unchanged.

auf
real*8
On entry, if RPARAM(9) = *ifactor = 0*, **auf** is an unspecified array of length at least *nau*. If *ifactor = 1*, **auf** is an array of length at least *nau*, containing the $U^T DU$ factorization of the matrix A stored in the skyline storage scheme, using either the profile-in or the diagonal-out storage mode. The factorization has been obtained by a prior call to the routine DSSKYF.
On exit, if *ifactor = 0*, **auf** contains the $U^T DU$ factorization of the matrix A stored in the skyline storage scheme, using either the profile-in or the diagonal-out storage mode. If *ifactor = 1*, then **auf** is unchanged.

iaudiag
integer*4
On entry, an array of length at least n for the profile-in storage mode and $(n + 1)$ for the diagonal-out storage mode, containing the pointers to the locations of the diagonal elements in arrays AU and AUF (if *ifactor =1*).
On exit, **iaudiag** is unchanged.

nau
integer*4
On entry, the number of elements in array AU. *nau* is also the envelope size of the symmetric part of the matrix A. For the profile-in storage mode, *nau* = IAUDIAG(n). For the diagonal-out storage mode, *nau* = IAUDIAG(n+*1*) - 1.
On exit, **nau** is unchanged.

b
real*8
On entry, a two-dimensional array B of order *ldb* by at least *nbx*, containing the *nbx* right sides.
On exit, **b** is unchanged.

ldb
integer*4
On entry, the leading dimension of array B. $ldb \geq n$.
On exit, **ldb** is unchanged.

x
real*8
On entry, a two-dimensional array X of order ldx by at least nbx.
On exit, **x** contains the solutions obtained after iterative refinement.

ldx
integer*4
On entry, the leading dimension of array X. $ldx \geq n$.
On exit, **ldx** is unchanged.

ferr
real*8
On entry, an array FERR of length at least nbx, whose elements are unspecified variables.
On exit, **ferr** contains the estimated error bounds for each of the nbx solution vectors.

berr
real*8
On entry, an array BERR of length at least nbx, whose elements are unspecified variables.
On exit, **berr** contains the component-wise relative backward error for each of the nbx solution vectors.

nbx
integer*4
On entry, the number of right sides.
On exit, **nbx** is unchanged.

iparam
integer*4
An array of length at least 100, containing the integer parameters for the expert driver.

iparam(1): niparam
On entry, defines the length of the array IPARAM. $niparam \geq 100$.
On exit, **iparam(1)** is unchanged.

iparam(2): nrparam
On entry, defines the length of the array RPARAM. $nrparam \geq 100$.
On exit, **iparam(2)** is unchanged.

iparam(3): niwrk
On entry, defines the size of the integer work array, IWRK. $niwrk \geq 3n$.
On exit, **iparam(3)** is unchanged.

iparam(4): nrwrk
On entry, defines the size of the real work array, RWRK. $nrwrk \geq 3n$.
On exit, **iparam(4)** is unchanged.

iparam(5): iounit
On entry, defines the I/O unit number for printing error messages and information from the routine DSSKYX. The I/O unit must be opened in the calling subprogram. If *iounit* $\leq$0, no output is generated.
On exit, **iparam(5)** is unchanged.

iparam(6): iolevel
On entry, defines the message level that determines the amount of information printed out to *iounit*, when *iounit* > 0:

iolevel = 0 : fatal error messages only
iolevel = 1 : error messages and minimal information
iolevel = 2 : error messages and detailed information

On exit, **iparam(6)** is unchanged.

iparam(7): idefault
On entry, defines if the default values should be used in arrays IPARAM and RPARAM. If *idefault* = 0, then the following default values are assigned:

IPARAM(1) = *niparam* = 100
IPARAM(2) = *nrparam* = 100
IPARAM(6) = *iolevel* = 0
IPARAM(8) = *istore* = 1
IPARAM(9) = *ifactor* = 0
IPARAM(10) = *idet* = 0
IPARAM(11) = *ipvt* = 0
IPARAM(13) = *inertia* = 0
IPARAM(17) = *itmax* = 5
RPARAM(1) = *pvt_sml* = 10^{-12}

If *idefault* = 1, then you must assign values to the above variables before the call to the DSSKYX routine.
On exit, **iparam(7)** is unchanged.

iparam(8): istore
On entry, defines the type of storage scheme used for the skyline matrix. If *istore* = 1, the matrix A is stored using the profile-in storage mode; if *istore* = 2, the matrix A is stored using the diagonal-out storage mode. Default: *istore* = 1.
On exit, **iparam(8)** is unchanged.

iparam(9): ifactor
On entry, defines if the matrix A has already been factored on input to the routine DSSKYX. If *ifactor = 0*, the matrix is unfactored and array AUF is unspecified. If *ifactor = 1*, the matrix has been factored by a prior call to the routine DSSKYF, and array AUF contains the $U^T DU$ factorization of A. Default: *ifactor = 0*.
On exit, **iparam(9)** is unchanged.

iparam(10): idet
On entry, defines if the determinant of the matrix A is to be calculated. If *idet* = 0, then the determinant is not calculated; if *idet* = 1, the determinant is calculated as *det_base* * $10^{\text{det\_pwr}}$. See RPARAM(4) and RPARAM(5). Default: *idet* = 0.
On exit, **iparam(10)** is unchanged.

iparam(11): ipvt
On entry, defines if the factorization should continue when a small pivot, defined by RPARAM(1), is encountered. If *ipvt* = 0 and the absolute value of the pivot element is smaller than *pvt_sml* = RPARAM(1), then the factorization process is stopped and control returned to the calling subprogram. If *ipvt* = 1 and a pivot smaller than RPARAM(1) in absolute value is encountered in the factorization, the process continues. If *ipvt* = 2 and a pivot smaller than RPARAM(1) in absolute value is encountered in the factorization, it is replaced by a predetermined value *pvt_new* = RPARAM(2), and the factorization is continued. Default: *ipvt* = 0.
On exit, **iparam(11)** is unchanged.

iparam(12): ipvt_loc
On entry, an unspecified variable.
On exit, **iparam(12)** contains the location of the first pivot element smaller in absolute value than *pvt_sml*. The pivot element is returned in *pvt_val* = RPARAM(3). If **iparam(12)** = 0, then no such pivot element exists. If *ifactor = 1*, then **iparam (12)** is unspecified.

iparam(13): inertia
On entry, defines if the inertia of the matrix A should be calculated during factorization. The inertia of the symmetric matrix A is the triplet of integers (*ipeigen, ineigen, izeigen*), consisting of the number of positive, negative and zero eigenvalues, respectively. If *inertia* = 0, then the inertia is not calculated; if *inertia* = 1, then the number of positive and negative eigenvalues are returned in *ipeigen* = IPARAM(14) and *ineigen* = IPARAM(15), respectively. An indication of the existence of zero eigenvalues is returned in *izeigen* = IPARAM(16). Default: *inertia* = 0.
On exit, **iparam(13)** is unchanged.

iparam(14): ipeigen
On entry, an unspecified variable.
On exit, if *inertia* = 1, **iparam(14)** contains the number of positive eigenvalues of the matrix A. If *ifactor* = 0, then **iparam(14)** is unspecified.

iparam(15): ineigen
On entry, an unspecified variable.
On exit, if *inertia* = 1, **iparam(15)** contains the number of negative eigenvalues of the matrix A. If *ifactor* = 0, then **iparam(15)** is unspecified.

iparam(16): izeigen
On entry, an unspecified variable.
On exit, if *inertia* = 1, **iparam(16)** indicates if the matrix A has any zero eigenvalues. If *izeigen* = 0, then the matrix A does not have a zero eigenvalue; if *izeigen* = 1, then the matrix A has at least one zero eigenvalue. If *ifactor* = 0, the **iparam(16)** is unspecified.

iparam(17): itmax
On entry, defines the maximum number of iterations for the iterative refinement process. Default: $itmax = 5$.
On exit, **iparam(17)** is unchanged.

rparam
real*8
An array of length at least 100, containing the real parameters for the expert driver.

rparam(1): pvt_sml
On entry, defines the value of the pivot element which is considered to be small. If a pivot element smaller than *pvt_sml*, in absolute value, is encountered in the factorization process, then, depending on the value of *ipvt* = IPARAM(11), the process either stops, continues or continues after the pivot is set equal to *pvt_new* = RPARAM(2). *pvt_sml* > 0. Recommended value: $10^{-15} \leq$ *pvt_sml* ≤ 1. Default: *pvt_sml* = 10^{-12}.
On exit, **rparam(1)** is unchanged.

rparam(2): pvt_new
On entry, defines the value to which the pivot element must be set if *ipvt* = 2 and the pivot element is less than *pvt_sml* in absolute value. *pvt_new* should be large enough to avoid overflow when calculating the reciprocal of the pivot element. If *ifactor = 1*, then **rparam(2)** is unspecified.
On exit, **rparam(2)** is unchanged.

rparam(3): pvt_val
On entry, an unspecified variable.
On exit, **rparam(3)** contains the value of the first pivot element smaller than *pvt_sml* in absolute value. This element occurs at the location returned in IPARAM(12). If no such pivot element is found, the value of *pvt_val* is unspecified. If *ifactor = 1*, then **rparam(3)** is unspecified.

rparam(4): det_base
On entry, an unspecified variable.
On exit, defines the base for the determinant of the matrix ***A***. If *idet* = 1, the determinant is calculated as *det_base* * $10^{\text{det\_pwr}}$. If *ifactor = 1*, then **rparam(4)** is unspecified. $1.0 \leq$ *det_base* ≤ 10.0).

rparam(5): det_pwr
On entry, an unspecified variable.
On exit, defines the power for the determinant of the matrix ***A***. If *idet* = 1, the determinant is calculated as *det_base* * $10^{\text{det\_pwr}}$. If *ifactor = 1*, then **rparam(5)** is unspecified.

rparam(6): anorm
On entry, an unspecified variable.
On exit, **rparam(6)** contains the 1-norm of the matrix ***A***.

rparam(7): ainorm
On entry, an unspecified variable.
On exit, **rparam(7)** contains the estimate of the 1-norm of $\boldsymbol{A}^{-1}$.

rparam(8): rcond
On entry, an unspecified variable.
On exit, **rparam(8)** contains the reciprocal of the estimate of the 1-norm condition number of the matrix ***A***.

iwrk
integer*4
On entry, an array of length at least $3n$ used for integer workspace. If $ifactor = 1$, then the first $2n$ elements of the array IWRK contain information generated by the routine DSSKYF. If $ifactor = 0$, then this information is unspecified.
On exit, the first $2n$ elements of the array IWRK contain information generated by the routine DSSKYF. This information is used by the routines DSSKYS and

DSSKYR, and should therefore remain unchanged between the call to the routine DSSKYX and any subsequent call to the routines DSSKYS and DSSKYR.

rwrk
real*8
On entry, an array of length at least $3n$ used for real workspace.
On exit, the first $3n$ elements of **rwrk** are overwritten.

ierror
integer*4
On entry, an unspecified variable.
On exit, **ierror** contains the error flag. A value of zero indicates a normal exit from the routine DSSKYX.

Description

DSSKYX is an expert driver routine that:

- Obtains the $U^T DU$ factorization of the matrix A via a call to the routine DSSKYF.
- If the factorization is successful, obtains the 1-norm condition number estimate of the matix A by a call to the routine DSSKYC.
- If the reciprocal of the condition number estimate is greater than the machine precision, DSSKYX uses the factorization to solve the system:

  $$AX = B$$

  using the routine DSSKYS.
- Improves the solution X via iterative refinement and obtains the error bounds using the routine DSSKYR.

DSSKYX first obtains the factorization of the symmetric matrix A as:

$$A = U^T DU$$

where D is a diagonal matrix, and U is a unit upper triangular matrix. The matrix A is stored in a skyline form, using either the profile-in storage mode or the diagonal-out storage mode. If the matrix is already factored, as indicated by *ifactor*, then this step is skipped.

The routine DSSKYF does not perform any pivoting to preserve the numerical stability of the $U^T DU$ factorization. It is therefore primarily intended for the solution of symmetric positive (or negative) definite systems as they do not require pivoting for numerical stability. Caution is urged when using this routine for symmetric indefinite systems.

If a small pivot, in absolute value, *pvt_sml*, is encountered in the process of factorization, you have the option of either stopping the factorization process and returning to the calling subprogram, continuing the factorization process with the small value of the pivot, or continuing after setting the pivot equal to some predetermined value, *pvt_new*. The location of the first occurrence of a small pivot is returned in *ipvt_loc* and its value in *pvt_val*.

In addition to the $U^T DU$ factorization, you can also obtain the determinant of A, the number of positive and negative eigenvalues of the matrix A, and an indication of the existence of zero eigenvalues. If the factorization process is stopped at row i due to a small pivot, then the inertia and determinant are evaluated for rows 1 through $(i-1)$.

The routine DSSKYX does not allow a partial factorization of the matrix A. If a partial factorization of A is required, the routine DSSKYF is recommended.

DSSKYC obtains the reciprocal of the estimate of the condition number of the symmetric matrix A as:

$$rcond(A) = \frac{1}{||A|| \cdot ||A^{-1}||}$$

The 1-norm of A^{-1} is obtained using the LAPACK routine DLACON, which uses Higham's modification [Higham 1988] of Hager's method [Hager 1984]. If the reciprocal of the condition number estimate is larger than the machine precision, the routine DSSKYX solves the system via a call to the routine DSSKYS and then improves on the solution via iterative refinement. This is done by calculating the matrix of residuals R using the matrix of solutions $\hat{X}$ obtained from DSSKYS, and obtaining a new matrix of solutions X_{new} as follows:

$$R = B - A\hat{X}$$

$$\delta X = A^{-1} R$$

and:

$$X_{new} = \hat{X} + \delta X$$

In addition to the iterative refinement of the solution vectors, the routine DSSKYX also provides the component-wise relative backward error, $berr$ and the estimated forward error bound, $ferr$, for each solution vector [Arioli, Demmel, Duff 1989, Anderson et. al. 1992]. $berr$ is the smallest relative change in any entry of A or b that makes $\hat{x}$ an exact solution. $ferr$ bounds the magnitude of the largest entry in $\hat{x} - x_{true}$ divided by the magnitude of the largest entry in $\hat{x}$.

The process of iterative refinement is continued as long as all of the following conditions are satisfied [Arioli, Demmel, Duff 1989]:

- The number of iterations of the iterative refinement process is less than IPARAM(10) = $itmax$.
- $berr$ reduces by at least a factor of 2 during the previous iteration.
- $berr$ is larger than the machine precision.

The first $4n$ elements of the integer workspace array IWRK, generated by DSSKYF, contain information for use by the routines DSSKYS and DSSKYR. They must therefore remain unchanged between the calls to the routine DSSKYX and any subsequent calls to the routines DSSKYS and DSSKYR.

DUSKYN
Unsymmetric Sparse Matrix Norm Evaluation Using Skyline Storage Scheme

Format

DUSKYN (n, au, iaudiag, nau, al, ialdiag, nal, iparam, rparam, iwrk, rwrk, ierror)

Arguments

n
integer*4
On entry, the order of the matrix *A*.
On exit, **n** is unchanged.

au
real*8
On entry, an array containing information on the matrix *A*. If *istore* = 1 or 2, then **au** contains the upper triangular part, including the diagonal, of the matrix *A*, stored in the profile-in or diagonal-out mode, respectively. Array AU is of length at least *nau*, where *nau* is the envelope size of the upper triangular part of *A*, including the diagonal. If *istore = 3*, then *au* contains the matrix *A*, stored in the structurally symmetric, profile-in storage mode. In this case, array AU is of length at least *nau*, where *nau* is the envelope size of the matrix *A*.
On exit, *au* is unchanged.

iaudiag
integer*4
On entry, an array containing the pointers to the locations of the diagonal elements in array AU. **iaudiag** is of length at least n for the profile-in and the structurally symmetric profile-in storage modes. **iaudiag** is of length at least $(n+1)$ for the diagonal-out storage mode.
On exit, **iaudiag** is unchanged.

nau
integer*4
On entry, the number of elements stored in array AU. If *istore* = 1 or 2, then *nau* is the envelope size of the upper triangular part of the matrix *A*. If *istore = 3*, then *nau* is the envelope size of the matrix *A*. For the profile-in and the structurally symmetric profile-in storage modes, *nau* = IAUDIAG(*n*). For the diagonal-out storage mode, *nau* = IAUDIAG(*n+1*) - 1.
On exit, **nau** is unchanged.

al
real*8
On entry, an array containing information on the matrix *A*. If *istore* = 1 or 2, then **al** contains the lower triangular part, including the diagonal, of the matrix *A*, stored in the profile-in or diagonal-out mode, respectively. Storage is allocated for the diagonal elements, though the elements themselves are not stored. Array AL is of length at least *nal*, where *nal* is the envelope size of the lower triangular part of *A*, including the diagonal. If *istore = 3*, then **al** is a dummy argument.
On exit, **al** is unchanged.

ialdiag
integer*4
On entry, an array containing the pointers to the locations of the diagonal elements in array AL. **ialdiag** is of length at least n for the profile-in storage mode. **ialdiag** is of length at least (n+1) for the diagonal-out storage mode. If *istore* = *3*, then **ialdiag** is a dummy argument.
On exit, **ialdiag** is unchanged.

nal
integer*4
On entry, the number of elements stored in array AL. If *istore* = 1 or 2, then nal is the envelope size of the lower triangular part of the matrix $\boldsymbol{A}$. For the profile-in storage mode, nal = IALDIAG(n). For the diagonal-out storage mode, nal = IALDIAG(n+1) - 1. If *istore* = *3*, then **nal** is a dummy argument.
On exit, **nal** is unchanged.

iparam
integer*4
An array of length at least 100, containing the integer parameters for the norm evaluation.

iparam(1): niparam
On entry, defines the length of the array IPARAM. $niparam \geq 100$.
On exit, **iparam(1)** is unchanged.

iparam(2): nrparam
On entry, defines the length of the array RPARAM. $nrparam \geq 100$.
On exit, **iparam(2)** is unchanged.

iparam(3): niwrk
On entry, defines the size of the integer work array, IWRK. $niwrk \geq n$.
On exit, **iparam(3)** is unchanged.

iparam(4): nrwrk
On entry, defines the size of the real work array, RWRK. As the real work array is not used at present, $nrwrk$ can be unspecified.
On exit, **iparam(4)** is unchanged.

iparam(5): iounit
On entry, defines the I/O unit number for printing error messages and information from the routine DUSKYN. The I/O unit must be opened in the calling subprogram. If $iounit \leq 0$, no output is generated.
On exit, **iparam(5)** is unchanged.

iparam(6): iolevel
On entry, defines the message level that determines the amount of information printed out to *iounit*, when *iounit* > 0:

iolevel = 0 : fatal error messages only
iolevel = 1 : error messages and minimal information
iolevel = 2 : error messages and detailed information

On exit, **iparam(6)** is unchanged.

iparam(7): idefault
On entry, defines if the default values should be used in arrays IPARAM and RPARAM. If *idefault* = 0, then the following default values are assigned:

IPARAM(1) = *niparam* = 100
IPARAM(2) = *nrparam* = 100
IPARAM(6) = *iolevel* = 0
IPARAM(8) = *istore* = 1
IPARAM(9) = *inorm* = 1

If *idefault* = 1, then you must assign values to the above variables before the call to the DUSKYN routine.
On exit, **iparam(7)** is unchanged.

iparam(8): istore
On entry, defines the type of storage scheme used for the skyline matrix. If *istore* = 1, the unsymmetric matrix A is stored using the profile-in storage mode; if *istore* = 2, the unsymmetric matrix A is stored using the diagonal-out storage mode. if *istore* = 3, the unsymmetric matrix A is stored using the structurally symmetric profile-in storage mode. Default: *istore* = 1.
On exit, **iparam(8)** is unchanged.

iparam(9): inorm
On entry, defines if the matrix quantity to be evaluated:

inorm = 1 : 1-norm of A
inorm = 2 : ∞-norm of A
inorm = 3 : Frobenius norm of A
inorm = 4 : Maximum absolute value of A

Default: *inorm = 1*
On exit, **iparam(9)** is unchanged.

rparam
real*8
An array of length at least 100, containing the real parameters for the norm evaluation.

rparam(1): anorm
On entry, is an unspecified variable.
On exit, **rparam(1)** contains the matrix quantity evaluated, as defined by the value of IPARAM(9) = *inorm*.

iwrk
integer*4
On entry, an array of length at least n used for integer workspace.
On exit, **iwrk** contains information for use by the routine DUSKYN. This information is not used by any other routine and can therefore be overwritten.

rwrk
real*8
On entry, an array of length at least n used for real workspace.
On exit, **rwrk** contains information used by the solver routine DUSKYN. This information is not used by any other routine and can therefore be overwritten.

ierror
integer*4
On entry, an unspecified variable.
On exit, **ierror** contains the error flag. A value of zero indicates a normal exit from the routine DUSKYN.

Description

DUSKYN evaluates the following quantities for the unsymmetric matrix A:

- 1-norm of A:
$$||A||_1 = \max_j \sum_i |a_{ij}|$$
- ∞-norm of A:
$$||A||_\infty = \max_i \sum_j |a_{ij}|$$
- Frobenius-norm of A:
$$||A||_F = \sqrt{\sum_i \sum_j |a_{ij}|^2}$$
- Largest absolute value of A:
$$\max_{i,j} |a_{ij}|$$

The last quantity in the above list is not a matrix norm. The quantity evaluated is determined by the value of IPARAM(9) = *inorm*.

The real and integer workspace used by the routine DUSKYN does not contain information for use by any other routines, and can therefore be overwritten.

As the routine DUSKYN requires the matrix A, it should be called prior to a call to the factorization routine DUSKYF, which overwrites the elements of A by the LDU factors.

DUSKYF
Unsymmetric Sparse Matrix Factorization Using Skyline Storage Scheme (Serial and Parallel Versions)

Format

DUSKYF (n, au, iaudiag, nau, al, ialdiag, nal, iparam, rparam, iwrk, rwrk, ierror)

Arguments

n
integer*4
On entry, the order of the matrix *A*.
On exit, **n** is unchanged.

au
real*8
On entry, an array containing information on the matrix *A*. If *istore* = 1 or 2, then **au** contains the upper triangular part, including the diagonal, of the matrix *A*, stored in the profile-in or diagonal-out mode, respectively. Array AU is of length at least *nau*, where *nau* is the envelope size of the upper triangular part of *A*, including the diagonal. If *istore* = *3*, then **au** contains the matrix *A*, stored in the structurally symmetric, profile-in storage mode. In this case, array AU is of length at least *nau*, where *nau* is the envelope size of the matrix *A*.
On exit, if *istore* = 1 or 2, **au** contains the factors *U* and *D* of the *LDU* factorization of the matrix *A*. If *istore* = *3*, then **au** contains the factors *L*, *U* and *D* of the *LDU* factorization of the matrix *A*. **au** must remain unchanged between the call to the routine DUSKYF and any routines that use the factors such as DUSKYS, DUSKYC, DUSKYR, and DUSKYX.

iaudiag
integer*4
On entry, an array containing the pointers to the locations of the diagonal elements in array AU. **iaudiag** is of length at least *n* for the profile-in and the structurally symmetric profile-in storage modes. **iaudiag** is of length at least $(n+1)$ for the diagonal-out storage mode.
On exit, **iaudiag** is unchanged.

nau
integer*4
On entry, the number of elements stored in array AU. If *istore* = 1 or 2, then *nau* is the envelope size of the upper triangular part of the matrix *A*. If *istore* = *3*, then *nau* is the envelope size of the matrix *A*. For the profile-in and the structurally symmetric profile-in storage modes, *nau* = IAUDIAG(*n*). For the diagonal-out storage mode, *nau* = IAUDIAG(*n+1*) - 1.
On exit, **nau** is unchanged.

al
real*8
On entry, an array containing information on the matrix *A*. If *istore* = 1 or 2, then **al** contains the lower triangular part, including the diagonal, of the matrix *A*, stored in the profile-in or diagonal-out mode, respectively. Storage is allocated for the diagonal elements, though the elements themselves are not stored. Array

AL is of length at least *nal*, where *nal* is the envelope size of the lower triangular part of *A*, including the diagonal. If *istore* = *3*, then **al** is a dummy argument. On exit, if *istore* = 1 or 2, **al** contains the factor *L* of the *LDU* factorization of the matrix *A*. If *istore* = *3*, then **al** is undefined. **al** must remain unchanged between the call to the routine DUSKYF and any routines that use the factors such as DUSKYS, DUSKYC, DUSKYR, and DUSKYX.

ialdiag
integer*4
On entry, an array containing the pointers to the locations of the diagonal elements in array AL. **ialdiag** is of length at least *n* for the profile-in storage mode. **ialdiag** is of length at least (*n*+1) for the diagonal-out storage mode. If *istore* = *3*, then **ialdiag** is a dummy argument.
On exit, **ialdiag** is unchanged.

nal
integer*4
On entry, the number of elements stored in array AL. If *istore* = 1 or 2, then *nal* is the envelope size of the lower triangular part of the matrix *A*. For the profile-in storage mode, *nal* = IALDIAG(*n*). For the diagonal-out storage mode, *nal* = IALDIAG(*n* + 1) - 1. If *istore* = *3*, then **nal** is a dummy argument.
On exit, **nal** is unchanged.

iparam
integer*4
An array of length at least 100, containing the integer parameters for the *LDU* factorization.

iparam(1): niparam
On entry, defines the length of the array IPARAM. niparam $\geq$ 100.
On exit, **iparam(1)** is unchanged.

iparam(2): nrparam
On entry, defines the length of the array IPARAM. nrparam $\geq$ 100.
On exit, **iparam(2)** is unchanged.

iparam(3): niwrk
On entry, defines the size of the integer work array, IWRK. *niwrk*$\geq 4n$.
On exit, **iparam(3)** is unchanged.

iparam(4): nrwrk
On entry, defines the size of the real work array, RWRK. As the real work array is not used at present, *nrwrk* may be unspecified.
On exit, **iparam(4)** is unchanged.

iparam(5): iounit
On entry, defines the I/O unit number for printing error messages and information from the routine DUSKYF. The I/O unit must be opened in the calling subprogram. If *iounit* $\leq$ *0*, then no output is generated.
On exit, **iparam(5)** is unchanged.

iparam(6): iolevel
On entry, defines the message level that determines the amount of information printed out to *iounit*, when *iounit* > *0*:

- *iolevel = 0 :* fatal error messages only

- *iolevel = 1 :* fatal errors, warnings and minimal information
- *iolevel = 2 :* detailed information and statistics

On exit, **iparam(6)** is unchanged.

iparam(7): idefault
On entry, defines if the default values should be used in arrays IPARAM and RPARAM. If *idefault = 0*, then the following default values are assigned:

IPARAM(1) = *niparam = 100*
IPARAM(2) = *nrparam = 100*
IPARAM(6) = *iolevel = 0*
IPARAM(8) = *istore = 1*
IPARAM(9) = *ibeg = 0*
IPARAM(10) = *idet = 0*
IPARAM(11) = *ipvt = 0*
RPARAM(1) = *pvt_sml* = 10^{-12}

If *idefault = 1*, then you must assign values to the above variables before the call to the DUSKYF routine.
On exit, **iparam(7)** is unchanged.

iparam(8): istore
On entry, defines the type of storage scheme used for the skyline matrix. If *istore = 1*, the unsymmetric matrix ***A*** is stored using the profile-in storage mode; if *istore = 2*, the unsymmetric matrix ***A*** is stored using the diagonal-out storage mode; if *istore = 3*, the unsymmetric matrix ***A*** is stored using the structurally symmetric profile-in storage mode. Default: *istore = 1*.
On exit, **iparam(8)** is unchanged.

iparam(9): ibeg
On entry, defines if full or partial factorization is to be performed. If *ibeg = 0*, then a full factorization is performed for rows and columns 1 through n. If *ibeg > 0*, then a partial factorization is performed starting from rows and columns $ibeg + 1$ through n, that is, rows and columns from 1 through $ibeg$ have already been factorized. Default: *ibeg = 0*.
On exit, **iparam(9)** is unchanged.

iparam(10): idet
On entry, defines if the determinant of the matrix ***A*** is to be calculated. If *idet = 0*, then the determinant is not calculated; if *idet = 1*, the determinant is calculated as *det_base * 10* $^{\text{det\_pwr}}$. See RPARAM(4) and RPARAM(5). Default: *idet = 0*.
On exit, **iparam(10)** is unchanged.

iparam(11): ipvt
On entry, defines if the factorization should continue when a small pivot, defined by RPARAM(1), is encountered. If *ipvt = 0* and the absolute value of the pivot element is smaller than *pvt_sml* = RPARAM(1), then the factorization process is stopped and control returned to the calling subprogram. If *ipvt = 1* and a pivot smaller than RPARAM(1) in absolute value is encountered in the factorization, the process continues. If *ipvt = 2* and a pivot smaller than RPARAM(1) in absolute value, is encountered in the factorization, it is replaced by a predetermined value *pvt_new* = RPARAM(2), and the factorization is continued. Default: *ipvt = 0*.
On exit, **iparam(11)** is unchanged.

iparam(12): ipvt_loc
On entry, an unspecified variable.
On exit, **iparam(12)** contains the location of the first pivot element, smaller in absolute value than *pvt_sml*. The pivot element is returned in *pvt_val* = RPARAM(3). If **iparam(12)** = 0, then no such pivot element exists.

rparam
real*8
An array of length at least 100, containing the real parameters for the *LDU* factorization.

rparam(1): pvt_sml
On entry, defines the value of the pivot element which is considered to be small. If a pivot element smaller than *pvt_sml* in absolute value is encountered in the factorization process, then, depending on the value of *ipvt* = IPARAM(11), the process either stops, continues or continues after the pivot is set equal to *pvt_new* = RPARAM(2). *pvt_sml > 0*. Recommended value: $10^{-15} \leq$ *pvt_sml* = ≤ 1. Default: *pvt_sml* = 10^{-12}.
On exit, **rparam(1)** is unchanged.

rparam(2): pvt_new
On entry, defines the value to which the pivot element must be set if *ipvt = 2* and the pivot element is less than *pvt_sml*. *pvt_new* must be large enough to avoid overflow when calculating the reciprocal of the pivot element.
On exit, **rparam(2)** is unchanged.

rparam(3): pvt_val
On entry, an unspecified variable.
On exit, **rparam(3)** contains the value of the first pivot element smaller than *pvt_sml* in absolute value. The location of this element is returned in *ipvt_loc* = IPARAM(12).

rparam(4): det_base
On entry, an unspecified variable.
On exit, defines the base for the determinant of the matrix ***A***. If *idet = 1*, the determinant is calculated as *det_base* * $10^{\text{det\_pwr}}$. $1.0 \leq$ *det_base* < *10.0*.

rparam(5): det_pwr
On entry, an unspecified variable.
On exit, defines the power for the determinant of the matrix ***A***. If *idet = 1*, the determinant is calculated as *det_base* * $10^{\text{det\_pwr}}$.

iwrk
integer*4
On entry, an array of length at least $4n$ used for integer workspace.
On exit, **iwrk** contains information for use by routines that use the factorization such as DUSKYS, DUSKYC, DUSKYR and DUSKYX. The first $4n$ elements of **iwrk** should therefore be passed unchanged to these routines.

rwrk
real*8
On entry, an array used for real workspace.
On exit, **rwrk** is unchanged. **rwrk** is not used by the routine DUSKYF at present but is reserved for future use. It can be a dummy variable.

ierror
integer*4
On entry, an unspecified variable.
On exit, **ierror** contains the error flag. A value of zero indicates a normal exit from the routine DUSKYF.

Description

DUSKYF obtains the factorization of the unsymmetric matrix A as:

$$A = LDU$$

where L is a unit lower triangular matrix, D is a diagonal matrix, and U is a unit upper triangular matrix. The matrix A is stored in a skyline form, using either the profile-in storage mode, the diagonal-out storage mode, or the structurally symmetric profile-in storage mode.

The routine DUSKYF does not perform any pivoting to preserve the numerical stability of the LDU factorization. It is therefore primarily intended for the solution of systems that do not require pivoting for numerical stability, such as diagonally dominant systems. Caution is urged when using this routine for problems that require pivoting.

If a small pivot in absolute value, *pvt_sml*, is encountered in the process of factorization, you have the option of either stopping the factorization process and returning to the calling program, continuing the factorization process, or continuing after setting the pivot equal to some predetermined value, *pvt_new*. The location of the first occurrence of a small pivot is returned in *ipvt_loc* and its value in *pvt_val*.

In addition to the LDU factorization, the routine DUSKYF can be used to obtain the determinant of A. A partial factorization can also be obtained by appropriately setting the value of *ibeg*. If $ibeg > 0$, then factorization begins at row and column $ibeg + 1$; the rows and columns from 1 to *ibeg* are assumed to have been already factorized. When $ibeg > 0$, the determinant of A and the statistics on the matrix are calculated from rows and columns *ibeg + 1* through n. If the factorization process is stopped at row i due to a small pivot, then the determinant and the statistics on the matrix are evaluated for rows $ibeg + 1$ through $(i - 1)$.

The data in the first $4n$ elements of the integer workspace array, IWRK, are used in routines that use the LDU factorization, such as DUSKYS, DUSKYC, and DUSKYR. This data must therefore remain unchanged between the call to DUSKYF and any one of these routines. The real workspace array, RWRK, is not used at present.

This routine is available in both serial and parallel versions. The routine names and parameter list are identical for both versions. For information about using the parallel library, see Chapter 4. For information about linking to the serial or to the parallel library, see Chapter 5.

DUSKYS
Unsymmetric Sparse Matrix Solve Using Skyline Storage Scheme

Format

DUSKYS (n, au, iaudiag, nau, al, ialdiag, nal, bx, ldbx, nbx, iparam, rparam, iwrk, rwrk, ierror)

Arguments

n
integer*4
On entry, the order of the matrix *A*.
On exit, **n** is unchanged.

au
real*8
On entry, if *istore* = 1 or 2, **au** contains the factors *U* and *D* of the *LDU* factorization of the matrix *A*. Array AU is of length at least *nau*, where *nau* is the envelope size of the upper triangular part of *A*, including the diagonal. If *istore* = *3*, then **au** contains the *LDU* factorization of the matrix *A*. In this case, array AU is of length at least *nau*, where *nau* is the envelope size of the matrix *A*. The *LDU* factorization has been obtained by a prior call to the routine DUSKYF. **au** must remain unchanged between calls to the routines DUSKYF and DUSKYS.
On exit, **au** is unchanged.

iaudiag
integer*4
On entry, an array containing the pointers to the locations of the diagonal elements in array AU. **iaudiag** is of length at least n for the profile-in and the structurally symmetric profile-in storage modes. **iaudiag** is of length at least $(n+1)$ for the diagonal-out storage mode.
On exit, **iaudiag** is unchanged.

nau
integer*4
On entry, the number of elements stored in array AU. If *istore* = 1 or 2, then *nau* is the envelope size of the upper triangular part of the matrix *A*. If *istore* = *3*, then *nau* is the envelope size of the matrix *A*. For the profile-in and the structurally symmetric profile-in storage modes, *nau* = IAUDIAG(*n*). For the diagonal-out storage mode, *nau* = IAUDIAG(*n+1*) - 1.
On exit, **nau** is unchanged.

al
real*8
On entry, if *istore* = 1 or 2, **al** contains the factor *L* of the *LDU* factorization of the matrix *A*. Array AL is of length at least *nal*, where *nal* is the envelope size of the lower triangular part of *A*, including the diagonal. If *istore* = *3*, then **al** is a dummy argument. The *LDU* factorization is obtained from a prior call to the routine DUSKYF. **al** must remain unchanged between calls to the routines DUSKYF and DUSKYS.
On exit, **al** is unchanged.

ialdiag
integer*4
On entry, an array containing the pointers to the locations of the diagonal elements in array AL. **ialdiag** is of length at least n for the profile-in storage mode. **ialdiag** is of length at least $n + 1$ for the diagonal-out storage mode. If *istore = 3*, then **ialdiag** is a dummy argument.
On exit, **ialdiag** is unchanged.

nal
integer*4
On entry, the number of elements stored in array AL. If *istore = 1 or 2*, then nal is also the envelope size of the lower triangular part of the matrix A. For the profile-in storage mode, nal = IALDIAG(n). For the diagonal-out storage mode, nal = IALDIAG(n+1) - 1. If *istore = 3*, then nal is a dummy argument.
On exit, **nal** is unchanged.

bx
real*8
On entry, a two-dimensional array BX of order *ldbx* by at least *nbx*, containing the *nbx* right sides.
On exit, **bx** contains the solutions for the *nbx* systems.

ldbx
integer*4
On entry, the leading dimension of array BX. $ldbx \geq n$.
On exit, **ldbx** is unchanged.

nbx
integer*4
On entry, the number of right sides.
On exit, **nbx** is unchanged.

iparam
integer*4
An array of length at least 100, containing the integer parameters for the matrix solve operation.

iparam(1): niparam
On entry, defines the length of the array IPARAM. $niparam \geq 100$.
On exit, **iparam(1)** is unchanged.

iparam(2): nrparam
On entry, defines the length of the array RPARAM. As the real parameter array is not used at present, *nrparam* may be unspecified.
On exit, **iparam(2)** is unchanged.

iparam(3): niwrk
On entry, defines the size of the integer work array, IWRK. $niwrk \geq 4n$.
On exit, **iparam(3)** is unchanged.

iparam(4): nrwrk
On entry, defines the size of the real work array, RWRK. As the real work array is not used at present, *nrwrk* may be unspecified.
On exit, **iparam(4)** is unchanged.

iparam(5): iounit
On entry, defines the I/O unit number for printing error messages and information from the routine DUSKYF. The I/O unit must be opened in the calling subprogram. If *iounit* $\leq$0, no output is generated.
On exit, **iparam(5)** is unchanged.

iparam(6): iolevel
On entry, defines the message level that determines the amount of information printed out to *iounit*, when *iounit* > 0:

iolevel = 0 : fatal error messages only
iolevel = 1 : error messages and minimal information
iolevel = 2 : error messages and detailed information

On exit, **iparam(6)** is unchanged.

iparam(7): idefault
On entry, defines if the default values should be used in arrays IPARAM and RPARAM. If *idefault* = 0, then the following default values are assigned:

IPARAM(1) = *niparam* = 100
IPARAM(6) = *iolevel* = 0
IPARAM(8) = *istore* = 1
IPARAM(9) = *itrans* = 0

If *idefault* = 1, then you must assign values to the above variables before the call to the DUSKYF routine.
On exit, **iparam(7)** is unchanged.

iparam(8): istore
On entry, defines the type of storage scheme used for the skyline matrix. If *istore* = 1, the unsymmetric matrix A is stored using the profile-in storage mode; if *istore* = 2, the unsymmetric matrix A is stored using the diagonal-out storage mode; if *istore* = 3, the unsymmetric matrix A is stored using the structurally symmetric profile-in storage mode. The storage scheme used in the routines DUSKYF and DUSKYS must be identical. Default: *istore* = 1.
On exit, **iparam(8)** is unchanged.

iparam(9): itrans
On entry, defines the form of matrix used in the solution. If *itrans = 0*, the system solved is $AX = B$; if *itrans = 1*, the system solved is $A^T X = B$. Default: *itrans = 0*.
On exit, **iparam(9)** is unchanged.

rparam
real*8
On entry, an array containing the real parameters for the solution.
On exit, **rparam** is unchanged. **rparam** is not used by the routine DUSKYS at present, but is reserved for future use.

iwrk
integer*4
On entry, an array of length at least $4n$ used for integer workspace. The first $4n$ elements of the array IWRK, generated by the routine DUSKYF, should be passed unchanged to the routine DUSKYS.
On exit, the first $4n$ elements of **iwrk** are unchanged.

rwrk
real*8
On entry, an array used for real workspace.
On exit, **rwrk** is unchanged. Presently, **rwrk** is not used by the routine DUSKYS, but is reserved for future use. It can be a dummy variable.

ierror
integer*4
On entry, an unspecified variable.
On exit, **ierror** contains the error flag. A value of zero indicates a normal exit from the routine DUSKYS.

Description

DUSKYS solves the system:

$$AX = B$$

or:

$$A^T X = B$$

where A is an unsymmetric matrix stored in a skyline form, using either the profile-in storage mode, the diagonal-out storage mode, or the structurally symmetric profile-in storage mode; B is a matrix of nbx right sides and x is the matrix of the corresponding nbx solution vectors. On entry to the routine DUSKYS, the array BX contains the nbx right sides; on exit, these are overwritten by the solution vectors. The variable $itrans$ determines whether the matrix A or A^T is used in the solution process.

The matrix A has been factorized as:

$$A = LDU$$

by a prior call to the routine DUSKYF. L is a unit lower triangular matrix, U is a unit upper triangular matrix, and D is a diagonal matrix. The first $4n$ elements of the integer workspace array IWRK, generated by DUSKYF, contain information for use by DUSKYS and therefore must remain unchanged between the calls to the routines DUSKYF and DUSKYS. The real work array RWRK is not used at present. The storage scheme used in the routines DUSKYF and DUSKYS must be identical.

Once the factorization has been obtained, the routine DUSKYS can be used to solve a system with multiple right sides, by setting $nbx > 1$. The routine can also be called repeatedly, provided the first $4n$ elements of the work array IWRK remain unchanged between calls.

DUSKYC
Unsymmetric Sparse Matrix Condition Number Estimation Using Skyline Storage Scheme

Format

DUSKYC (n, au, iaudiag, nau, al, ialdiag, nal, iparam, rparam, iwrk, rwrk, ierror)

Arguments

n
integer*4
On entry, the order of the matrix *A*.
On exit, **n** is unchanged.

au
real*8
On entry, an array containing information on the *LDU* factorization of the matrix *A*. The factorization has been obtained by a previous call to the routine DUSKYF, and **au** must remain unchanged between the calls to DUSKYF and DUSKYC. If *istore* = 1 or 2, then **au** contains the upper triangular part, including the diagonal, of the factorization of the matrix *A*, stored in the profile-in or diagonal-out mode, respectively. Array AU is of length at least *nau*, where *nau* is the envelope size of the upper triangular part of *A*, including the diagonal. If *istore = 3*, then **au** contains the *LDU* factorization of the matrix *A*, stored in the structurally symmetric, profile-in storage mode. In this case, array AU is of length at least *nau*, where *nau* is the envelope size of the matrix *A*.
On exit, **au** is unchanged.

iaudiag
integer*4
On entry, an array containing the pointers to the locations of the diagonal elements in array AU. **iaudiag** is of length at least n for the profile-in and the structurally symmetric profile-in storage modes. **iaudiag** is of length at least $(n + 1)$ for the diagonal-out storage mode.
On exit, **iaudiag** is unchanged.

nau
integer*4
On entry, the number of elements stored in array AU. If *istore* = 1 or 2, then *nau* is the envelope size of the upper triangular part of the matrix *A*. If *istore = 3*, then *nau* is the envelope size of the matrix *A*. For the profile-in and the structurally symmetric profile-in storage modes, *nau* = IAUDIAG(*n*). For the diagonal-out storage mode, *nau* = IAUDIAG(*n+1*) - 1.
On exit, **nau** is unchanged.

al
real*8
On entry, an array containing information on the *LDU* factorization of the matrix *A*. The factorization has been obtained by a previous call to the routine DUSKYF and **al** must remain unchanged between the calls to the routines DUSKYF and DUSKYC. If *istore* = 1 or 2, then **al** contains the lower triangular part, including the diagonal, of the *LDU* factorization of the matrix *A*, stored in the profile-in or diagonal-out mode, respectively. Storage is allocated for the diagonal elements,

though the elements themselves are stored as part of **au**. Array AL is of length at least *nal*, where *nal* is the envelope size of the lower triangular part of A, including the diagonal. If *istore* = *3*, then **al** is a dummy argument.
On exit, **al** is unchanged.

ialdiag
integer*4
On entry, an array containing the pointers to the locations of the diagonal elements in array AL. **ialdiag** is of length at least n for the profile-in storage mode. **ialdiag** is of length at least $(n+1)$ for the diagonal-out storage mode. If *istore* = *3*, then **ialdiag** is a dummy argument.
On exit, **ialdiag** is unchanged.

nal
integer*4
On entry, the number of elements stored in array AL. If *istore* = *1 or 2*, then *nal* is the envelope size of the lower triangular part of the matrix A. For the profile-in storage mode, $nal = IALDIAG(n)$. For the diagonal-out storage mode, $nal = IALDIAG(n+1) - 1$. If *istore* = *3*, then **nal** is a dummy argument.
On exit, **nal** is unchanged.

iparam
integer*4
An array of length at least 100, containing the integer parameters for the condition number estimator.

iparam(1): niparam
On entry, defines the length of the array IPARAM. $niparam \geq 100$.
On exit, **iparam(1)** is unchanged.

iparam(2): nrparam
On entry, defines the length of the array RPARAM. $nrparam \geq 100$.
On exit, **iparam(2)** is unchanged.

iparam(3): niwrk
On entry, defines the size of the integer work array, IWRK. $niwrk \geq 5n$.
On exit, **iparam(3)** is unchanged.

iparam(4): nrwrk
On entry, defines the size of the integer work array, IWRK. $niwrk \geq 2n$.
On exit, **iparam(4)** is unchanged.

iparam(5): iounit
On entry, defines the I/O unit number for printing error messages and information from the routine DUSKYC. The I/O unit must be opened in the calling subprogram. If $iounit \leq 0$, then no output is generated.
On exit, **iparam(5)** is unchanged.

iparam(6): iolevel
On entry, defines the message level that determines the amount of information printed out to *iounit*, when *iounit* > 0:

iolevel = 0 : fatal error messages only
iolevel = 1 : error messages and minimal information
iolevel = 2 : error messages and detailed information

On exit, **iparam(6)** is unchanged.

iparam(7): idefault
On entry, defines if the default values should be used in arrays IPARAM and RPARAM. If *idefault* = 0, then the following default values are assigned:

IPARAM(1) = *niparam* = 100
IPARAM(2) = *nrparam* = 100
IPARAM(6) = *iolevel* = 0
IPARAM(8) = *istore* = 1
IPARAM(9) = *inorm* = 1

If *idefault* = 1, then you must assign values to the above variables before the call to the DUSKYC routine.
On exit, **iparam(7)** is unchanged.

iparam(8): istore
On entry, defines the type of storage scheme used for the skyline matrix. If *istore* = 1, the unsymmetric matrix A is stored using the profile-in storage mode; if *istore* = 2, the unsymmetric matrix A is stored using the diagonal-out storage mode; if *istore* = 3, the unsymmetric matrix A is stored using the structurally symmetric storage mode. Default: *istore* = 1.
On exit, **iparam(8)** is unchanged.

iparam(9): inorm
On entry, defines the matrix quantity to be evaluated:

inorm = 1 : 1-norm
inorm = 2 : ∞-norm

On exit, **iparam(9)** is unchanged.

rparam
real*8
An array of length at least 100, containing the real parameters for the condition number estimator.

rparam(1): anorm
On entry, **rparam(1)** contains the norm of the matrix A that has been obtained by a prior call to the routine DUSKYN. If *inorm = 1*, then *anorm* must contain the 1-norm of A. If *inorm = 2*, then *anorm* must contain the ∞-norm of A.
On exit, **rparam(1)** is unchanged.

rparam(2): ainorm
On entry, **rparam(2)** is an unspecified variable.
On exit, **rparam(2)** contains the estimate of the norm of the matrix A^{-1} that is evaluated by the routine DUSKYC. If *inorm = 1*, then *ainorm* contains the estimate of the 1-norm of A^{-1}. If *inorm = 2*, then *ainorm* contains the estimate of the ∞-norm of A^{-1}.

rparam(3): rcond
On entry, **rparam(3)** is an unspecified variable.
On exit, **rparam(3)** contains the reciprocal of the estimate of the condition number of the matrix A that is evaluated by the routine DUSKYC. If *inorm = 1*, then *rcond* contains the reciprocal of the estimate of the 1-norm condition number of A. If *inorm = 2*, then *rcond* contains the reciprocal of the estimate of the ∞-norm condition number of A.

iwrk
integer*4
On entry, an array of length at least $5n$ used for integer workspace. The first $4n$ elements of the array IWRK generated by the routine DUSKYF must be passed unchanged to the routine DUSKYC.
On exit, the first $4n$ elements of **iwrk** are unchanged. The next n elements are used as integer workspace by the routine DUSKYC.

rwrk
real*8
On entry, an array of length at least $2n$ used for real workspace.
On exit, the first $2n$ elements of **rwrk** are overwritten.

ierror
integer*4
On entry, an unspecified variable.
On exit, **ierror** contains the error flag. A value of zero indicates a normal exit from the routine DUSKYC.

Description

DUSKYC obtains the reciprocal of the estimate of the condition number of the symmetric matrix A as:

$$rcond(A) = \frac{1}{||A|| \cdot ||A^{-1}||}$$

where either the 1-norm or the ∞-norm is used.

The 1-norm of A^{-1} or A^{-T} is obtained using the LAPACK routine DLACON, that uses Higham's modification [Higham 1988] of Hager's method [Hager 1984]. This routine uses reverse communication for the evaluation of matrix-vector products. As the matrix under consideration is A^{-1} for the 1-norm case and A^{-T} for the ∞-norm case, routine DUSKYC requires calls to the routine DUSKYS. Hence the first $4n$ elements of the integer work array, IWRK, which are generated by the routine DUSKYF and used by the routine DUSKYS, must remain unchanged between the calls to the routines DUSKYF and DUSKYC. The storage scheme used in the routines DUSKYF and DUSKYC must be identical.

DUSKYR
Unsymmetric Sparse Iterative Refinement Using Skyline Storage Scheme

Format

DUSKYR (n, au, auf, iaudiag, nau, al, alf, ialdiag, nal, b, ldb, x, ldx, ferr, berr, nbx, iparam, rparam, iwrk, rwrk, ierror)

Arguments

n
integer*4
On entry, the order of the matrix *A*.
On exit, **n** is unchanged.

au
real*8
On entry, an array containing information on the matrix *A*. If *istore* = 1 or 2, then **au** contains the upper triangular part, including the diagonal, of the matrix *A*, stored in the profile-in or diagonal-out mode, respectively. Array AU is of length at least *nau*, where *nau* is the envelope size of the upper triangular part of *A*, including the diagonal. If *istore* = *3*, then **au** contains the matrix *A*, stored in the structurally symmetric, profile-in storage mode. In this case, array AU is of length at least *nau*, where *nau* is the envelope size of the matrix *A*.
On exit, **au** is unchanged.

auf
real*8
On entry, if *istore* = 1 or 2, **auf** contains the factors *U* and *D* of the *LDU* factorization of the matrix *A*. Array AUF is of length at least *nau*, where *nau* is the envelope size of the upper triangular part of *A*, including the diagonal. If *istore* = *3*, then auf contains the *LDU* factorization of the matrix *A*. In this case, array AUF is of length at least *nau*, where *nau* is the envelope size of the matrix *A*. The *LDU* factorization has been obtained by a prior call to the routine DUSKYF. **auf** must remain unchanged between calls to the routines DUSKYF and DUSKYR.
On exit, **auf** is unchanged.

iaudiag
integer*4
On entry, an array containing the pointers to the locations of the diagonal elements in the arrays AU and AUF. **iaudiag** is of length at least n for the profile-in and the structurally symmetric profile-in storage modes. **iaudiag** is of length at least $(n+1)$ for the diagonal-out storage mode.
On exit, **iaudiag** is unchanged.

nau
integer*4
On entry, the number of elements stored in array AU. If *istore* = 1 or 2, then *nau* is the envelope size of the upper triangular part of the matrix *A*. If *istore* = *3*, then *nau* is the envelope size of the matrix *A*. For the profile-in and the structurally symmetric profile-in storage modes, *nau* = IAUDIAG(*n*). For the diagonal-out storage mode, *nau* = IAUDIAG(*n+1*) - 1.

On exit, **nau** is unchanged.

al
real*8
On entry, an array containing information on the matrix A. If *istore* = 1 or 2, then **al** contains the lower triangular part, including the diagonal, of the matrix A, stored in the profile-in or diagonal-out mode, respectively. Storage is allocated for the diagonal elements, though the elements themselves are not stored. Array AL is of length at least nal, where nal is the envelope size of the lower triangular part of A, including the diagonal. If *istore* = *3*, then **al** is a dummy argument.
On exit, **al** is unchanged.

alf
real*8
On entry, if *istore* = 1 or 2, **alf** contains the factor L of the LDU factorization of the matrix A. Array ALF is of length at least nal, where nal is the envelope size of the lower triangular part of A, including the diagonal. If *istore* = *3*, then **alf** is a dummy argument. The LDU factorization is obtained from a prior call to the routine DUSKYF. **alf** must remain unchanged between calls to the routines DUSKYF and DUSKYR.
On exit, **alf** is unchanged.

ialdiag
integer*4
On entry, an array containing the pointers to the locations of the diagonal elements in the arrays AL and ALF. **ialdiag** is of length at least n for the profile-in storage mode. **ialdiag** is of length at least $(n+1)$ for the diagonal-out storage mode. If *istore* = *3*, then **ialdiag** is a dummy argument.
On exit, **ialdiag** is unchanged.

nal
integer*4
On entry, the number of elements stored in array AL. If *istore* = 1 or 2, then nal is the envelope size of the lower triangular part of the matrix A. For the profile-in storage mode, nal = IALDIAG(n). For the diagonal-out storage mode, nal = IALDIAG(n+1) - 1. If *istore* = *3*, then **nal** is a dummy argument.
On exit, **nal** is unchanged.

b
real*8
On entry, a two-dimensional array B of order ldb by at least nbx, containing the nbx right sides.
On exit, **b** is unchanged.

ldb
integer*4
On entry, the leading dimension of array B. $ldb \geq n$.
On exit, **ldb** is unchanged.

x
real*8
On entry, a two-dimensional array X of order ldx by at least nbx, containing the nbx solution vectors obtained after a call to the routine DUSKYS.
On exit, **x** contains the improved solutions obtained after iterative refinement.

ldx
integer*4
On entry, the leading dimension of array X. $ldx \geq n$.
On exit, **ldx** is unchanged.

ferr
real*8
On entry, an array FERR of length at least nbx, whose elements are unspecified variables.
On exit, **ferr** contains the estimated error bounds for each of the nbz solution vectors.

berr
real*8
On entry, an array BERR of length at least nbx, whose elements are unspecified variables.
On exit, **berr** contains the component-wise relative backward error for each of the nbz solution vectors.

nbx
integer*4
On entry, the number of right sides.
On exit, **nbx** is unchanged.

iparam
integer*4
An array of length at least 100, containing the integer parameters for the iterative refinement and error bounds calculation.

iparam(1): niparam
On entry, defines the length of the array IPARAM. $niparam \geq 100$.
On exit, **iparam(1)** is unchanged.

iparam(2): nrparam
On entry, defines the length of the array RPARAM. As the real parameters array is not used at present, *nrparam* may be unspecified.
On exit, **iparam(2)** is unchanged.

iparam(3): niwrk
On entry, defines the size of the integer work array, IWRK. $niwrk \geq 5n$.
On exit, **iparam(3)** is unchanged.

iparam(4): nrwrk
On entry, defines the size of the real work array, RWRK. $nrwrk \geq 3n$.
On exit, **iparam(4)** is unchanged.

iparam(5): iounit
On entry, defines the I/O unit number for printing error messages and information from the routine DUSKYR. The I/O unit must be opened in the calling subprogram. If $iounit \leq 0$, no output is generated.
On exit, **iparam(5)** is unchanged.

iparam(6): iolevel
On entry, defines the message level that determines the amount of information printed out to *iounit*, when $iounit > 0$:

iolevel = 0 : fatal error messages only

iolevel = 1 : error messages and minimal information
iolevel = 2 : error messages and detailed information

On exit, **iparam(6)** is unchanged.

iparam(7): idefault
On entry, defines if the default values should be used in arrays IPARAM and RPARAM. If *idefault* = 0, then the following default values are assigned:

IPARAM(1) = *niparam* = 100
IPARAM(6) = *iolevel* = 0
IPARAM(8) = *istore* = 1
IPARAM(9) = *itrans* = 0
IPARAM(10) = *itmax* = 5

If *idefault* = 1, then you must assign values to the above variables before the call to the DUSKYR routine.
On exit, **iparam(7)** is unchanged.

iparam(8): istore
On entry, defines the type of storage scheme used for the skyline matrix. If *istore* = 1, the unsymmetric matrix A is stored using the profile-in storage mode; if *istore* = 2, the unsymmetric matrix A is stored using the diagonal-out storage mode; if *istore* = 3, the unsymmetric matrix A is stored using the structurally symmetric profile-in storage mode. Default: *istore* = 1.
On exit, **iparam(8)** is unchanged.

iparam(9): itrans
On entry, defines the form of matrix used in the iterative refinement. If *itrans* = *0*, the system refined is $AX = B$; if *itrans* = *1*, the system refined is $A^T X = B$. Default: *itrans* = *0*.
On exit, **iparam(9)** is unchanged.

iparam(10): itmax
On entry, defines the maximum number of iterations for the iterative refinement process. Default: *itmax* = *5*.
On exit, **iparam(10)** is unchanged.

rparam
real*8
An array of length at least 100, containing the real parameters for the iterative refinement and error bounds calculation.
On exit, **rparam** is unchanged. **rparam** is not used by the routine DUSKYR at present, but is reserved for future use. It can be a dummy variable.

iwrk
integer*4
On entry, an array of length at least $5n$ used for integer workspace. The first $4n$ elements of the array IWRK, generated by the routine DUSKYF, should be passed unchanged to the routine DUSKYR.
On exit, the first $4n$ elements of **iwrk** are unchanged.

rwrk
real*8
On entry, an array of length at least $3n$ used for real workspace.
On exit, the first $3n$ elements of **rwrk** are overwritten.

ierror
integer*4
On entry, an unspecified variable.
On exit, **ierror** contains the error flag. A value of zero indicates a normal exit from the routine DUSKYR.

Description

DUSKYR obtains an improved solution to the system:

$$AX = B$$

or:

$$A^T X = B$$

via iterative refinement. This is done by calculating the matrix of residuals R using the matrix of solutions $\hat{X}$ obtained from DUSKYS, and obtaining a new matrix of solutions X_{new} as follows:

For *itrans = 0*:

$$R = B - A\hat{X}$$

$$\delta X = A^{-1} R$$

and:

$$X_{new} = \hat{X} + \delta X$$

For *itrans = 1*:

$$R = B - A^T \hat{X}$$

$$\delta X = A^{-T} R$$

and:

$$X_{new} = \hat{X} + \delta X$$

The process of iterative refinement therefore requires both the original matrix A as well as the LDU factorization obtained via the routine DUSKYF. Since this routine overwrites the matrix A by the factorization, a copy of the matrix must be made before the call to DUSKYF. Further, both the right sides B and the solution vectors $\hat{X}$ are required during iterative refinement. Since the solution process in the routine DUSKYS overwrites the right sides with the solution vectors, a copy of the right sides must be made before the call to the routine DUSKYS.

In addition to the iterative refinement of the solution vectors, the routine DUSKYR also provides the component-wise relative backward error, $berr$ and the estimated forward error bound, $ferr$, for each solution vector [Arioli, Demmel, Duff 1989, Anderson et. al. 1992]. $berr$ is the smallest relative change in any entry of A or b that makes $\hat{x}$ an exact solution. $ferr$ bounds the magnitude of the largest entry in $\hat{x} - x_{true}$ divided by the magnitude of the largest entry in $\hat{x}$.

The process of iterative refinement is continued as long as all of the following conditions are satisfied [Arioli, Demmel, Duff 1989]:

- The number of iterations of the iterative refinement process is less than IPARAM(10) = *itmax*.
- $berr$ is reduced by at least a factor of 2 during the previous iteration.

- $berr$ is larger than the machine precision.

The routine DUSKYR is called after a call to the routine DSSKYF to obtain the LDU factorization and a call to the routine DUSKYS to obtain the solution $\hat{x}$. The first $4n$ elements of the integer workspace array IWRK, generated by DUSKYF, contain information for use by DUSKYR and therefore must remain unchanged between the calls to the routines DUSKYF and DUSKYR. The storage scheme used in the routines DUSKYF, DUSKYS, and DUSKYR must be identical. The value of $itrans$ must be the same in the calls to the routines DUSKYS and DUSKYR.

DUSKYD
Unsymmetric Sparse Simple Driver Using Skyline Storage Scheme

Format

DUSKYD (n, au, iaudiag, nau, al, ialdiag, nal, bx, ldbx, nbx, iparam, rparam, iwrk, rwrk, ierror)

Arguments

n
integer*4
On entry, the order of the matrix *A*.
On exit, **n** is unchanged.

au
real*8
On entry, an array containing information on the matrix *A*. If *istore* = 1 or 2, then **au** contains the upper triangular part, including the diagonal, of the matrix *A*, stored in the profile-in or diagonal-out mode, respectively. Array AU is of length at least *nau*, where *nau* is the envelope size of the upper triangular part of *A*, including the diagonal. If *istore = 3*, then **au** contains the matrix *A*, stored in the structurally symmetric, profile-in storage mode. In this case, array AU is of length at least *nau*, where *nau* is the envelope size of the matrix *A*.
On exit, if *istore* = 1 or 2, **au** contains the factors *U* and *D* of the *LDU* factorization of the matrix *A*. If *istore = 3*, then **au** contains the factors *L*, *U* and *D* of the *LDU* factorization of the matrix *A*. **au** must remain unchanged between the call to the routine DUSKYD and any routines that use the factors such as DUSKYS, DUSKYC, and DUSKYR.

iaudiag
integer*4
On entry, an array containing the pointers to the locations of the diagonal elements in the arrays AU. **iaudiag** is of length at least n for the profile-in and the structurally symmetric profile-in storage modes. **iaudiag** is of length at least $(n + 1)$ for the diagonal-out storage mode.
On exit, **iaudiag** is unchanged.

nau
integer*4
On entry, the number of elements stored in array AU. If *istore* = 1 or 2, then *nau* is the envelope size of the upper triangular part of the matrix *A*. If *istore = 3*, then *nau* is the envelope size of the matrix *A*. For the profile-in and the structurally symmetric profile-in storage modes, *nau* = IAUDIAG(*n*). For the diagonal-out storage mode, *nau* = IAUDIAG(*n+1*) - 1.
On exit, **nau** is unchanged.

al
real*8
On entry, an array containing information on the matrix *A*. If *istore* = 1 or 2, then **al** contains the lower triangular part, including the diagonal, of the matrix *A*, stored in the profile-in or diagonal-out mode, respectively. Storage is allocated for the diagonal elements, though the elements themselves are not stored. Array AL is of length at least *nal*, where *nal* is the envelope size of the lower triangular part of *A*, including the diagonal. If *istore = 3*, then **al** is a dummy argument.

On exit, if *istore* = 1 or 2, **al** contains the factor *L* of the *LDU* factorization of the matrix *A*. If *istore* = *3*, then **al** is undefined. **al** must remain unchanged between the call to the routine DUSKYD and any routines that use the factors such as DUSKYS, DUSKYC, and DUSKYR.

ialdiag
integer*4
On entry, an array containing the pointers to the locations of the diagonal elements in the array AL. **ialdiag** is of length at least n for the profile-in storage mode. **ialdiag** is of length at least $n + 1$ for the diagonal-out storage mode. If *istore* = *3*, then **ialdiag** is a dummy argument.
On exit, **ialdiag** is unchanged.

nal
integer*4
On entry, the number of elements stored in array AL. If *istore* = 1 or 2, then nal is the envelope size of the lower triangular part of the matrix *A*. For the profile-in storage mode, nal = IALDIAG(n). For the diagonal-out storage mode, nal = IALDIAG(n+1) - 1. If *istore* = *3*, then **nal** is a dummy argument.
On exit, **nal** is unchanged.

bx
real*8
On entry, a two-dimensional array BX of order nbx, containing the nbx right sides.
On exit, **bx** contains the solutions for the nbx systems.

ldbx
integer*4
On entry, the leading dimension of array BX. $ldbx \geq n$.
On exit, **ldbx** is unchanged.

nbx
real*4
On entry, the number of right sides.
On exit, **nbx** is unchanged.

iparam
integer*4
An array of length at least 100, containing the integer parameters for the simple driver.

iparam(1): niparam
On entry, defines the length of the array IPARAM. $niparam \geq 100$.
On exit, **iparam(1)** is unchanged.

iparam(2): nrparam
On entry, defines the length of the array RPARAM. $nrparam \geq 100$.
On exit, **iparam(2)** is unchanged.

iparam(3): niwrk
On entry, defines the size of the integer work array, IWRK. $niwrk \geq 4n$.
On exit, **iparam(3)** is unchanged.

iparam(4): nrwrk
On entry, defines the size of the real work array, RWRK. As the real work array is not used at present, *nrwrk* can be unspecified.
On exit, **iparam(4)** is unchanged.

iparam(5): iounit
On entry, defines the I/O unit number for printing error messages and information from the routine DUSKYD. The I/O unit must be opened in the calling subprogram. If *iounit* $\leq$0, no output is generated.
On exit, **iparam(5)** is unchanged.

iparam(6): iolevel
On entry, defines the message level that determines the amount of information printed out to *iounit*, when *iounit* > 0:

iolevel = 0 : fatal error messages only
iolevel = 1 : error messages and minimal information
iolevel = 2 : error messages and detailed information

On exit, **iparam(6)** is unchanged.

iparam(7): idefault
On entry, defines if the default values should be used in arrays IPARAM and RPARAM. If *idefault* = 0, then the following default values are assigned:

IPARAM(1) = *niparam* = 100
IPARAM(2) = *nrparam* = 100
IPARAM(6) = *iolevel* = 0
IPARAM(8) = *istore* = 1
IPARAM(9) = *ipvt* = 0
IPARAM(11) = *itrans* = 0
RPARAM(1) = *pvt_sml* = 10^{-12}

If *idefault* = 1, then you must assign values to the above variables before the call to the DUSKYD routine.
On exit, **iparam(7)** is unchanged.

iparam(8): istore
On entry, defines the type of storage scheme used for the skyline matrix. If *istore* = 1, the matrix A is stored using the profile-in storage mode; if *istore* = 2, the matrix A is stored using the diagonal-out storage mode; if *istore* = 3, the unsymmetric matrix A is stored using the structurally symmetric profile-in storage mode. Default: *istore* = 1.
On exit, **iparam(8)** is unchanged.

iparam(9): ipvt
On entry, defines if the factorization should continue when a small pivot, defined by RPARAM(1), is encountered. If *ipvt* = 0 and the absolute value of the pivot element is smaller than *pvt_sml* = RPARAM(1), then the factorization process is stopped and control returned to the calling subprogram. If *ipvt* = 1 and a pivot smaller than RPARAM(1) in absolute value is encountered in the factorization, the process continues. If *ipvt* = 2 and a pivot smaller than RPARAM(1) in absolute value is encountered in the factorization, it is replaced by a predetermined value *pvt_new* = RPARAM(2), and the factorization is continued. Default: *ipvt* = 0.
On exit, **iparam(9)** is unchanged.

iparam(10): ipvt_loc
On entry, an unspecified variable.
On exit, **iparam(10)** contains the location of the first pivot element smaller in absolute value than *pvt_sml*. The pivot element is returned in *pvt_val* = RPARAM(3). If **iparam(10)** = 0, then no such pivot element exists.

iparam(11): itrans
On entry, defines the form of matrix used in the solution. If *itrans = 0*, the system solved is $AX = B$; if *itrans = 1*, the system solved is $A^T X = B$. Default: *itrans = 0*.
On exit, **iparam(11)** is unchanged.

rparam
real*8
An array of length at least 100, containing the real parameters for the simple driver.

rparam(1): pvt_sml
On entry, defines the value of the pivot element which is considered to be small. If a pivot element smaller than *pvt_sml*, in absolute value, is encountered in the factorization process, then, depending on the value of *ipvt* = IPARAM(9), the process either stops, continues or continues after the pivot is set equal to *pvt_new* = RPARAM(2). *pvt_sml > 0*. Recommended value: $10^{-15} \leq pvt\_sml \leq 1$. Default: $pvt\_sml = 10^{-12}$.
On exit, **rparam(1)** is unchanged.

rparam(2): pvt_new
On entry, defines the value to which the pivot element must be set if *ipvt* = 2 and the pivot element is less than *pvt_sml* in absolute value. *pvt_new* should be large enough to avoid overflow when calculating the reciprocal of the pivot element.
On exit, **rparam(2)** is unchanged.

rparam(3): pvt_val
On entry, an unspecified variable.
On exit, **rparam(3)** contains the value of the first pivot element smaller than *pvt_sml* in absolute value. This element occurs at the location returned in IPARAM(10). If no such pivot element is found, the value of *pvt_val* is unspecified.

iwrk
integer*4
On entry, an array of length at least $4n$ used for integer workspace.
On exit, the first $4n$ elements of the array IWRK contain information generated by the factorization routine DUSKYF. This information is required by routines that use the factorization, such as DUSKYS, DUSKYC, and DUSKYR, and should remain unchanged between the call to DUSKYD and any subsequent calls to one of these routines.

rwrk
real*8
On entry, an array used for real workspace.
On exit, **rwrk** is unchanged. Presently, **rwrk** is not used by the routine DSSKYF. It can be a dummy variable.

ierror
integer*4
On entry, an unspecified variable.
On exit, **ierror** contains the error flag. A value of zero indicates a normal exit from the routine DUSKYD.

Description

DUSKYD is a simple driver routine that factors and solves the system:

$$AX = B$$

or:

$$A^T X = B$$

where A is an unsymmetric matrix stored in a skyline form, using either the profile-in storage mode, the diagonal-out storage mode, or the structurally symmetric profile-in storage mode; B is a matrix of nbx right sides and x is the matrix of the corresponding nbx solution vectors. On entry to the routine DUSKYS, the array BX contains the nbx right sides; on exit, these are overwritten by the solution vectors. The variable $itrans$ determines whether the matrix A or A^T is used in the solution process.

The matrix A is first factorized as:

$$A = LDU$$

by a call to the routine DUSKYF. L is a unit lower triangular matrix, U is a unit upper triangular matrix, and D is a diagonal matrix. The routine DUSKYF does not perform any pivoting to preserve the numerical stability of the LDU factorization. It is therefore primarily intended for the solution of systems that do not require pivoting for numerical stability, such as diagonally dominant systems. Caution is urged when using this routine for problems that require pivoting.

If a small pivot, in absolute value, *pvt_sml*, is encountered in the process of factorization, you have the option of either stopping the factorization process and returning to the calling program, continuing the factorization process, or continuing after setting the pivot equal to some predetermined value, *pvt_new*. The location of the first occurrence of a small pivot is returned in *ipvt_loc* and its value in *pvt_val*.

After the factorization has been obtained without any errors, the routine DUSKYD calls the solve routine, DUSKYS, to solve the system. The call to the routine DUSKYD can be followed by a call to the routines DUSKYS, DUSKYC, and DUSKYR, provided that the first $4n$ elements of the integer workspace array IWRK remain unchanged between calls. The real work array RWRK is not used at present.

DUSKYX
Unsymmetric Sparse Expert Driver Using Skyline Storage Scheme

Format

DUSKYX (n, au, auf, iaudiag, nau, al, alf, ialdiag, nal, b, ldb, x, ldx, ferr, berr, nbx, iparam, rparam, iwrk, rwrk, ierror)

Arguments

n
integer*4
On entry, the order of the matrix *A*.
On exit, **n** is unchanged.

au
real*8
On entry, an array containing information on the matrix *A*. If *istore* = 1 or 2, then **au** contains the upper triangular part, including the diagonal, of the matrix *A*, stored in the profile-in or diagonal-out mode, respectively. Array AU is of length at least *nau*, where *nau* is the envelope size of the upper triangular part of *A*, including the diagonal. If *istore* = *3*, then **au** contains the matrix *A*, stored in the structurally symmetric, profile-in storage mode. In this case, array AU is of length at least *nau*, where *nau* is the envelope size of the matrix *A*.
On exit, **au** is unchanged.

auf
real*8
On entry, if RPARAM(9) = *ifactor* = *0*, **auf** is an unspecified array of length at least *nau*. If *ifactor* = *1*, then **auf** contains information on the *LDU* factorization of the matrix *A*. If *istore* = 1 or 2, **auf** contains the factors *U* and *D* of the *LDU* factorization of the matrix *A*. Array AUF is of length at least *nau*, where *nau* is the envelope size of the upper triangular part of *A*, including the diagonal. If *istore* = *3*, then **auf** contains the *LDU* factorization of the matrix *A*. In this case, array AUF is of length at least *nau*, where *nau* is the envelope size of the matrix *A*. The *LDU* factorization has been obtained by a prior call to the routine DUSKYF.
On exit, if *ifactor* = *0*, then **auf** contains information on the *LDU* factorization of the matrix *A*. If *istore* = 1 or 2, **auf** contains the factors *U* and *D* of the *LDU* factorization of the matrix *A*. Array AUF is of length at least *nau*, where *nau* is the envelope size of the upper triangular part of *A*, including the diagonal. If *istore* = *3*, then **auf** contains the *LDU* factorization of the matrix *A*. In this case, array AUF is of length at least *nau*, where *nau* is the envelope size of the matrix *A*. If *ifactor* = *1*, then **auf** is unchanged.

iaudiag
integer*4
On entry, an array containing the pointers to the locations of the diagonal elements in the arrays AU and AUF (if *ifactor* = *0*). **iaudiag** is of length at least *n* for the profile-in and the structurally symmetric profile-in storage modes. **iaudiag** is of length at least $(n+1)$ for the diagonal-out storage mode.
On exit, **iaudiag** is unchanged.

nau
integer*4
On entry, the number of elements stored in array AU. If *istore* = 1 or 2, then *nau* is the envelope size of the upper triangular part of the matrix ***A***. If *istore* = *3*, then *nau* is the envelope size of the matrix ***A***. For the profile-in and the structurally symmetric profile-in storage modes, *nau* = IAUDIAG(*n*). For the diagonal-out storage mode, *nau* = IAUDIAG(*n+1*) - 1.
On exit, **nau** is unchanged.

al
real*8
On entry, an array containing information on the matrix ***A***. If *istore* = 1 or 2, then **al** contains the lower triangular part, including the diagonal, of the matrix ***A***, stored in the profile-in or diagonal-out mode, respectively. Storage is allocated for the diagonal elements, though the elements themselves are not stored. Array AL is of length at least *nal*, where *nal* is the envelope size of the lower triangular part of ***A***, including the diagonal. If *istore* = *3*, then **al** is a dummy argument.
On exit, **al** is unchanged.

alf
real*8
On entry, if IPARAM(9) = *ifactor* = 0, **alf** is an unspecified array of length at least *nal*. If *ifactor* = *1*, then **alf** contains information on the *LDU* factorization of the matrix ***A***. On entry, if *istore* = 1 or 2, **alf** contains the factor *L* of the *LDU* factorization of the matrix ***A***. Array ALF is of length at least *nal*, where *nal* is the envelope size of the lower triangular part of ***A***, including the diagonal. If *istore* = *3*, then **alf** is a dummy argument. The *LDU* factorization is obtained from a prior call to the routine DUSKYF.
On exit, if $ifactor = 0$, **alf** contains information on the *LDU* factorization of the matrix ***A***. If *istore* = 1 or 2, **alf** contains the factor *L* of the *LDU* factorization of the matrix ***A***. Array ALF is of length at least *nal*, where *nal* is the envelope size of the lower triangular part of ***A***, including the diagonal. If *istore* = *3*, then **alf** is a dummy argument. If $ifactor = 1$, then **alf** is unchanged.

ialdiag
integer*4
On entry, an array containing the pointers to the locations of the diagonal elements in the arrays AL and ALF (if *ifactor* = *1*). **ialdiag** is of length at least *n* for the profile-in storage mode. **ialdiag** is of length at least $(n + 1)$ for the diagonal-out storage mode. If *istore* = *3*, then **ialdiag** is a dummy argument.
On exit, **ialdiag** is unchanged.

nal
integer*4
On entry, the number of elements stored in array AL. If *istore* = 1 or 2, then *nal* is the envelope size of the lower triangular part of the matrix ***A***. For the profile-in storage mode, *nal* = IALDIAG(*n*). For the diagonal-out storage mode, *nal* = IALDIAG($n + 1$) - 1. If *istore* = *3*, then **nal** is a dummy argument.
On exit, **nal** is unchanged.

b
real*8
On entry, a two-dimensional array B of order *ldb* by at least *nbx*, containing the *nbx* right sides.
On exit, **b** is unchanged.

ldb
integer*4
On entry, the leading dimension of array B. $ldb \geq n$.
On exit, **ldb** is unchanged.

x
real*8
On entry, a two-dimensional array X of order ldx by at least nbx, containing the nbx solution vectors obtained after a call to the routine DUSKYS.
On exit, **X** contains the improved solutions obtained after iterative refinement.

ldx
integer*4
On entry, the leading dimension of array X. $ldx \geq n$.
On exit, **ldx** is unchanged.

ferr
real*8
On entry, an array FERR of length at least nbx, whose elements are unspecified variables.
On exit, **ferr** contains the estimated error bounds for each of the nbx solution vectors.

berr
real*8
On entry, an array BERR of length at least nbx, whose elements are unspecified variables.
On exit, **berr** contains the component-wise relative backward error for each of the nbx solution vectors.

nbx
integer*4
On entry, the number of right sides.
On exit, **nbz** is unchanged.

iparam
integer*4
An array of length at least 100, containing the integer parameters for the expert driver.

iparam(1): niparam
On entry, defines the length of the array IPARAM. $niparam \geq 100$.
On exit, **iparam(1)** is unchanged.

iparam(2): nrparam
On entry, defines the length of the array RPARAM. $nrparam \geq 100$.
On exit, **iparam(2)** is unchanged.

iparam(3): niwrk
On entry, defines the size of the integer work array, IWRK. $niwrk \geq 5n$.
On exit, **iparam(3)** is unchanged.

iparam(4): nrwrk
On entry, defines the size of the real work array, RWRK. $nrwrk \geq 3n$.
On exit, **iparam(4)** is unchanged.

iparam(5): iounit
On entry, defines the I/O unit number for printing error messages and information from the routine DUSKYX. The I/O unit must be opened in the calling subprogram. If *iounit* $\leq$0, no output is generated.
On exit, **iparam(5)** is unchanged.

iparam(6): iolevel
On entry, defines the message level that determines the amount of information printed out to *iounit*, when *iounit* > 0:

iolevel = 0 : fatal error messages only
iolevel = 1 : error messages and minimal information
iolevel = 2 : error messages and detailed information

On exit, **iparam(6)** is unchanged.

iparam(7): idefault
On entry, defines if the default values should be used in arrays IPARAM and RPARAM. If *idefault* = 0, then the following default values are assigned:

IPARAM(1) = *niparam* = 100
IPARAM(2) = *nrparam* = 100
IPARAM(6) = *iolevel* = 0
IPARAM(8) = *istore* = 1
IPARAM(9) = *ifactor* = 0
IPARAM(10) = *idet* = 0
IPARAM(11) = *ipvt* = 0
IPARAM(13) = *itrans* = 0
IPARAM(14) = *itmax* = 5
RPARAM(1) = *pvt_sml* = 10^{-12}

If *idefault* = 1, then you must assign values to the above variables before the call to the DUSKYX routine.
On exit, **iparam(7)** is unchanged.

iparam(8): istore
On entry, defines the type of storage scheme used for the skyline matrix. If *istore* = 1, the unsymmetric matrix ***A*** is stored using the profile-in storage mode; if *istore* = 2, the unsymmetric matrix ***A*** is stored using the diagonal-out storage mode; if *istore* = 3, the unsymmetric matrix ***A*** is stored using the structurally symmetric profile-in storage mode. Default: *istore* = 1.
On exit, **iparam(8)** is unchanged.

iparam(9): ifactor
On entry, defines if the matrix ***A*** has already been factored. If *ifactor = 0*, the matrix is unfactored and arrays AUF and ALF are unspecified. If *ifactor = 1*, the matrix has been factored by a prior call to the routine DUSKYF, and the arrays AUF and array ALF contain the *LDU* factorization of ***A***. Default: *ifactor = 0*.
On exit, **iparam(9)** is unchanged.

iparam(10): idet
On entry, defines if the determinant of the matrix ***A*** is to be calculated. If *idet* = 0, then the determinant is not calculated; if *idet* = 1, the determinant is calculated as *det_base* * $10^{\text{det\_pwr}}$. See RPARAM(4) and RPARAM(5). If *ifactor* = 1, then IPARAM(10) is unspecified. Default: *idet* = 0.
On exit, **iparam(10)** is unchanged.

iparam(11): ipvt
On entry, defines if the factorization should continue when a small pivot, defined by RPARAM(1), is encountered. If *ipvt* = 0 and the absolute value of the pivot element is smaller than *pvt_sml* = RPARAM(1), then the factorization process is stopped and control returned to the calling subprogram. If *ipvt* = 1 and a pivot smaller than RPARAM(1) in absolute value is encountered in the factorization, the process continues. If *ipvt* = 2 and a pivot smaller than RPARAM(1) in absolute value is encountered in the factorization, it is replaced by a predetermined value *pvt_new* = RPARAM(2), and the factorization is continued. If *ifactor* = 1, then IPARAM(11) is unspecified. Default: *ipvt* = 0.
On exit, **iparam(11)** is unchanged.

iparam(12): ipvt_loc
On entry, an unspecified variable.
On exit, **iparam(12)** contains the location of the first pivot element smaller in absolute value than *pvt_sml*. The pivot element is returned in *pvt_val* = RPARAM(3). If **iparam(12)** = 0, then no such pivot element exists. If *ifactor* = 1, then IPARAM(12) is unspecified.

iparam(13): itrans
On entry, defines the form of matrix used in the iterative refinement. If *itrans* = *0*, the system refined is $AX = B$; if *itrans* = *1*, the system refined is $A^T X = B$. Default: *itrans* = *0*.
On exit, **iparam(13)** is unchanged.

iparam(14): itmax
On entry, defines the maximum number of iterations for the iterative refinement process. Default: *itmax* = *5*.
On exit, **iparam(14)** is unchanged.

rparam
real*8
An array of length at least 100, containing the real parameters for the expert driver.

rparam(1): pvt_sml
On entry, defines the value of the pivot element which is considered to be small. If a pivot element smaller than *pvt_sml*, in absolute value, is encountered in the factorization process, then, depending on the value of *ipvt* = IPARAM(11), the process either stops, continues or continues after the pivot is set equal to *pvt_new* = RPARAM(2). If *ifactor* = *1*, then RPARAM (1) is unspecified. *pvt_sml* > *0*. Recommended value: $10^{-15} \leq pvt\_sml \leq 1$. Default: *pvt_sml* = 10^{-12}.
On exit, **rparam(1)** is unchanged.

rparam(2): pvt_new
On entry, defines the value to which the pivot element must be set if *ipvt* = 2 and the pivot element is less than *pvt_sml* in absolute value. *pvt_new* should be large enough to avoid overflow when calculating the reciprocal of the pivot element. If *ifactor* = *1*, then RPARAM(2) is unspecified.
On exit, **rparam(2)** is unchanged.

rparam(3): pvt_val
On entry, an unspecified variable.
On exit, **rparam(3)** contains the value of the first pivot element smaller than *pvt_sml* in absolute value. The location of this element is returned in *ipvt_loc* = IPARAM(12). If *ifactor* = *1*, then the RPARAM(3) is unspecified.

rparam(4): det_base
On entry, an unspecified variable.
On exit, defines the base for the determinant of the matrix A. If *idet* = 1, the determinant is calculated as *det_base* * $10^{\text{det\_pwr}}$. If *ifactor* = *1*, then RPARAM(4) is unspecified. $1.0 \leq$ *det_base* ≤ 10.0).

rparam(5): det_pwr
On entry, an unspecified variable.
On exit, defines the power for the determinant of the matrix A. If *idet* = 1, the determinant is calculated as *det_base* * $10^{\text{det\_pwr}}$. If $ifactor = 1$, then RPARAM(5) is unspecified.

rparam(6): anorm
On entry, an unspecified variable.
On exit, **rparam(6)** contains the 1-norm or the ∞-norm of the matrix A.

rparam(7): ainorm
On entry, an unspecified variable.
On exit, **rparam(7)** contains the estimate of the 1-norm or the ∞-norm of A^{-1}.

rparam(8): rcond
On entry, an unspecified variable.
On exit, **rparam(8)** contains the reciprocal of the estimate of the 1-norm or the ∞-norm condition number of the matrix A.

iwrk
integer*4
On entry, an array of length at least $5n$ used for integer workspace. If $ifactor = 1$, then the first $4n$ elements of the array IWRK contain information generated by the routine DUSKYF. If $ifactor = 0$, then this information is unspecified.
On exit, the first $4n$ elements of the array IWRK contain information generated by the routine DUSKYF. This information is used by the routines DUSKYS and DUSKYR, and should therefore remain unchanged between the call to the routine DUSKYX and any subsequent call to the routines DUSKYS and DUSKYR.

rwrk
real*8
On entry, an array of length at least $3n$ used for real workspace.
On exit, the first $3n$ elements of **rwrk** are overwritten.

ierror
integer*4
On entry, an unspecified variable.
On exit, **ierror** contains the error flag. A value of zero indicates a normal exit from the routine DUSKYX.

Description

DUSKYX is an expert driver routine that:

- Obtains the LDU factorization of the matrix A via a call to the routine DUSKYF.
- If the factorization is successful, obtains the 1-norm (or ∞-norm) condition number estimate of the matrix A by a call to the routine DUSKYC.

- If the reciprocal of the condition number estimate is greater than the machine precision, DUSKYX uses the factorization to solve the system:

$$AX = B$$

  or:

$$A^T X = B$$

  using the routine DUSKYS.

- Improves the solution X via iterative refinement and obtains the error bounds using the routine DUSKYR.

DUSKYX first obtains the factorization of the symmetric matrix A as:

$$A = LDU$$

where L is a unit triangular matrix, D is a diagonal matrix and U is a unit upper triangular matrix. The matrix A is stored in a skyline form, using either the profile-in storage mode or the diagonal-out storage mode or the structurally symmetric profile-in storage mode. If the matrix is already factored, as indicated by *ifactor*, then this step is skipped.

The routine DUSKYF does not perform any pivoting to preserve the numerical stability of the LDU factorization. It is therefore primarily intended for the solution of systems that do not require pivoting for numerical stability, such as diagonally dominant systems. Caution is urged when using this routine for problems that require pivoting.

If a small pivot, in absolute value, *pvt_sml*, is encountered in the process of factorization, you have the option of either stopping the factorization process and returning to the calling subprogram, continuing the factorization process with the small value of the pivot, or continuing after setting the pivot equal to some predetermined value, *pvt_new*. The location of the first occurrence of a small pivot is returned in *ipvt_loc* and its value in *pvt_val*.

In addition to the LDU factorization, the routine DUSKYF can be used to obtain the determinant of A. If factorization process is stopped at row i due to a small pivot, then the determinant is evaluated for rows 1 through $(i-1)$.

The routine DUSKYX does not allow a partial factorization of the matrix A. If a partial factorization of A is required, the routine DUSKYF is recommended.

DUSKYC obtains the reciprocal of the estimate of the condition number of the unsymmetric matrix A as:

$$rcond(A) = \frac{1}{||A|| \cdot ||A^{-1}||}$$

If the system being solved is:

$$AX = B$$

the reciprocal of the 1-norm condition number estimate is calculated. If the system being solved is:

$$A^T X = B$$

the reciprocal if the ∞-norm of condition number estimate is calculated. The 1-norm of A^{-1} or A^{-T} is obtained using the LAPACK routine DLACON, which uses Higham's modification [Higham 1988] of Hager's method [Hager 1984]. If the reciprocal of the condition number estimate is larger than the machine precision, the routine DUSKYX solves the system via a call to the routine DUSKYS and then improves on the solution via iterative refinement. This is done by calculating the matrix of residuals R using the matrix of solutions $\hat{X}$ obtained from DUSKYS, and obtaining a new matrix of solutions X_{new} as follows:

For *itrans = 0:*

$$R = B - A\hat{X}$$

$$\delta X = A^{-1}R$$

and:

$$X_{new} = \hat{X} + \delta X$$

For *itrans = 1:*

$$R = B - A^{T}\hat{X}$$

$$\delta X = A^{-T}R$$

and:

$$X_{new} = \hat{X} + \delta X$$

In addition to the iterative refinement of the solution vectors, the routine DUSKYX also provides the component-wise relative backward error, $berr$ and the estimated forward error bound, $ferr$, for each solution vector [Arioli, Demmel, Duff 1989, Anderson et. al. 1992]. $berr$ is the smallest relative change in any entry of A or b that makes $\hat{x}$ an exact solution. $ferr$ bounds the magnitude of the largest entry in $\hat{x} - x_{true}$ divided by the magnitude of the largest entry in $\hat{x}$.

The process of iterative refinement is continued as long as all of the following conditions are satisfied [Arioli, Demmel, Duff 1989]:

- The number of iterations of the iterative refinement process is less than IPARAM(10) = $itmax$.
- $berr$ reduces by at least a factor of 2 during the previous iteration.
- $berr$ is larger than the machine precision.

The first $4n$ elements of the integer workspace array IWRK generated by DUSKYF, contain information for use by the routines DUSKYS and DUSKYR and therefore must remain unchanged between the calls to the routine DUSKYX and any subsequent calls to the routine DUSKYS and DUSKYR.

14
Using the VLIB Routines

DXML includes a special set of routines that are similar to industry standard array processor library routines. This special set of run-time library routines operates on vectors—thus the name VLIB. This chapter provides information about the following topics:

- Operations performed by the VLIB subprograms (Section 14.1)
- Vector storage (Section 14.2)
- Subprogram naming conventions (Section 14.3)
- Subprogram summaries (Section 14.4)
- Calling VLIB subprograms (Section 14.5)
- Arguments used in the subprograms (Section 14.6)
- Error handling for VLIB subprograms (Section 14.7)
- A look at a VLIB subprogram and its use (Section 14.8)

A description of each VLIB subprogram follows this chapter.

14.1 VLIB Operations

VLIB operations work with vectors. These routines make it easier to port existing array processor-oriented code, as well as provide enhanced performance, where possible. Many simple array-oriented routines (such as adding a constant to each element of an array) are more suitably coded by using the corresponding loop and letting compiler optimizations improve performance. More complex routines are suitably encapsulated in highly tuned routines such as those in VLIB.

For example, the VLIB routines include routines for transcendental functions. In this case, the VLIB functions generally deliver performance 1.5 to 2 times faster than the alternative of simply calling the appropriate run-time library function in a loop. Careful code scheduling and algorithm design within the VLIB routines take advantage of the fact that the input is a vector.

14.1.1 Types of Operations

The VLIB subprograms operate on only one vector (or possibly scalar), returning one or more vectors (or possibly scalars) as output. The results of these operations do not depend on the order in which the elements of the vector are processed.

14.1.2 Accuracy

The VLIB subprograms provide the same accuracy as the corresponding run-time library routines.

14.2 Vector Storage

For the VLIB subprograms, a vector is stored in a one-dimensional array. The calling conventions for negative increment accesses to an array differ from BLAS1 conventions, but follow conventions used in existing array processor libraries. See Section 14.2.2.

14.2.1 Defining a Vector in an Array

A vector is usually stored in a one-dimensional array. The elements of a vector are stored in order, but the elements are not necessarily contiguous.

An array can be much larger than a vector that is stored in the array. The storage of a vector is defined using three arguments in a DXML subprogram argument list:

- Vector length: Number of elements in the vector
- Vector location: Base address of the vector in the array
- Stride: Space, or increment, between consecutive elements of the vector as stored in the array

These three arguments together specify which elements of an array are selected to become the vector.

14.2.1.1 Vector Length

To specify the length n of a vector, you specify an integer value for a length argument, such as the **n** argument. The length of a vector can be less than the length of the array that specifies the vector.

Vector length can also be thought of as the number of elements of the associated array that a subroutine will process. Processing continues until n elements have been processed.

14.2.1.2 Vector Location

The location of a vector is specified by the argument for the vector in the DXML subprogram argument list. Usually, an array such as X is declared, for example, X(1:20) or X(20). In this case, if you want to specify vector x as starting at the first element of an array X, the argument is specified as X(1) or X. If you want to specify vector x as starting at the fifth element of X, the argument is specified as X(5).

However, in an array X that is declared as X(3:20), with a lower bound and an upper bound given for the dimension, specifying vector x as starting at the fifth element of X means that the argument is specified as X(7).

Most of the examples shown in this manual assume that the lower bound in each dimension of an array is 1. Therefore, the lower bound is not specified, and the value of the upper bound is the number of elements in that dimension. So, a declaration of X(50) means X has 50 elements.

14.2.1.3 Stride of a Vector

The spacing parameter, called the increment or stride, indicates how to move from the starting point through the array to select the vector elements from the array. The increment is specified by an argument in the DXML subprogram argument list, such as the **incx** argument.

The vector elements are stored in the array in the order $x_1, x_2, ..., x_n$. An increment of 1 indicates that the vector elements are contiguous in the array.

14.2.1.4 Selecting Vector Elements from an Array

DXML VLIB routines use the stride to select elements from the array to construct the vector composed of these elements. The stride associates consecutive elements of the vector with equally spaced elements of the array.

14.2.2 Storing a Vector in an Array

Suppose X is a real one-dimensional array of $ndim$ elements. Let vector x have length n and let $incx$ be the increment used to access the elements of vector x whose components x_i, $i = 1, \ldots, n$, are stored in X.

If $incx > 0$, and if the first element of the vector is specified at the first element of the array, then x_i is stored in the array location as shown in (14–1):

$$\mathrm{X}(1 + (i - 1) * incx) \tag{14–1}$$

Therefore, $ndim$, the number of elements in the array, should satisfy the condition shown in (14–2):

$$ndim \geq 1 + (n - 1) * |incx| \tag{14–2}$$

For the general case where the first element of the vector in the array is at the point X(BP) rather than at the first element of the array, (14–3) can be used to find the position of each vector element x_i in a one-dimensional array.

For $incx \neq 0$, the position of x_i is as follows:

$$X(\mathrm{BP} + (i - 1) * incx) \tag{14–3}$$

For example, suppose that BP = 3, $ndim = 20$, and $n = 5$. Then a value of $incx = 2$ implies that x_1, x_2, x_3, x_4, and x_5 are stored in array elements X(3), X(5), X(7), X(9), and X(11). Using BP = 11 and $incx = -2$ would mean that x_1, x_2, x_3, x_4, x_5 were stored in X(11), X(9), X(7), X(5), X(3).

14.3 Naming Conventions

Table 14–1 shows the characters used in the names of the VLIB subprograms and what the characters mean.

Table 14–1 Naming Conventions: VLIB Subprograms

Character Group	Mnemonic	Meaning
First group	V	Operates on a vector.
Second group	A combination of letters at the end such as SIN or RECIP	Type of computation such as sine (SIN) of a vector or reciprocal (RECIP) of the elements of a vector.

For example, the name VSQRT is the subprogram for computing the square-root of the elements of a vector. All VLIB routines accept double-precision input arrays and return double-precision output arrays.

14.4 Summary of VLIB Subprograms

Table 14–2 summarizes the VLIB subprograms.

Table 14–2 Summary of VLIB Subprograms

Subprogram Name	Operation
VCOS	Calculates, in double-precision arithmetic, the cosine of the elements of a real vector.
VCOS_SIN	Calculates, in double-precision arithmetic, the sine and cosine of the elements of a real vector.
VEXP	Calculates, in double-precision arithmetic, the exponential of the elements of a real vector.
VLOG	Calculates, in double-precision arithmetic, the natural logarithm of the elements of a real vector.
VRECIP	Calculates, in double-precision arithmetic, the reciprocal of the elements of a real vector.
VSIN	Calculates, in double-precision arithmetic, the sine of the elements of a real vector.
VSQRT	Calculates, in double-precision arithmetic, the square root of the elements of a real vector.

14.5 Calling Subprograms

The VLIB subprograms consist of only subroutines.

14.6 Argument Conventions

The VLIB subprograms use a list of arguments to specify the requirements and control the result of the subprogram. All arguments are required. The argument list is in the same order for each subprogram:

- Arguments that describe the input and output vectors
  The following arguments describe a vector:
  - The arguments **x**, **y**, and **z** define the location of the vectors x, y, and z in the array. In the usual case, the argument **x** specifies the location in the array as X(1), but the location can be specified at any other element of the array. An array can be much larger than the vector that it contains.
  - The arguments **incx**, **incy**, and **incz** provide the increment between the locations of the elements of the vector x, vector y, and vector z, respectively.
- Arguments that define the number of elements to process
  The **n** argument specifies the number of elements to process. If $n \leq 0$, the output vector is unchanged.

14.7 Error Handling

Where applicable, the elements of the input vector, to the VLIB routines, are checked for the possibility that a later arithmetic exception will occur. Nonfinite operands will trap within the routine. Other situations such as finite inputs that are illegal or exception causing inputs to the corresponding RTL routine are typically caught by detecting the offending input argument, and then calling the corresponding RTL routine with the offending argument. Thus, essentially the same exception behavior as with an RTL call is preserved.

14.8 A Look at a VLIB Subprogram

To understand the meaning of the arguments, consider the subroutine VSQRT. VSQRT computes the square root of a real (n-element) vector x, and the result is returned in the vector y. VSQRT has the arguments **x**, **incx**, **y**, **incy** and **n**.

For example, suppose that arrays X and Y are declared as follows:

```
REAL*8 X(-10:10), Y(41)
```

Then, the statements:

```
INCX = 1
INCY = 2
N = 21
CALL VSQRT(X,INCX,Y,INCY,N))
```

yield the following results:

```
Y(1)  = SQRT(X(-10))
Y(3)  = SQRT(X(-9)))
Y(5)  = SQRT(X(-8))
      .
      .
      .
Y(39) = SQRT(X(9))
Y(41) = SQRT(X(10))
```

This call to routine VSQRT obtains the same results as the following Fortran code segment:

```
DO I = 1, 41, 2
   Y(I) = SQRT( X( (I+1)/2 - 11) )
END DO
```

The argument **x** specifies the array X with 21 elements and specifies X(-10) as the location of the vector x whose elements are embedded in X. Since **n** = 21, the vector also has 21 elements. The length of the array X is the same as the length of the vector x. The value of the argument **incx** = 1 specifies that the vector elements are contiguous in the array. Since **incy** is 2, the square root of each element of the array X is stored in array Y, beginning at Y(1), in the locations Y(1), Y(3), Y(5), and so on.

VLIB Routines

This section provides descriptions of the VLIB subprograms.

VCOS
Vector Cosine

Format

VCOS (x, incx, y, incy, n)

Arguments

x
real*8
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
On exit, **incx** is unchanged.

y
real*8
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$.
On exit, if $n \leq 0$, **y** is unchanged. If $n > 0$, **y** is overwritten; y_i is replaced by $\cos(x_i)$.

incy
integer*4
On entry, the increment for the array Y.
On exit, **incy** is unchanged.

n
integer*4
On entry, the number of elements to process.
On exit, **n** is unchanged.

Description

The VCOS function computes the cosine of n elements of a vector as follows:

$$y_i \leftarrow \cos x_i$$

where x and y are vectors.

Example

```
INTEGER*4 N, INCX, INCY
REAL*8 X(20), Y(20)
INCX = 1
INCY = 1
N = 20
CALL VCOS(X,INCX,Y,INCY, N)
```

This Fortran code shows how the cosine of all elements of the real vector x is obtained and set equal to the corresponding elements of the vector y.

VCOS_SIN
Vector Cosine and Sine

Format

VCOS_SIN (x, incx, y, incy, z, incz, n)

Arguments

x
real*8
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
On exit, **incx** is unchanged.

y
real*8
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$.
On exit, if $n \leq 0$, **y** is unchanged. If $n > 0$, **y** is overwritten; y_i is replaced by $\cos(x_i)$.

incy
integer*4
On entry, the increment for the array Y.
On exit, **incy** is unchanged.

z
real*8
On entry, a one-dimensional array Z of length at least $(1 + (n - 1) * |incz|)$.
On exit, if $n \leq 0$, **z** is unchanged. If $n > 0$, **z** is overwritten; z_i is replaced by $\sin(x_i)$.

incz
integer*4
On entry, the increment for the array Z.
On exit, **incz** is unchanged.

n
integer*4
On entry, the number of elements to process.
On exit, **n** is unchanged.

Description

The VCOS_SIN function computes the cosine and sine of n elements of a vector as follows:

$$y_i \leftarrow \cos x_i$$

$$z_i \leftarrow \sin x_i$$

where x, y and z are vectors. If both the sine and cosine of a vector are required, this routine is faster than calling VCOS and VSIN separately.

Example

```
INTEGER*4 N, INCX, INCY
REAL*8 X(20), Y(20), Z(20)
INCX = 1
INCY = 1
INCZ = 1
N = 20
CALL VCOS_SIN(X,INCX,Y,INCY,Z,INCZ, N)
```

This Fortran code shows how the cosine and sine of all elements of the real vector x is obtained and set equal to the corresponding elements of the vectors y and z, respectively.

VEXP
Vector Exponential

Format

VEXP (x, incx, y, incy, n)

Arguments

x
real*8
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
On exit, **incx** is unchanged.

y
real*8
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$.
On exit, if $n \leq 0$, **y** is unchanged. If $n > 0$, **y** is overwritten; y_i is replaced by $\exp(x_i)$.

incy
integer*4
On entry, the increment for the array Y.
On exit, **incy** is unchanged.

n
integer*4
On entry, the number of elements to process.
On exit, **n** is unchanged.

Description

The VEXP function computes the exponential of *n* elements of a vector as follows:

$$y_i \leftarrow \exp x_i$$

where x and y are vectors.

Example

```
INTEGER*4 N, INCX, INCY
REAL*8 X(20), Y(20)
INCX = 1
INCY = 1
N = 20
CALL VEXP(X,INCX,Y,INCY, N)
```

This Fortran code shows how the exponential of all elements of the real vector x is obtained and set equal to the corresponding elements of the vector y.

VLOG
Vector Logarithm

Format

VLOG (x, incx, y, incy, n)

Arguments

x
real*8
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
On exit, **incx** is unchanged.

y
real*8
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$.
On exit, if $n \leq 0$, **y** is unchanged. If $n > 0$, **y** is overwritten; y_i is replaced by $\log(x_i)$.

incy
integer*4
On entry, the increment for the array Y.
On exit, **incy** is unchanged.

n
integer*4
On entry, the number of elements to process.
On exit, **n** is unchanged.

Description

The VLOG function computes the natural logarithm of *n* elements of a vector as follows:

$$y_i \leftarrow \log x_i$$

where x and y are vectors.

Example

```
INTEGER*4 N, INCX, INCY
REAL*8 X(20), Y(20)
INCX = 1
INCY = 1
N = 20
CALL VLOG(X,INCX,Y,INCY, N)
```

This Fortran code shows how the natural logarithm of all elements of the real vector x is obtained and set equal to the corresponding elements of the vector y.

VRECIP
Vector Reciprocal

Format

VRECIP (x, incx, y, incy, n)

Arguments

x
real*8
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
On exit, **incx** is unchanged.

y
real*8
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$.
On exit, if $n \leq 0$, **y** is unchanged. If $n > 0$, **y** is overwritten; y_i is replaced by $\frac{1}{x_i}$.

incy
integer*4
On entry, the increment for the array Y.
On exit, **incy** is unchanged.

n
integer*4
On entry, the number of elements to process.
On exit, **n** is unchanged.

Description

The VRECIP function computes the reciprocal of *n* elements of a vector as follows:

$$y_i \leftarrow \frac{1}{x_i}$$

where x and y are vectors.

Example

```
INTEGER*4 N, INCX, INCY
REAL*8 X(20), Y(20)
INCX = 1
INCY = 1
N = 20
CALL VRECIP(X,INCX,Y,INCY, N)
```

This Fortran code shows how the reciprocal of all elements of the real vector x is obtained and set equal to the corresponding elements of the vector y.

VSIN
Vector Sine

Format

VSIN (x, incx, y, incy, n)

Arguments

x
real*8
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
On exit, **incx** is unchanged.

y
real*8
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$.
On exit, if $n \leq 0$, **y** is unchanged. If $n > 0$, **y** is overwritten; y_i is replaced by $\sin(x_i)$.

incy
integer*4
On entry, the increment for the array Y.
On exit, **incy** is unchanged.

n
integer*4
On entry, the number of elements to process.
On exit, **n** is unchanged.

Description

The VSIN function computes the sine of *n* elements of a vector as follows:

$$y_i \leftarrow \sin x_i$$

where x and y are vectors.

Example

```
INTEGER*4 N, INCX, INCY
REAL*8 X(20), Y(20)
INCX = 1
INCY = 1
N = 20
CALL VSIN(X,INCX,Y,INCY, N)
```

This Fortran code shows how the sine of all elements of the real vector x is obtained and set equal to the corresponding elements of the vector y.

VSQRT
Vector Square Root

Format

VSQRT (x, incx, y, incy, n)

Arguments

x
real*8
On entry, a one-dimensional array X of length at least $(1 + (n - 1) * |incx|)$, containing the elements of the vector x.
On exit, **x** is unchanged.

incx
integer*4
On entry, the increment for the array X.
On exit, **incx** is unchanged.

y
real*8
On entry, a one-dimensional array Y of length at least $(1 + (n - 1) * |incy|)$.
On exit, if $n \leq 0$, **y** is unchanged. If $n > 0$, **y** is overwritten; y_i is replaced by $\sqrt{x_i}$.

incy
integer*4
On entry, the increment for the array Y.
On exit, **incy** is unchanged.

n
integer*4
On entry, the number of elements to process.
On exit, **n** is unchanged.

Description

The VSQRT function computes the square root of *n* elements of a vector as follows:

$$y_i \leftarrow \sqrt{x_i}$$

where x and y are vectors.

Example

```
INTEGER*4 N, INCX, INCY
REAL*8 X(20), Y(20)
INCX = 1
INCY = 1
N = 20
CALL VSQRT(X,INCX,Y,INCY, N)
```

This Fortran code shows how the square root of all elements of the real vector x is obtained and set equal to the corresponding elements of the vector y.

15

Using Random Number Generator Subprograms

DXML provides four random number generator (RNG) subprograms and two auxiliary input subprograms for parallel applications. This chapter provides information about the following topics:

- Standard Uniform RNG Subprograms (Section 15.2)
- Long Period Uniform RNG Subprogram (Section 15.3)
- Normally Distributed RNG Subprogram (Section 15.4)
- Input Subprograms for Parallel Applications Using RNG Subprograms (Section 15.5)
- Summary of RNG Subprograms (Section 15.6)
- Error Handling (Section 15.7)

The reference descriptions of the RNG subprograms are at the end of this chapter.

15.1 Introduction

RNGs are an important part of many simulation programs and test procedures. DXML provides the following RNG subprograms:

- Three subprograms that generate uniform[0,1] random number distributions using algorithms based on the following:
  - — Multiplicative generators — See Section 15.2 for a description of the RAN16807 subprogram.
  - — Linear congruential generators — See Section 15.2 for a description of the RAN69069 subprogram.
  - — Combined multiplicative generators — See Section 15.3 for a description of the RANL subprogram.
- One subprogram that generates normally distributed (N(0,1)) random numbers using a sum-type algorithm based on the central limit theorem — see Section 15.4 for a description of the RANL_NORMAL subprogram.
- Two subprograms that generate input for two other DXML RNG subprograms when they are used in parallel computing applications. Both input subprograms use a repeated squaring algorithm — see Section 15.5 for descriptions of the RANL_SKIP2 and RANL_SKIP64 subprograms.

15.2 Standard Uniform RNG Subprograms

DXML provides the RAN16807 and the RAN69069 subprograms to generate full period (m or $m-1$), sequences of uniform random numbers. Both subprograms use a single precision, function call interface. DXML provides these RNGs since they are perhaps the two most commonly used 32-bit generators.

The RAN16807 subprogram uses an algorithm that corresponds to the "minimal standard generator" recommended by Park and Miller. For further information see Section A.8. The algorithm is based on a multiplicative generator of the form:

$s = a * s \mod m$, with $s \geq 1$, and $m = 2**31 - 1$.
$x = s/m$

The RAN69069 subprogram uses an algorithm that has been used in Digital run-time libraries. The algorithm is based on a linear congruential generator of the form:

$s = a * s + c \mod m$, with $m = 2**32$.
$x = s/m$

For a further discussion of this algorithm, refer to Knuth in Section A.8.

15.3 Long Period Uniform RNG Subprogram

DXML provides the RANL subprogram to generate very long period, uniform random numbers. The RANL subprogram implements the combined multiplicative algorithm introduced by L'Ecuyer. This algorithm uses two 32-bit seeds and combines two separate generators to yield a very long period generator.

Here is a brief summary of the L'Ecuyer algorithm:

- The initial seeds $s1, s2$ are user input, with $1 \leq s1 \leq m1$, $1 \leq s2 \leq m2$. (See $m1, m2$ values in the following paragraphs.)
- In each step, updated values of $s1, s2$ are used to produce the next single precision uniform random number u according to the following:

  $s1 = a1 * s1 \pmod{m1}$ $\quad m1 = 2147483563,\ a1 = 40014$
  $s2 = a2 * s2 \pmod{m2}$ $\quad m2 = 2147483399,\ a2 = 40692$

  The algorithm then calculates $z = s1 - s2 \pmod{(m1-1)}$, and if z is 0, puts $z = m1 - 1$.
- Finally, the algorithm returns with $u = z/m1$.

The period can be shown to be $(m1-1)(m2-1)/2$.

DXML provides a subroutine call interface that can return a vector v(1),...,v(n) of outputs. The vector form is very useful when speed is paramount.

For more information about the L'Ecuyer algorithm, see Section A.8.

For parallel applications using the RANL subprogram, DXML provides two auxiliary, input subprograms that generate nonoverlapping streams of independent random numbers. See Section 15.5 for more information.

15.4 Normally Distributed RNG Subprogram

DXML provides the RANL_NORMAL subprogram to generate random normal N(0,1) numbers. The RANL_NORMAL subprogram uses an algorithm based on the central limit, which uses the following sum:

$$s = x1 + x2 + ... + x12 - 6.0$$

to approximate a normally distributed variate.

The number 12 is used, because the variance of a uniform[0,1] variable is 1/12. Summing 12 uniform variables and subtracting their mean should yield an N(0,1) variate. The RANL_NORMAL subprogram generalizes this procedure to an arbitrary number of summands and then uses scaling to get back to an N(0,1) result. The RANL_NORMAL subprogram overcomes the common drawback of sum-type algorithms, in that it uses DXML's vector call interface to the RANL subprogram's uniform[0,1] random number generator. Consequently, the RANL_NORMAL subprogram can obtain, for example, 12 uniform random numbers per subroutine call.

For parallel applications using the RANL_NORMAL subprogram, DXML provides two auxiliary, input subprograms that generate nonoverlapping streams of independent random numbers. See Section 15.5 for more information.

15.5 Input Subprograms for Parallel Applications Using RNG Subprograms

For parallel applications using either the RANL or the RANL_NORMAL subprograms, DXML provides two auxiliary, input subprograms RANL_SKIP2 and RANL_SKIP64. Both subprograms skip over a user specified number of seeds. The RANL_SKIP2 subprogram skips a number $2**d$ $(d \geq 0)$ of seeds. The RANL_SKIP64 subprogram skips an arbitrary 64-bit number d $(d \geq 0)$ of seeds. These subprograms provide a way to generate nonoverlapping streams of independent random numbers. Both subprograms use a well-known, repeated squaring algorithm for computing:

$$a**(2**k)*s \mod m$$

This algorithm for finding the next seed skips over $2**k$ steps of the multiplicative algorithm:

$$s = a*s \mod m$$

The RANL_SKIP64 subprogram makes it convenient to generate nonoverlapping, random sequences. This is useful in parceling out the same set of random numbers to different numbers of processors in a multiprocessor environment.

15.6 Summary of RNG Subprograms

Table 15–1 gives a description of each RNG subprogram.

Table 15–1 Summary of RNG Subprograms

Subprogram Name	Description
RANL	Generates a vector of uniform[0,1] random numbers.
RANL_SKIP2	Generates starting seeds for parallel, independent streams of random numbers. Auxiliary subprogram for parallel applications using RANL or RANL_NORMAL.
RANL_SKIP64	Generates starting seeds for parallel, independent streams of random numbers by skipping forward a given number d of seeds. Auxiliary subprogram for parallel applications using RANL or RANL_NORMAL.
RANL_NORMAL	Generates a vector of N(0,1) normally distributed random numbers.
RAN69069	Generates single precision uniform[0,1] random numbers using $a = 69069$ in the linear multiplicative algorithm.
RAN16807	Generates single precision uniform[0,1] random numbers using $a = 16807$ in the "minimal standard" multiplicative generator.

15.7 Error Handling

The RNG subprograms assume that input parameters are correct and provide no feedback when errors occur.

Random Number Generator Subprograms

This section provides descriptions of the random number generator (RNG) subroutines.

RANL
Random Number Generator Based on L'Ecuyer Method

Format

RANL (s1, s2, v, n)

Arguments

s1
integer*4
On entry, **s1**≥1, the first part of the two-integer seed.
On exit, **s1** is changed according to **n** steps of the L'Ecuyer method.

s2
integer*4
On entry, **s2**≥1, the second part of the two-integer seed.
On exit, **s2**, is changed according to **n** steps of the L'Ecuyer method.

v
real*4
On entry, a one-dimensional vector of **n** elements. **v** can be a scalar variable if **n**=1.
On exit, contains *n* pseudorandom uniform[0,1] random numbers generated according to the L'Ecuyer algorithm.

n
integer*4
On entry, a positive integer specifying the number of random numbers to store in v(1),...,v(**n**).
On exit, unchanged.

Description

The RANL routine returns a vector of uniform[0,1] random numbers. After you give arguments **s1** and **s2** initial values, you need not change these, except to restart the sequence or to skip to a new subsequence.

For parallel applications using the RANL routine, DXML provides two auxiliary input programs. Refer to the descriptions of RANL_SKIP64 and RANL_SKIP2.

Example

```
c     Monte Carlo Method
      integer n,ndim,m
      parameter ( m = 10 )
      integer*4 s1,s2,i,k
      real*4 pt(m),vol,sum,r,vol_all
      print*,'nr. pts to use: '
      read*,n
      print*,' n= ',n
      print*,'       vol(10-d sphere)    vol(10-d cube) '
      sum=0.0
      s1=12345
      s2=67890
      do k=1,n
         call ranl(s1,s2,pt ,m)
          r=0.0
          do i=1,m
          r=r+pt(i)*pt(i)
          end do
      if(r.le.1.0)sum=sum+1.0
      end do
      vol = sum/n
      vol_all = 2**m*vol
      write(6,900)vol_all,2.0**m
 900  format(1x,2x,f14.2,4x,f14.0)
      end
```

This example computes the volume of a 10-dimensional sphere, using a Monte Carlo technique. The program generates n points in the 10-dimensional unit quadrant and then counts the number of these n points inside the 10-dimensional unit sphere to yield a value for sum. Next, the program divides sum, by the total n points inside the unit quadrant to approximate the volume of the spherical quadrant, vol. There are $2 * *10$ such quadrants. Therefore, the program multiplies vol by $2 * *10$ to approximate the total volume, vol_all, of the 10-dimensional sphere.

The exact value of vol_all is 2.55 to 2 decimals. With 10 million points, this program yields 2.56. This is in marked contrast to the case in lower dimensions where the volume of the unit sphere is nearly equal to the volume of the containing cube.

See also the example for RANL_SKIP2.

RANL_SKIP2
Routine to Skip Forward a Given Number of Seeds for the RANL and RANL_NORMAL Random Number Generators

Format

RANL_SKIP2 (d, s1, s2, s1_new, s2_new)

Arguments

d
integer*4
On entry, an integer **d**>0 specifying the number $2**\mathbf{d}$ of seeds to skip starting at (s1,s2) in the L'Ecuyer algorithm.

s1
integer*4
On entry, a starting seed **s1**$\geq$1.

s2
integer*4
On entry, a starting seed **s2**$\geq$1.

s1_new
integer*4
On exit, a new starting seed $2**\mathbf{d}$ iterations away from **s1**.

s2_new
integer*4
On exit, a new starting seed $2**\mathbf{d}$ iterations away from **s2**.

Description

The RANL_SKIP2 routine is used to get starting seeds for parallel, independent streams of random numbers. The new starting seeds are computed using the following algorithms:

$$\mathbf{s1\_new} = a1 * *(2 * *d) * s1 \mod m1$$
$$\mathbf{s2\_new} = a2 * *(2 * *d) * s2 \mod m2$$

Where $a1, a2, m1, m2$ are the constants defining the L'Ecuyer method.

Example

```
      integer nprocs,n,hop
      parameter (nprocs=4)
      parameter (n=16384,hop=14)
      integer*4 j,k
      real*4 v(n,nprocs)
      integer*4 s1val(nprocs),s2val(nprocs)
      real*4 sum1(nprocs)
c     get seeds (2**hop apart) for separate streams
      s1val(1)=1
      s2val(1)=1
      do j=2,nprocs
      call ranl_skip2(hop,s1val(j-1),s2val(j-1),s1val(j),s2val(j))
      end do
```

```
c     parallel calls to ranl can be done, for example, with KAP directives:
C*$*  ASSERT CONCURRENT CALL
C*$*  ASSERT DO (CONCURRENT)
      do j=1,nprocs
         call ranl(s1val(j),s2val(j),v(1,j),n)
         sum1(j)=0.0
         do k=1,n
            sum1(j)=sum1(j)+v(k,j)
         end do
      end do

      print*,' per-stream averages'
      do j=1,nprocs
         print*,sum1(j)/n
      end do

      end
```

This example calls RANL_SKIP2 to set up separate seeds for 4 streams, (nprocs=4), from RANL. It computes the averages per stream.

RANL_SKIP64
Routine to Skip Forward a Given Number d of Seeds for the RANL and RANL_NORMAL Random Number Generators

Format

RANL_SKIP64 (d, s1, s2, s1_new, s2_new)

Arguments

d
integer*8
On entry, an integer **d** $\geq$0 specifying the number **d** of seeds to skip starting at (s1,s2) in the L'Ecuyer algorithm.

s1
integer*4
On entry, a starting seed **s1**$\geq$1.

s2
integer*4
On entry, a starting seed **s2**$\geq$1.

s1_new
integer*4
On exit, a new starting seed **d** iterations away from **s1**.

s2_new
integer*4
On exit, a new starting seed **d** iterations away from **s2**.

Description

The RANL_SKIP64 routine is used to get starting seeds for parallel, independent streams of random numbers. The new starting seeds are computed using the following algorithms:

$$\mathbf{s1\_new} = a1 * *d * s1 \mod m1$$
$$\mathbf{s2\_new} = a2 * *d * s2 \mod m2$$

Where $a1, a2, m1, m2$ are the constants defining the L'Ecuyer method.

Example

```
integer nprocs,n
integer*8 hop
parameter (nprocs=4)
parameter (n=16384,hop=1000000)
integer*4 j,k,nsum
real*4 v(n,nprocs)
integer*4 s1val(nprocs),s2val(nprocs)
real*4 sum1(nprocs)
```

```
c     get seeds (hop apart) for separate streams
      s1val(1)=1
      s2val(1)=1
      nsum=12
      do j=2,nprocs
      call ranl_skip64(hop,s1val(j-1),s2val(j-1),s1val(j),s2val(j))
      end do

c     parallel calls to ranl_normal can be done, for example, with KAP directives:
C*$*  ASSERT CONCURRENT CALL
C*$*  ASSERT DO (CONCURRENT)
      do j=1,nprocs
         call ranl_normal(s1val(j),s2val(j),nsum,v(1,j),n)
         sum1(j)=0.0
         do k=1,n
            sum1(j)=sum1(j)+v(k,j)
         end do
      end do

      print*,' per-stream averages'
      do j=1,nprocs
         print*,sum1(j)/n
      end do

      end
```

This example calls RANL_SKIP64 to set up separate seeds for 4 streams, (nprocs=4), from RANL_NORMAL. It computes the averages per stream.

RANL_NORMAL
Routine to Generate Normally Distributed Random Numbers Using Summation of Uniformly Distributed Random Numbers

Format

RANL_NORMAL (s1, s2, nsum, vnormal, n)

Arguments

s1
integer*4
On entry, **s1**≥1, the first part of the two-integer seed.
On exit, **s1** is changed according to **n**∗**nsum** steps of the L'Ecuyer method.

s2
integer*4
On entry, **s2**≥1, the second part of the two-integer seed.
On exit, **s2** is changed according to **n**∗**nsum** steps of the L'Ecuyer method.

nsum
integer*4
On entry, number of uniform[0,1] numbers to sum to get each N(0,1) output.

vnormal
real*4
On exit, a vector **vnormal**(1),...,**vnormal**(**n**) of N(0,1) normally distributed numbers.

n
integer*4
On entry **n**≥1 specifies the number of N(0,1) results to return in **vnormal**(1),...,**vnormal**(**n**).

Description

The RANL_NORMAL routine returns a vector of N(0,1) normally distributed random numbers.

For parallel applications using the RANL_NORMAL routine, DXML provides two auxiliary input programs. Refer to the descriptions of RANL_SKIP64 and RANL_SKIP2.

Example

See the example for RANL_SKIP64.

RAN69069
Routine to Generate Single Precision Random Numbers Using a=69069 and m=2**32

Format

RAN69069 (s)

Function Value

ran69069
real*4

The uniform[0,1] value returned.

Arguments

s
integer*4
On input, a seed **s** being set initially or left unchanged from a previous iteration. On exit, the updated seed.

Description

The RAN69069 routine computes updated seeds using the linear multiplicative algorithm as follows:

$$\mathbf{s} = 69069 * s + 1, \quad mod\ 2 * *32$$

Returns $s * 2.0 * *(-32)$, as its uniform[0,1] output.

Example

```
      integer ix,iy,i,j,iseed,nsteps,nwalks
      real*4 x
c     random walk: go N E S W each with probability 0.25
      nsteps = 1000000
      nwalks = 10
      iseed = 1234

      do j=1,nwalks
      ix=0
      iy=0
      do i = 1, nsteps
      x=ran69069(iseed)
      if(x.le.0.25)then
         ix=ix+1
      else if(x.le.0.5)then
         iy=iy+1
      else if(x.le.0.75)then
         ix=ix-1
      else
         iy=iy-1
      end if
      end do
```

```
print*,'final position ',ix,iy
end do
end
```

This example simulates a random walk using RAN69069. A particle starts at (0,0) and proceeds N, E, S, or W. with probability 0.25 each.

RAN16807
Routine to Generate Single Precision Random Numbers Using a=16807 and m=2**31-1

Format

RAN16807 (s)

Function Value

ran16807
real*4

The uniform[0,1] value returned.

Arguments

s
integer*4
On entry, the seed **s**$\geq$1.
On exit, the updated seed.

Description

The RAN16807 routine implements the "minimal standard" or Lehmer multiplicative generator to compute updated seeds using $a = 16807$, $m = 2**31-1$, as follows:

$$\mathbf{s} = 16807 * s \mod 2**31 - 1$$

Returns $s * (1.0/m)$ as its uniform[0,1] output.

Example

```
      integer ix,iy,i,j,iseed,nsteps,nwalks
      real*4 x
c     random walk: go N E S W each with probability 0.25
      nsteps = 1000000
      nwalks = 10
      iseed = 1234

      do j=1,nwalks
      ix=0
      iy=0
      do i = 1, nsteps
      x=ran16807(iseed)
      if(x.le.0.25)then
         ix=ix+1
      else if(x.le.0.5)then
         iy=iy+1
      else if(x.le.0.75)then
         ix=ix-1
      else
         iy=iy-1
      end if
      end do
```

```
print*,'final position ',ix,iy
end do
end
```

This example simulates a random walk using RAN16807. A particle starts at (0,0) and proceeds N, E, S, or W. with probability 0.25 each.

16
Using Sort Subprograms

This chapter provides information about DXML quick sort and general purpose sort subprograms as follows:

- Quick Sort Subprograms (Section 16.1)
- General Purpose Sort (Section 16.2)
- Naming Conventions (Section 16.3)
- Summary of Sort Subprograms (Section 16.4)
- Error Handling (Section 16.5)

The reference descriptions of the sort subprograms are at the end of this chapter.

16.1 Quick Sort Subprograms

The _SORTQ and _SORTQX subprograms use a quick sort algorithm to sort a vector of data. In the case of _SORTQX, the data vector is indexed. Both subprograms implement the quick sort algorithm by recursing until the partition size is less than 16. At that point, _SORTQ uses a simple replacement sort to sort the elements of the partition, while _SORTQX uses a modified insertion sort to sort the elements of the partition. Subprogram _SORTQX permutes only elements of the index vector, leaving the data vector unchanged.

16.2 General Purpose Sort Subprograms

The GEN_SORT subprogram is a general purpose, in memory, sort routine that uses a radix algorithm to sort the data. The GEN_SORTX subprogram is a general purpose, in memory, indexed sort routine that uses an indexed radix algorithm to sort the data.

16.3 Naming Conventions

Table 16–1 shows the characters used in the names of the quick sort subprograms and what the characters mean.

Table 16–1 Naming Conventions: _SORTQ_ Subprograms

Character Group	Mnemonic	Meaning
First group	I	Integer
	S	Single-precision real
	D	Double-precision real
Second group	SORTQ	Quick sort algorithm
Third group	X	Refers to indexed quick sort

For example, the name DSORTQX is the subprogram for performing an indexed quick sort on a vector of double-precision data.

16.4 Summary of Sort Subprograms

Table 16–2 gives a description of each sort subprogram.

Table 16–2 Summary of Sort Subprograms

Subprogram Name	Description
ISORTQ	Quick sorts the elements of a vector of integer data.
SSORTQ	Quick sorts the elements of a vector of single-precision data.
DSORTQ	Quick sorts the elements of a vector of double-precision data.
ISORTQX	Performs an indexed quick sort of a vector of integer data.
SSORTQX	Performs an indexed quick sort of a vector of single-precision data.
DSORTQX	Performs an indexed quick sort of a vector of double-precision data.
GEN_SORT	Performs a general purpose, in memory, sort routine.
GEN_SORTX	Performs a general purpose, in memory, sort of an indexed vector of data.

16.5 Error Handling

The sort subprograms assume that input parameters are correct and provide no feedback when errors occur.

Sort Subprograms

This section provides descriptions of the Sort subprograms.

ISORTQ, SSORTQ, DSORTQ
Sorts the Elements of a Vector

Format

{I,S,D}SORTQ (order, n, x, incx)

Arguments

order
character*4
On entry, **order** specifies the operation to be performed as follows:

If **order** = 'A' or 'a', **x** is sorted in ascending sequence.
If **order** = 'D' or 'd', **x** is sorted in descending sequence.

On exit, **order** is unchanged.

n
integer*4
On entry, the length of vector **x**.
On exit, **n** is unchanged.

x
integer*4 | real*4 | real*8
On entry, a length **n** vector of data to be sorted.
On exit, **x** is overwritten by a length **n** vector of sorted data.

incx
integer*4
On entry, **incx** specifies the distance between elements of vector **x**. The argument **incx** must be positive.
On exit, **incx** is unchanged.

Description

The _SORTQ routines sort a vector of data using the quick sort algorithm. Data is sorted in ascending order if **order** is 'A' or 'a' and in descending order if **order** is 'D' or 'd'. The quick sort algorithm is implemented by recursing until the partition size is less than 16. At that point, a simple replacement sort is used to sort the elements of the partition.

Example

```
REAL*4 DATA( 100 )
N = 100
CALL SSORTQ( 'A',N,DATA,1 )
```

This Fortran code sorts a 100 element single real vector.

ISORTQX, SSORTQX, DSORTQX
Performs an Indexed Sort of a Vector

Format

{I,S,D}SORTQX (order, n, x, incx, index)

Arguments

order
character*1
On entry, **order** specifies the operation to be performed as follows:

If **order** = 'A' or 'a', **x** is sorted in ascending sequence.
If **order** = 'D' or 'd', **x** is sorted in descending sequence.

On exit, **order** is unchanged.

n
integer*4
On entry, the length of vector **x**.
On exit, **n** is unchanged.

x
integer*4 | real*4 | real*8
On entry, a length **n** vector of data to be sorted.
On exit, **x** is unchanged.

incx
integer*4
On entry, **incx** specifies the distance between elements of vector **x**. The argument **incx** must be positive.
On exit, **incx** is unchanged.

index
integer*4
On entry, the content of **index** is ignored.
On exit, **index** contains a permuted vector of indices that may be used to access data vector **x** in the sorted order specified by **order**.

Description

The _SORTQX routines sort an indexed vector of data using the quick sort algorithm. Data is sorted in ascending order if **order** is 'A' or 'a' and in descending order if **order** is 'D' or 'd'. The quick sort algorithm is implemented by recursing until the partition size is less than 16. At that point, a modified insertion sort is used to sort the elements of the partition. Only elements of the index vector are permuted, the data vector is left unchanged.

Example

```
REAL*4 DATA( 100 ),INDEX( 100 )
N = 100
CALL SSORTQX( 'A',N,DATA,1,INDEX )
DO I=1,N
   PRINT *,DATA( INDEX(I) )
ENDDO
```

This Fortran code sorts a 100 element single real vector and prints its contents in sorted order.

GEN_SORT
Sorts the Elements of a Vector

Format

GEN_SORT (order, type, size, n, x, incx, y, incy)

Arguments

order
character*1
On entry, **order** specifies the operation to be performed as follows:

If **order** = 'A' or 'a', **x** is sorted in ascending sequence.
If **order** = 'D' or 'd', **x** is sorted in descending sequence.

On exit, **order** is unchanged.

type
character*1
On entry, **type** specifies the data type of vectors **x** and **y**. The following types are valid:

If **type** = 'B' or 'b', binary (unsigned integer)
If **type** = 'C' or 'c', character string
If **type** = 'I' or 'i', integer (signed)
If **type** = 'L' or 'l', logical - Fortran .TRUE. or .FALSE.
If **type** = 'R' or 'r', IEEE floating point
If **type** = 'V' or 'v', VAXG floating point

On exit, **type** is unchanged.

size
integer*4
On entry, **size** specifies the size, in bytes, of each element of data vectors **x** and **y**. Valid combinations of **type** and **size** include:

If **type** = 'B' or 'b' - **size** = { 1,2,4,8 }
If **type** = 'C' or 'c' - **size** = { 0 < **size** < 65536 }
If **type** = 'I' or 'i' - **size** = { 1,2,4,8 }
If **type** = 'L' or 'l' - **size** = { 1,2,4,8 }
If **type** = 'R' or 'r' - **size** = { 4,8,16 }
If **type** = 'V' or 'v' - **size** = { 4,8 }

On exit, **size** is unchanged.

n
integer*4
On entry, the length of the **x** and **y** vectors.
On exit, **n** is unchanged.

x
data vector
On entry, a length **n** vector of data to be sorted. Each element of vector **x** is of **type** and **size** specified.
On exit, vector **x** is unchanged, unless it overlaps vector **y**.

incx
integer*4
On entry, **incx** specifies the distance between elements of vector **x**. The argument **incx** must be positive.
On exit, **incx** is unchanged.

y
data vector
On entry, **y** is ignored.
On exit, vector **y** is overwritten with sorted data. Each element of vector **y** is of **type** and **size** specified.

incy
integer*4
On entry, vector **incy** specifies the distance between elements of vector **y**. The argument **incy** must be positive.
On exit, **incy** is unchanged.

Description

GEN_SORT is a general purpose, in memory, sort routine. GEN_SORT accepts the following Fortran data types:

INTEGER*1, INTEGER*2, INTEGER*4, INTEGER*8
LOGICAL*1, LOGICAL*2, LOGICAL*4, LOGICAL*8
REAL*4, REAL*8, REAL*16
CHARACTER(*)

GEN_SORT also accepts unsigned "binary" integers of 1, 2, 4, or 8 byte sizes.

A radix algorithm is employed to sort the data. For all data types except REAL*16 and CHARACTER*(*), an N*16 byte work space is aquired from heap storage. For REAL*16 data, a work space of N*24 bytes is taken from heap storage. Heap work space required for CHARACTER data is N*((SIZE-1)/8+3)*8 bytes in size.

Example

```
REAL*4 DATA( 100 )
N = 100
CALL GEN_SORT( 'A','R',4,N,DATA,1,DATA,1 )
```

This Fortran code sorts a 100 element single-precision real vector in place.

GEN_SORTX
Sorts the Elements of an Indexed Vector

Format

GEN_SORTX (order, status, type, size, n, x, incx, index)

Arguments

order
character*1
On entry, **order** specifies the operation to be performed as follows:

If **order** = 'A' or 'a', **x** is sorted in ascending sequence.
If **order** = 'D' or 'd', **x** is sorted in descending sequence.

On exit, **order** is unchanged.

status
character*1
On entry, **status** specifies the operation to be performed as follows:

If **status** = 'N' or 'n', the **index** vector is new (empty) on input. Elements of data vector **x** are initially accessed in sequential order.
If **status** = 'O' or 'o', the **index** vector is old (full) on input. Elements of data vector **x** are initially accessed in the order specified by the **index** vector.

On exit, **status** is unchanged.

type
character*1
On entry, **type** specifies the data type of vectors **x** and **y**. The following types are valid:

If **type** = 'B' or 'b', binary (unsigned integer)
If **type** = 'C' or 'c', character string
If **type** = 'I' or 'i', integer (signed)
If **type** = 'L' or 'l', logical - Fortran .TRUE. or .FALSE.
If **type** = 'R' or 'r', IEEE floating point
If **type** = 'V' or 'v', VAXG floating point

On exit, **type** is unchanged.

size
integer*4
On entry, **size** specifies the size, in bytes, of each element of data vector **x**. Valid combinations of **type** and **size** include:

If **type** = 'B' or 'b' - **size** = { 1,2,4,8 }
If **type** = 'C' or 'c' - **size** = { 0 < **size** < 65536 }
If **type** = 'I' or 'i' - **size** = { 1,2,4,8 }
If **type** = 'L' or 'l' - **size** = { 1,2,4,8 }
If **type** = 'R' or 'r' - **size** = { 4,8,16 }
If **type** = 'V' or 'v' - **size** = { 4,8 }

On exit, **size** is unchanged.

n
integer*4
On entry, the length of the **x** vector.
On exit, **n** is unchanged.

x
data vector
On entry, a length **n** vector of data to be sorted. Each element of vector **x** is of **type** and **size** specified.
On exit, vector **x** is unchanged.

incx
integer*4
On entry, **incx** specifies the distance between elements of vector **x**. The argument **incx** must be positive.
On exit, **incx** is unchanged.

index
integer*4
On entry, vector **index** is ignored if **status** is 'N' or 'n', and used as an index vector for data vector **x** if **status** is 'O' or 'o'.
On exit, vector **index** is overwritten by a permuted vector of indices that may be used to access data vector **x** in the sorted order specified by **order**.

Description

GEN_SORTX is a general purpose, in memory, indexed sort routine. GEN_SORTX accepts the following Fortran data types:

INTEGER*1, INTEGER*2, INTEGER*4, INTEGER*8
LOGICAL*1, LOGICAL*2, LOGICAL*4, LOGICAL*8
REAL*4, REAL*8, REAL*16
CHARACTER(*)

GEN_SORTX also accepts unsigned "binary" integers of 1, 2, 4, or 8 byte sizes.

An indexed radix algorithm is employed to sort the data. For all data types except REAL*16 and CHARACTER*(*), an N*16 byte work space is aquired from heap storage. For REAL*16 data, a work space of N*24 bytes is taken from heap storage. Heap work space required for CHARACTER data is N*((SIZE-1)/8+3)*8 bytes in size.

GEN_SORTX is stable and may be used to perform multikey record sorts.

Example

```
REAL    DATA( 100 )
INTEGER INDEX( 100 )
N = 100
CALL GEN_SORTX( 'Descend','New','Real',SIZEOF(DATA(1)),N,DATA,1,INDEX )
DO I=1,N
   PRINT *,DATA(I),DATA( INDEX(I) )
ENDDO
```

This Fortran code sorts a 100 element single-precision real vector by index, leaving the data unchanged. Unsorted and sorted data are printed in adjacent columns.

A
Bibliography

This bibliography lists books and articles related to DXML computations. The list is divided into categories with appropriate headings.

The following books provide general information about mathematics in the areas of DXML computations:

Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. Philadelphia: SIAM, 1995.

Dongarra, J., I. Duff, D. Sorensen, and H. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. Philadelphia: SIAM, 1991.

Dongarra, J. J., C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK Users' Guide*. Philadelphia: SIAM, 1979.

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. Baltimore: The Johns Hopkins University Press, 1989.

A.1 Level 1 BLAS

Lawson, C. L., R. J. Hanson, D. R. Kincaid, and F. T. Krogh. "Basic linear algebra subprograms for Fortran usage." *ACM Transactions on Mathematical Software*, Vol. 5, No. 3, 1979.

A.2 Sparse Level 1 BLAS

Dodson, D. S., R. G. Grimes, and J. G. Lewis. "Sparse extensions to the Fortran basic linear algebra subprograms." *Boeing Computer Services Technical Report*, ETA-TR-63, August 1987.

Dodson, D. S., R. G. Grimes, and J. G. Lewis. "Model implementation and test package for the sparse basic linear algebra subprograms." *Boeing Computer Services Technical Report*, ETA-TR-63, August 1987.

Dodson, D. S., R. G. Grimes, and J. G. Lewis. "Sparse extensions to the FORTRAN basic linear algebra subprograms." *ACM Transactions on Mathematical Software*, Vol. 17, No. 2, pp. 253–263, June 1991.

Saad, Y., and H. Wijshoff. "SPARK: a benchmark package for sparse computations." *CSRD Technical Report*, University of Illinois at Urbana–Champaign, January 1990.

A.3 Level 2 BLAS

Dongarra, J., J. DuCroz, S. Hammarling, and R. Hanson. "A proposal for an extended set of FORTRAN basic linear algebra subprograms." *ACM SIGNUM Newsletter*, Vol. 20, No.1, 1985.

Dongarra, J., J. DuCroz, S. Hammarling, and R. Hanson. "An update notice on the extended BLAS." *ACM SIGNUM Newsletter*, Vol. 21, No. 4, 1986.

Dongarra, J., J. DuCroz, S. Hammarling, and R. Hanson. "An extended set of FORTRAN basic linear algebra subprograms." *ACM Transactions on Mathematical Software*, Vol. 14, No. 1, 1988.

Dongarra, J., J. DuCroz, S. Hammarling, and R. Hanson. "Algorithm 656. An extended set of basic linear algebra subprograms: model implementation and test programs." *ACM Transactions on Mathematical Software*, Vol. 14, No. 1, 1988.

A.4 Level 3 BLAS

Dongarra, J., J. DuCroz, I. Duff, and S. Hammarling. "A proposal for a set of level 3 basic linear algebra subprograms." *ACM SIGNUM Newsletter*, Vol. 22, No. 3, 1987.

Dongarra, J., J. DuCroz, I. Duff, and S. Hammarling. "A set of level 3 basic linear algebra subprograms." *ACM Transactions on Mathematical Software*, Vol. 16, No. 1, 1990.

A.5 Signal Processing

Antoniou, A. *Digital Filters: Analysis and Design*. New York: McGraw–Hill, 1979.

Blackman, R. B., and J. W. Tukey. *The Measurement of Power Spectra*. New York: Dover Publications, 1958.

Bose, N. K. *Digital Filters: Theory and Application*. New York: Elsevier Science Publishing Company, 1988.

Bracewell, R. N. *The Fourier Transform and Its Application*. New York: McGraw–Hill, 1986.

Brigham, E. O. *The Fast Fourier Transform*. Englewood Cliffs, N. J.: Prentice Hall, 1974.

Brigham, E. O. *The Fast Fourier Transform and Its Applications*. Englewood Cliffs, N. J.: Prentice Hall, 1988.

Burrus, C. S., and T. W. Parks. *DFT/FFT and Convolution Algorithms*. New York: Wiley–Interscience, 1985.

Elliot, D. F. *Handbook of Digital Signal Processing, Engineering Applications*. San Diego: Academic Press, 1987.

Elliot, D. F., and K. R. Rao. *Fast Transforms: Algorithms, Analyses, Applications*. Orlando: Academic Press, 1982.

Hamming, R. W. *Digital Filters*. New York: Prentice Hall, 1983.

Oppenheim, A. V., and R. W. Schafer. *Digital Signal Processing*. Englewood Cliffs, N. J.: Prentice Hall, 1975.

Rao, K. R., and P. Yip. *Discrete Cosine Transform: Algorithms, Advantages, Applications*. San Diego, California.: Academic Press, Inc., pp 15–20, 1990.

A.6 Iterative Solvers

Ashby, A., and M. Seager. "A proposed standard for iterative linear solvers, Version 1.0." *Lawrence Livermore National Laboratory Preprint*, UCRL-102860, January 1990.

Barrett, R., M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadelphia: SIAM, 1993.

Björck, A., and T. Elfving. "Accelerated projection methods for computing pseudo-inverse solutions of systems of linear equations." *BIT*, Vol. 31, pp. 145–163, 1979.

Dongarra, J., I. Duff, D. Sorensen, and H. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. Philadelphia: SIAM, 1991.

Fletcher, R. "Conjugate gradient methods for indefinite systems." In *Lecture Notes in Mathematics*, Vol. 506, Berlin: Springer–Verlag, 1976.

Freund, R., and Nachtigal, N., "QMR: a quasi-minimal residual method for non-Hermitian linear systems", *Numer. Math.*, 60, pp 315–339, 1991.

Freund, R. "A transpose-free quasi-minimal residual method for non-Hermitian linear systems", *SIAM Journal of Scientific Computing*, Vol. 14, pp. 470–482, March 1993.

Hestenes, M., and E. Stiefel. "Methods of conjugate gradients for solving linear systems." *J. Res. Natl. Bur. Stds.*, Vol. 49, pp. 409–436, 1952.

Kelley, C.T. *Iterative Methods for Linear and Nonlinear Equations*, SIAM, Philadelphia, 1995.

Meijerink, J. A., and H. A. van der Vorst. "An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix." *Math. Comp.*, Vol. 31, pp. 148–162, 1977.

Reid, J. K. "On the method of conjugate gradients for the solution of large sparse systems of linear equations." In *Large Sparse Sets of Linear Equations*, J. K. Reid, Ed., New York, pp. 231–254, 1971.

Saad, Y. *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, Boston, 1996.

Saad, Y., and M. Schultz. "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems." *SIAM J. Sci. Stat. Comput.*, Vol. 7, pp. 856–869, 1986.

Sonneveld, P. "CGS, A fast Lanczos-type solver for nonsymmetric linear systems." *SIAM J. Sci. Stat. Comput.*, Vol. 10, pp. 36–52, 1989.

A.7 Direct Solvers

Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. Philadelphia: SIAM, 1995.

Arioli, M., J. W. Demmel, and I. S. Duff. "Solving sparse linear systems with sparse backward error." *SIAM J. Matrix Anal. Appl.*, Vol. 10, pp. 165–190, 1989.

Demmel, J., J. Du Croz, S. Hammarling, and D. Sorensen. "Guidelines for the design of symmetric eigenroutines, SVD, and iterative refinement and condition estimation for linear systems." *LAPACK Working Note #4*, ANL, MCS-TM-111, 1988.

Duff, I. S., A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. New York: Oxford University Press, 1986.

Felippa, C. "Solution of linear equations with skyline stored symmetric matrix." *Computers and Structures*, Vol. 5, pp. 13–29, 1975.

George, A., and J. W-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Englewood Cliffs, N. J.: Prentice Hall, 1981.

Hager, W. W. "Condition estimators." *SIAM J. Sci. Stat. Comput.*, Vol. 5, pp. 311–316, 1984.

Higham, N. J. "FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation." *ACM Transactions Mathematical Software*, Vol. 14, pp. 381–396, 1988.

Jennings, A. "A compact storage scheme for the solution of symmetric linear simultaneous equations." *Computer Journal*, Vol. 9, pp. 281–285, 1966.

Pissanetzky, S. *Sparse Matrix Technology*. Miami: Academic Press, 1984.

Skeel, R. D. "Iterative refinement implies numerical stability for Gaussian elimination." *Math. Comp.*, Vol. 35, pp. 817–832, 1980.

A.8 Random Number Generators

Bratley, P., Fox, B., and Schrage, L.E., *A Guide to Simulation*, Second Edition, Springer-Verlag, New York, N.Y., 1987.

Knuth, D., "The Art of Computer Programming", Vol. 2, *Seminumerical Algorithms*, Second Edition, Addison-Wesley, Reading, Mass., 1981.

L'Ecuyer, P., "Efficient and Portable Combined Random Number Generators", *CACM* Vol. 31, Nr. 6, June 1988.

Fishman, G., Moore, L., "An Exhaustive Analysis of Multiplicative Congruential Random Number Generators with modulus 2**31-1", *Siam J. Sci. Stat. Comput.*, Vol. 7, No.1, January 1986.

Park, S., Miller, K., "Random Number Generators: Good Ones are Hard to Find", *CACM* Vol. 31, Nr. 10, October 1988.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

Z