Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard

Basic Linear Algebra Subprograms Technical (BLAST) Forum

August 21, 2001

Contents

Acknowledgments

The technical development was carried out by subgroups, whose work was reviewed by the full committee. During the period of development of the Basic Linear Algebra Subprograms (BLAS) Technical Forum Standard, many people served in positions of responsibility and are listed below.

- Jack Dongarra and Sven Hammarling, Conveners and Meeting Chairs
- Susan Blackford and Andrew Lumsdaine, Minutes
- Susan Blackford, Editor

The primary chapter authors are the following:

- Susan Blackford, Jack Dongarra, and Sven Hammarling, Chapter 1
- Susan Blackford, Jack Dongarra, and Sven Hammarling, Linda Kaufman, Zohair Maany, Antoine Petitet, Chapter 2
- Iain Duff, Mike Heroux, Roldan Pozo, Karin Remington, Chapter 3
- Jim Demmel, Greg Henry, Velvel Kahan, Xiaoye Li, Chapter 4
- Clint Whaley, C Interface to the Legacy BLAS
- Jack Dongarra, Fred Krogh, Journal of Development – Environmental routines
- Antoine Petitet, Journal of Development – Distributed-Memory Dense BLAS
- Sven Hammarling, Zohair Maany, Journal of Development – Fortran95 Thin BLAS
- George Corliss, Chenyi Hu, Baker Kearfoot, Bill Walster, J. Wolff v. Gudenberg, Journal of Development – Interval BLAS

We would like to thank the individuals from the following organizations who have written the reference implementations: University of California, Berkeley, University of Houston, Downtown, University of Notre Dame, University of Tennessee, HP/Convex, NAG, NIST, and CERFACS.

Specifically, we thank the following students at the University of California, Berkeley, for their work on the reference implementations and proofreading of various versions of the document: Ben Wanzo, Berkat Tung, Weihua Shen, Anil Kapur, Michael Martin, Jimmy Iskandar, Yozo Hida, Teresa Tung, Yulin Li.

We would like to thank the following vendors and ISPs: Cray, Digital/Compaq, HP/Convex, IBM, Intel, NEC, SGI, Tera, NAG, and VNI.

We thank Paul McMahan of the University of Tennessee for preparing the commenting and voting pages on the BLAS Technical Forum webpage.

We would like to thank the members of the global community who have posted comments, suggestions, and proposals to the email reflector and the BLAS Technical Forum webpage.

And lastly, we would like to thank the attendees of the BLAS Technical Forum meetings:

Andy Anda, Ed Anderson, Zhaojun Bai, David Bailey, Satish Balay, Puri Bangalore, Claus Bendtsen, Jesse Bennett, Mike Berry, Jeff Bilmes, Susan Blackford, Phil Bording, Clay Breshears, Sandra Carney, Mimi Celis, Andrew Chapman, Samar Choudhary, Edmond Chow, Almadena Chtchelkanova, Andrew Cleary, Isom Crawford, Michel Daydé, John Dempsey, Theresa Do, Dave Dodson, Jack Dongarra, Craig Douglas, Paul Dressel, Jeremy Du Croz, Iain Duff, Carter Edwards, Salvatore Filippone, Rob Gjertsen, Roger Golliver, Cormac Garvey, Ian Gladwell, Bruce Greer, Bill Gropp, John Gunnels, Fred Gustavson, Sven Hammarling, Richard Hanson, Hidehiko Hasegawa, Satomi Hasegawa, Greg Henry, Mike Heroux, Jeff Horner, Gary Howell, Mary Beth Hribar, Chenyi Hu, Steve Huss-Lederman, Melody Ivory, Naoki Iwata, Bo Kågström, Velvel Kahan, Chandrika Kamath, Linda Kaufman, David Kincaid, Jim Koehler, Vipin Kumar, Rich Lee, Steve Lee, Guangye Li, Jin Li, Sherry Li, Hsin-Ying Lin, John Liu, Andew Lumsdaine, Dave Mackay, Kristin Marshoffe, Kristi Maschhoff, Brian McCandless, Joan McComb, Noel Nachtigal, Jim Nagy, Esmond Ng, Tom Oppe, Antoine Petitet, Roldan Pozo, Avi Purkayastha, Padma Raghavan, Karin Remington, Yousef Saad, Majed Sidani, Jeremy Siek, Tony Skjellum, Barry Smith, Ken Stanley, Pete Stewart, Shane Story, Chuck Swanson, Françoise Tisseur, Anne Trefethen, Anna Tsao, Robert van de Geijn, Phuong Vu, Kevin Wadleigh, David Walker, Bob Ward, Jerzy Waśniewski, Clint Whaley, Yuan-Jye Jason Wu, Chao Yang, and Guodong Zhang.

Suggestions for Reading

This document is divided into chapters, appendices, a journal of development, and an index of routine names. It is large, and it is not necessary for a user to read it in its entirety. A user may choose to not read certain chapters or sections within this document, depending upon his/her areas of interest. **Chapters 2–4** contain a functionality discussion and language bindings for dense and band, sparse, and mixed and extended precision BLAS, respectively. Thus, these chapters may be read independently, referring to **Chapter 1** and the **Appendix** for notation and implementation details common to all chapters. Refer to section 1.3 for a more detailed description of the organization of this document.

Chapter 1

Introduction

1.1 Introduction

This document defines the BLAS Technical Forum standard, a specification of a set of kernel routines for linear algebra, historically called the Basic Linear Algebra Subprograms and commonly known as the BLAS. In addition to this publication, the complete standard can be found on the BLAS Technical Forum webpage (`http://www.netlib.org/blas/blast-forum/`).

Numerical linear algebra, particularly the solution of linear systems of equations, linear least squares problems, eigenvalue problems and singular value problems, is fundamental to most calculations in scientific computing, and is often the computationally intense part of such calculations. Designers of computer programs involving linear algebraic operations have frequently chosen to implement certain low level operations, such as the dot product or the matrix vector product, as separate subprograms. This may be observed both in many published codes and in codes written for specific applications at many computer installations.

This approach encourages structured programming and improves the self-documenting quality of the software by specifying basic building blocks and identifying these operations with unique mnemonic names. Since a significant amount of execution time in complicated linear algebraic programs may be spent in a few low level operations, reducing the execution time spent in these operations leads to an overall reduction in the execution time of the program. The programming of some of these low level operations involves algorithmic and implementation subtleties that need care, and can be easily overlooked. If there is general agreement on standard names and parameter lists for some of these basic operations, then portability and efficiency can also be achieved.

The first major concerted effort to achieve agreement on the specification of a set of linear algebra kernels resulted in the Level 1 Basic Linear Algebra Subprograms (BLAS)[1] [39] and associated test suite. The Level 1 BLAS are the specification and implementation in Fortran of subprograms for scalar and vector operations. This was the result of a collaborative project in 1973-77. Following the distribution of the initial version of the specifications to people active in the development of numerical linear algebra software, a series of open meetings were held at conferences and, as a result, extensive modifications were made in an effort to improve the design and make the subprograms more robust. The Level 1 BLAS were extensively and successfully exploited by LINPACK [23], a software package for the solution of dense and banded linear equations and linear least squares problems.

With the advent of vector machines, hierarchical memory machines and shared memory parallel machines, specifications for the Level 2 and 3 BLAS [26, 25], concerned with matrix-vector and

[1] Originally known just as the BLAS, but in the light of subsequent developments now known as the Level 1 BLAS

matrix-matrix operations respectively, were drawn up in 1984-86 and 1987-88. These specifications made it possible to construct new software to utilize the memory hierarchy of modern computers more effectively. In particular, the Level 3 BLAS allowed the construction of software based upon block-partitioned algorithms, typified by the linear algebra software package LAPACK [6]. LAPACK is state-of-the-art software for the solution of dense and banded linear equations, linear least squares, eigenvalue and singular value problems, makes extensive use of all levels of BLAS and particularly utilizes the Level 2 and 3 BLAS for portable performance. LAPACK is widely used in application software and is supported by a number of hardware and software vendors.

To a great extent, the user community embraced the BLAS, not only for performance reasons, but also because developing software around a core of common routines like the BLAS is good software engineering practice. Highly efficient machine-specific implementations of the BLAS are available for most modern high-performance computers. The BLAS have enabled software to achieve high performance with portable code.

The original BLAS concentrated on dense and banded operations, but many applications require the solution of problems involving sparse matrices, and there have also been efforts to specify computational kernels for sparse vector and matrix operations [22, 27].

In the spirit of the earlier BLAS meetings and the standardization efforts of the MPI and HPF forums, a technical forum was established to consider expanding the BLAS in the light of modern software, language, and hardware developments. The BLAS Technical Forum meetings began with a workshop in November 1995 at the University of Tennessee. Meetings were hosted by universities, government institutions, and software and hardware vendors. Detailed minutes were taken for each of the meetings, and these minutes are available on the BLAS Technical Forum webpage (`http://www.netlib.org/blas/blast-forum/`).

Various working groups within the Technical Forum were established to consider issues such as the overall functionality, language interfaces, sparse BLAS, distributed-memory dense BLAS, extended and mixed precision BLAS, interval BLAS, and extensions to the existing BLAS. The rules of the forum were adopted from those used for the MPI and HPF forums. In other words, final acceptance of each of the chapters in the BLAS Technical Forum standard were decided at the meetings using *Robert's Rules*. Drafts of the document were also available on the BLAS Technical Forum webpage, and attendees were permitted to edit chapters, give comments, and vote on-line in "virtual meetings", as well as to conduct discussions on the email reflector. The efforts of these working groups are summarized in this document. Most of these discussions resulted in definitive proposals which led to the specifications given in Chapters 2 - 4. Not all of the discussions resulted in definitive proposals, and such discussions are summarized in the Journal of Development in the hope that they may encourage future efforts to take those discussions to a successful conclusion.

A major aim of the standards defined in this document is to enable linear algebra libraries (both public domain and commercial) to interoperate efficiently, reliably and easily. We believe that hardware and software vendors, higher level library writers and application programmers all benefit from the efforts of this forum and are the intended end users of these standards.

The specification of the original BLAS was given in the form of Fortran 66 and subsequently Fortran 77 subprograms. In this document we provide specifications for Fortran 95[2], Fortran 77 and C. Reference implementations of the standard are provided on the BLAS Technical Forum webpage (`http://www.netlib.org/blas/blast-forum/`). Alternative language bindings for C++ and Java were also discussed during the meetings of the forum, but the specifications for these bindings were postponed for a future series of meetings.

The remainder of this chapter is organized as follows. Section 1.2 provides motivation for the

[2] the current Fortran standard

functionality. Section 1.3 outlines the organization of the document, and section 1.4 summarizes the nomenclature and conventions used in the document. Section 1.5 presents tables of functionality for the routines, and section 1.6 discusses issues concerning the numerical accuracy of the BLAS. Section 1.7 briefly describes the presentation of the specifications for the routines, and section 1.8 details the error handling mechanisms utilized within the routines.

1.2 Motivation

The motivation for the kernel operations is proven functionality. Many of the new operations are based upon auxiliary routines in LAPACK [6] (e.g., SUMSQ, GEN_GROT, GEN_HOUSE, SORT, GE_NORM, GE_COPY). Only after the LAPACK project was begun was it realized that there were operations like the matrix copy routine (GE_COPY), the computation of a norm of a matrix (GE_NORM) and the generation of Householder transformations (GEN_HOUSE) that occurred so often that it was wise to make separate routines for them.

A second group of these operations extended the functionality of some of the existing BLAS (e.g., AXPBY, WAXPBY, GER, SYR/HER, SPR/HPR, SYR2/HER2, SPR2/HPR2). For example, the Level 3 BLAS for the rank k update of a symmetric matrix only allows a positive update, which means that it cannot be used for the reduction of a symmetric matrix to tridiagonal form (to facilitate the computation of the eigensystem of a symmetric matrix), or for the factorization of a symmetric indefinite matrix, or for a quasi-Newton update in an optimization routine.

Other extensions (e.g., AXPY_DOT, GE_SUM_MV, GEMVT, TRMVT, GEMVER) perform two Level 1 BLAS (or Level 2 BLAS) routine calls simultaneously to increase performance by reducing memory traffic.

One important feature of the new standard is the inclusion of sparse matrix computational routines. Because there are many formats commonly used to represent sparse matrices, the Level 2 and Level 3 Sparse BLAS routines utilize an abstract representation, or handle, rather than a fixed storage description (e.g. compressed row, or skyline storage). This handle-based representation allows one to write portable numerical algorithms using the Sparse BLAS, independent of the matrix storage implementation, and gives BLAS library developers the best opportunity for optimizing and fine-tuning their kernels for specific architectures or application domains.

The original Level 2 BLAS included, as an appendix, the specification of extended precision subprograms. With the widespread adoption of hardware supporting the IEEE extended arithmetic format [37], as well as other forms of extended precision arithmetic, together with the increased understanding of algorithms to successfully exploit such arithmetic, it was felt to be timely to include a complete specification for a set of extra precise BLAS.

1.3 Organization of the Document

This document is divided into chapters, appendices, a journal of development, and an index. It is large, and it is not necessary for a user to read it in its entirety. A user may choose to not read certain chapters or sections within this document, depending upon his/her areas of interest. **Chapters 2–4** contain a functionality discussion and language bindings for dense and band, sparse, and mixed and extended precision BLAS, respectively. The **Journal of Development** presents areas of research that are not yet mature enough to be considered as chapters, but were nevertheless discussed at the meetings of the forum. A **Bibliography** is also provided, as well as an **Index** of routine names.

All users are encouraged to frequently refer to the list of notation denoted in sections 1.4, 2.3, and 3.4.

- **Chapter 1: Introduction** provides a brief overview of the background, motivation, and history of the BLAS Technical Forum effort. It also outlines the structure of the document, conventions in notation, and overall functionality contained in the chapters.
- **Chapter 2: Dense and Banded BLAS** presents the functionality and language bindings for proposed "new" dense and banded BLAS routines for serial and shared memory computing.
- **Chapter 3: Sparse BLAS** presents the functionality and language bindings for proposed "new" sparse BLAS routines for serial and shared memory computing.
- **Chapter 4: Extended and Mixed Precision BLAS** presents the functionality and language bindings for proposed extended- precision and mixed-precision BLAS routines for serial and shared memory computing.
- **Appendix** contains pertinent definitions and implementation details for the chapters.
- **Legacy BLAS** contains alternative language bindings for the legacy Level 1, 2, and 3 BLAS for dense and band matrix computations.
- **Journal of Development** contains separate proposals for environmental enquiry routines, Distributed-memory dense BLAS, Fortran 95 Thin BLAS, and Interval BLAS.

1.4 Nomenclature and Conventions

This section addresses mathematical notation and definitions, as well as the numerical accuracy for the BLAS routines. Language-independent issues are also presented.

1.4.1 Notation

The following notation is used throughout the document.

- A, B, C – matrices
- D, D_L, D_R – diagonal matrices
- H – Householder matrix
- J – symmetric tridiagonal matrix (including 2×2 blocked diagonal)
- P – permutation matrix
- T – triangular matrix
- $op(A)$ – denotes A, or A^T or A^H where A is a matrix.
- transpose – denotes A^T where A is a matrix.
- conjugate-transpose – denotes A^H where A is a complex Hermitian matrix.

- u, v, w, x, y, z – vectors
- $\bar{x}$ – specifies the conjugate of the complex vector x
- $incu$, $incv$, $incw$, $incx$, $incy$, $incz$ – stride between successive elements of the respective vector
- Greek letters - scalars (but not exclusively Greek letters)
- x_i - an element of a one-dimensional array
- $y|_x$ – refers to the elements of y that have common indices with the sparse vector x.
- ϵ - machine epsilon
- $\leftarrow$ – assignment statement
- $\leftrightarrow$ – swap (assignment) statement
- $||\cdot||_p$ – the p-norm of a vector or matrix

Additional notation for sparse matrices can be found in 3.4.

For the mathematical formulation of the operations, as well as their algorithmic presentation, we have chosen to index the vector and matrix operands starting from zero. This decision was taken to simplify the presentation of the document but has no impact on the convention a particular language binding may choose.

1.4.2 Operator Arguments

Some BLAS routines take input-only arguments that are called "operator" arguments. These arguments allow for the specification of multiple related operations to be performed by a single function.

The operator arguments used in this document are norm, sort, side, uplo, trans, conj, diag, jrot, order, index_base, and prec. Their possible meanings are defined as follows:

norm: this argument is used by the routines computing the norm of a vector or matrix. Eight possible distinct values are valid that specify the norm to be computed, namely the one-norm, real one-norm, infinity-norm and real infinity norms for vectors and matrices, the 2-norm for vectors, and the Frobenius-norm, max-norm and real max-norm for matrices.

sort: this argument is used by the sorting routines. Two possible distinct values are valid that specify whether the data should be sorted in increasing or decreasing order.

side: this argument is used only by functions computing the product of two matrices A and B. Two possible distinct values are valid, that specify whether $A \cdot B$ or $B \cdot A$ should be computed.

uplo: this argument refers to triangular and symmetric (Hermitian) matrices. Two possible distinct values are valid distinguishing whether the matrix, or its storage representation, is upper or lower triangular.

trans: this argument is used by the routines applying a matrix, say A, to another vector or another matrix. Three possible distinct values are valid that specify whether the matrix A, its transpose A^T or its conjugate transpose A^H should be applied. We use the notation $op(A)$ to refer to A, A^T or A^H depending on the input value of the trans operator argument.

- conj: this argument is used by the complex routines operating with $\bar{x}$ or x.
- diag: this argument refers exclusively to triangular matrices. Two possible distinct values are valid distinguishing whether the triangular matrix has unit-diagonal or not.
- jrot: this argument is used by the routine to generate Jacobi rotations. Three possible distinct values are valid and specify whether the rotation is an inner rotation, an outer rotation, or a sorted rotation.
- order: this argument is used by the C bindings to specify if elements within a row of an array are contiguous, or if elements within a column of an array are contiguous (see section 2.6.6).
- index_base: this argument is used by Chapter 3 to specify either one-based or zero-based indexing (see section 3.4.1).
- prec: this argument is used in Chapter 4 and specifies the internal precision to be used by an extended precision routine. Four distinct values are valid and specify whether the internal precision is single precision, double precision, indigenous, or extra. Details on these settings can be found in section 4.3.1.

All possible meanings for each operator are listed in section A.3. Their representation is defined in the interface issues for the specific programming language: sections 2.4, 3.6.1, and 4.4.1 for Fortran 95; sections 2.5, 3.6.2, and 4.4.2 for Fortran 77; and sections 2.6, 3.6.3, and 4.4.3 for C. The values of the Fortran 95 derived types (for Chapters 2 and 4) are defined in the Fortran 95 module `blas_operator_arguments`, and the values of the Fortran 95 named constants (for Chapter 3) are defined in `blas_sparse_namedconstants`, see section A.4. Similarly, the values of the Fortran 77 named constants are defined in the Fortran 77 include file `blas_namedconstants.h`, in section A.5. And finally, the values of the C enumerated types are defined in the C include file `blas_enum.h`, in section A.6.

Rationale. The intent is to provide each language binding with the opportunity to choose the most appropriate form these arguments should take. For example, in Fortran 95, derived types with named constants have been selected for Chapters 2 and 4, whereas derived types could not be used in Chapter 3 (see section 3.6.1 for details). In Fortran 77, integers with named constants have been chosen. And finally, in C, operator arguments are represented by enumerated types. (*End of rationale.*)

1.4.3 Scalar Arguments

Many scalar arguments are used in the specifications of the BLAS routines. For example, the size of a vector or matrix operand is determined by the integer argument(s) m and/or n. Note that it is permissible to call the routines with m or n equal to zero, in which case the routine exits immediately without referencing its vector/matrix elements. Some routines return a displacement denoted by the integer argument k. The scaling of a vector or matrix is often denoted by the arguments alpha and beta.

The following symbols are used: a, b, c, d, r, s, t, alpha, beta and tau.

1.4.4 Vector Operands

A n-length vector operand x is specified by two arguments – x and incx. x is an array that contains the entries of the n-length vector x. incx is the stride within x between two successive elements of the vector x.

The following lowercase letters are used to denote a vector: u, v, w, x, y, and z. The corresponding strides are respectively denoted incu, incv, incw, incx, incy, and incz.

> *Advice to implementors.* The increment arguments incu, incv, incw, incx, incy and incz may not be zero. (*End of advice to implementors.*)

Example: The mathematical function returning the inner-product r of two real n-length vectors x and y can be defined by:

$$r = x^T y = \sum_{i=0}^{n-1} x_i y_i.$$

> *Rationale.* The arguments incx, and incy do not play a role in the mathematical formulation of the operation. These arguments allow for the specification of subvector operands in various language bindings. Therefore, some of these arguments may not be present in all language-dependent specifications. (*End of rationale.*)

1.4.5 Matrix Operands

A m-by-n matrix operand A is specified by the argument A. A is a language-dependent data structure containing the entries of the matrix operand A. The representation of the matrix entry $a_{i,j}$ in A is denoted by A(i,j) for all (i,j) in the interval $[0 \ldots m-1] \times [0 \ldots n-1]$.

Capital letters are used to denote a matrix. The functions involving matrices use only four symbols, namely A, B, C, and T.

1.4.6 Naming Conventions

Language bindings are specified for Fortran 95, Fortran 77, and C.

The Fortran 95 language bindings have routine names of the form <**name**>, where <**name**> is in lowercase letters and indicates the computation performed. These bindings use generic interfaces to manipulate the data type of the routine, and thus their names do not contain a letter to denote the data type.

The Fortran 77 and C language bindings have routine names of the form **BLAS_x**<**name**>, where the letter **x**, indicates the data type as follows:

Data type	x	Fortran 77	x	C
s.p. real	S	REAL	s	float
d.p. real	D	DOUBLE PRECISION	d	double
s.p. complex	C	COMPLEX	c	float
d.p.complex	Z	COMPLEX*16 or DOUBLE COMPLEX	z	double

The suffix <**name**> in the routine name indicates the computation performed. In the matrix-vector and matrix-matrix routines of Chapters 2 and 4 (and Appendix C.4), the type of the matrix (or of the most significant matrix) is also specified as part of this <**name**> name of the routine. Most of these matrix types apply to both real and complex matrices; a few apply specifically to one or the other, as indicated below. Note that for Appendix C.4, these matrix types apply to interval matrices.

GB	general band
GE	general (i.e., unsymmetric, in some cases rectangular)
HB	(complex) Hermitian band
HE	(complex) Hermitian
HP	(complex) Hermitian, packed storage
SB	(real) symmetric band
SP	symmetric, packed storage
SY	symmetric
TB	triangular band
TP	triangular, packed storage
TR	triangular (or in some cases quasi-triangular)
US	unstructured sparse

For Fortran 77, routine names are in uppercase letters; however, for the C interfaces all routine names are in lowercase letters. To avoid possible name collisions, programmers are strongly advised not to declare variables or functions with names beginning with these prefixes.

A detailed discussion of the format of the <**name**> naming convention is contained in each respective chapter of the document.

1.5 Overall Functionality

This section summarizes, in tabular form, the functionality of the proposed routines. Issues such as storage formats or data types are not addressed. The functionality of the existing Level 1, 2 and 3 BLAS [39, 22, 26, 25] is a subset of the functionality proposed in this document.

In the original BLAS, each level was categorized by the type of operation; Level 1 addressed scalar and vector operations, Level 2 addressed matrix-vector operations, while Level 3 addressed matrix-matrix operations. The functionality tables in this document are categorized in a similar manner, with additional categories to cover operations which were not addressed in the original BLAS.

Unless otherwise specified, the operations apply to both real and complex arguments. For the sake of compactness the complex operators are omitted, so that whenever a transpose operation is given the conjugate transpose should also be assumed for the complex case.

The last column of each table denotes in which chapter of this document the functionality occurs. Specifically,

- “D” denotes dense and banded BLAS (Chapter 2),
- “S” denotes sparse BLAS (Chapter 3), and
- “E” denotes extended and mixed precision BLAS (Chapter 4).

1.5.1 Scalar and Vector Operations

This section lists scalar and vector operations. The functionality tables are organized as follows. Table 1.1 lists the scalar and vector reduction operations, Table 1.2 lists the vector rotation operations, Table 1.3 lists the vector operations, and Table 1.4 lists those vector operations that involve only data movement.

For the Sparse BLAS, x is a compressed sparse vector and y is a dense vector. Details of data structures are in Section 3.4.1.

For further details of vector norm notation, refer to section 2.1.1.

Dot product	$r \leftarrow \beta r + \alpha x^T y$	D,E
	$r \leftarrow x^T y$	S
Vector norms	$r \leftarrow \|\|x\|\|_1,$	D
	$r \leftarrow \|\|x\|\|_{1R},$	D
	$r \leftarrow \|\|x\|\|_2,$	D
	$r \leftarrow \|\|x\|\|_\infty,$	D
	$r \leftarrow \|\|x\|\|_{\infty R},$	D
Sum	$r \leftarrow \sum_i x_i$	D,E
Min value & location	$k, x_k, ; k = \arg\min_i x_i$	D
Min abs value & location	$k, x_k, k = \arg\min_i(\|Re(x_i)\| + \|Im(x_i)\|)$	D
Max value & location	$k, x_k, ; k = \arg\max_i x_i$	D
Max abs value & location	$k, x_k, k = \arg\max_i(\|Re(x_i)\| + \|Im(x_i)\|)$	D
Sum of squares	$(scl, ssq) \leftarrow \sum x_i^2,$	D
	$ssq \cdot scl^2 = \sum x_i^2$	D

Table 1.1: Reduction Operations

Generate Givens rotation	$(c, s, r) \leftarrow \mathrm{rot}(a, b)$	D
Generate Jacobi rotation	$(a, b, c, s) \leftarrow \mathrm{jrot}(x, y, z)$	D
Generate Householder transform	$(\alpha, x, \tau) \leftarrow \mathrm{house}(\alpha, x),$	D
	$H = I - \alpha u u^T$	

Table 1.2: Generate Transformations

Reciprocal Scale	$x \leftarrow x/\alpha$	D
Scaled vector accumulation	$y \leftarrow \alpha x + \beta y,$	D,E
	$y \leftarrow \alpha x + y$	S
Scaled vector addition	$w \leftarrow \alpha x + \beta y$	D,E
Combined axpy & dot product	$\begin{cases} \hat{w} \leftarrow w - \alpha v \\ r \leftarrow \hat{w}^T u \end{cases}$	D
Apply plane rotation	$(\; x \quad y \;) \leftarrow (\; x \quad y \;) R$	D

Table 1.3: Vector Operations

Copy	$y \leftarrow x$	D
Swap	$y \leftrightarrow x$	D
Sort vector	$x \leftarrow \mathrm{sort}(x)$	D
Sort vector & return index vector	$(p, x) \leftarrow \mathrm{sort}(x)$	D
Permute vector	$x \leftarrow Px$	D
Sparse gather	$x \leftarrow y\|_x$	S
Sparse gather and zero	$x \leftarrow y\|_x; \; y\|_x \leftarrow 0$	S
Sparse scatter	$y\|_x \leftarrow x$	S

Table 1.4: Data Movement with Vectors

1.5.2 Matrix-Vector Operations

This section lists matrix-vector operations in table 1.5. The matrix arguments A, B and T are dense or banded or sparse. In addition, where appropriate, the matrix A can be symmetric (Hermitian) or triangular or general. The matrix T represents an upper or lower triangular matrix, which can be unit or non-unit triangular. For the Sparse BLAS, the matrix A is sparse, the matrix T is sparse triangular, and the vectors x and y are dense.

Details of the data structures are discussed in sections 2.2, and 3.4.1.

Matrix-vector product	$y \leftarrow \alpha A x + \beta y,\ y \leftarrow \alpha A^T x + \beta y$	D,S,E
	$x \leftarrow \alpha T x,\ x \leftarrow \alpha T^T x$	D,E
	$y \leftarrow \alpha A x + y,\ y \leftarrow \alpha A^T x + y$	S
Summed matrix-vector multiplies	$y \leftarrow \alpha A x + \beta B x$	D,E
Multiple matrix-vector multiplies	$\begin{cases} x \leftarrow T^T y \\ w \leftarrow T z \end{cases}$	D
	$\begin{cases} x \leftarrow \beta A^T y + z \\ w \leftarrow \alpha A x \end{cases}$	D
Multiple matrix-vector mults and low rank updates	$\begin{cases} \hat{A} \leftarrow A + u_1 v_1^T + u_2 v_2^T \\ x \leftarrow \beta \hat{A}^T y + z \\ w \leftarrow \alpha \hat{A} x \end{cases}$	D
Triangular solve	$x \leftarrow \alpha T^{-1} x,\ x \leftarrow \alpha T^{-T} x$	D,S,E
Rank one updates	$A \leftarrow \alpha x y^T + \beta A$	D
and symmetric ($A = A^T$)	$A \leftarrow \alpha x x^T + \beta A$	D
rank one & two updates	$A \leftarrow (\alpha x) y^T + y (\alpha x)^T + \beta A$	D

Table 1.5: Matrix-Vector Operations

1.5.3 Matrix Operations

This section lists a variety of matrix operations. The functionality tables are organized as follows. Table 1.6 lists single matrix operations and matrix operations that involve $O(n^2)$ operations, Table 1.7 lists the $O(n^3)$ matrix-matrix operations and Table 1.8 lists those matrix operations that involve only data movement. Where appropriate one or more of the matrices can also be symmetric (Hermitian) or triangular or general. The matrix T represents an upper or lower triangular matrix, which can be unit or non-unit triangular. D, D_L, and D_R represent diagonal matrices, and J represents a symmetric tridiagonal matrix (including 2×2 block diagonal).

Details of the data structures are discussed in sections 2.2, and 3.4.1.

For further details of matrix norm notation, refer to section 2.1.3.

Matrix norms	$r \leftarrow \|\|A\|\|_1, r \leftarrow \|\|A\|\|_{1R}$	D
	$r \leftarrow \|\|A\|\|_F, r \leftarrow \|\|A\|\|_\infty, r \leftarrow \|\|A\|\|_{\infty R}$	D
	$r \leftarrow \|\|A\|\|_{max}, r \leftarrow \|\|A\|\|_{maxR}$	D
Diagonal scaling	$A \leftarrow DA,\ A \leftarrow AD,\ A \leftarrow D_L A D_R$	D
	$A \leftarrow DAD$	D
	$A \leftarrow A + BD$	D
Matrix acc and scale	$C \leftarrow \alpha A + \beta B$	D
Matrix add and scale	$B \leftarrow \alpha A + \beta B,\ B \leftarrow \alpha A^T + \beta B$	D

Table 1.6: Matrix Operations – $O(n^2)$ floating point operations

Matrix-matrix product	$C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha A^T B + \beta C$	D,E
	$C \leftarrow \alpha AB^T + \beta C,\ C \leftarrow \alpha A^T B^T + \beta C$	D,E
	$C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha A^T B + \beta C$	S
Triangular multiply	$B \leftarrow \alpha TB,\ B \leftarrow \alpha BT$	D,E
	$B \leftarrow \alpha T^T B,\ B \leftarrow \alpha BT^T$	D,E
Triangular solve	$B \leftarrow \alpha T^{-1} B,\ B \leftarrow \alpha T^{-T} B$	D,S,E
	$B \leftarrow \alpha BT^{-1},\ B \leftarrow \alpha BT^{-T}$	D,E
Symmetric rank k & $2k$ updates ($C = C^T$)	$C \leftarrow \alpha AA^T + \beta C,\ C \leftarrow \alpha A^T A + \beta C$	D,E
	$C \leftarrow \alpha AJA^T + \beta C,\ C \leftarrow \alpha A^T JA + \beta C$	D
	$C \leftarrow (\alpha A)B^T + B(\alpha A)^T + \beta C,$ $C \leftarrow (\alpha A)^T B + B^T(\alpha A) + \beta C$	D,E
	$C \leftarrow (\alpha AJ)B^T + B(\alpha AJ)^T + \beta C,$ $C \leftarrow (\alpha AJ)^T B + B^T(\alpha AJ) + \beta C$	D

Table 1.7: Matrix-Matrix Operations - $O(n^3)$ floating point operations

Matrix copy	$B \leftarrow A,\ B \leftarrow A^T$	D
Matrix transpose	$A \leftarrow A^T$	D
Permute Matrix	$A \leftarrow PA,\ A \leftarrow AP$	D

Table 1.8: Data Movement with Matrices

1.6 Numerical Accuracy and Environmental Enquiry

To understand the numerical behavior of the routines proposed here, certain floating point parameters are necessary. Detailed error bounds and limitations due to overflow and underflow are discussed in individual chapters (see sections 2.7, 3.7, 4.3.3, and C.4.4) but all of them depend on details of how floating point numbers are represented. These details are available by calling an environmental enquiry function called FPINFO.

Floating point numbers are represented in scientific notation as follows. This discussion follows the IEEE Floating Point Arithmetic Standard 754 [7].[3]

$$x = \pm d.d \cdots d * BASE^{E}$$

where $d.d \cdots d$ is a number represented as a string of T significant digits in base BASE with the "point" to the right of the leftmost digit, and E is an integer exponent. E ranges from EMIN up to EMAX. This means that the largest representable number, which is also called the *overflow threshold* or OV, is just less than $BASE^{EMAX+1}$, This also means that the smallest positive "normalized" representable number (i.e. where the leading digit of $d.d \cdots d$ is nonzero) is $BASE^{EMIN}$, which is also called the *underflow threshold* or UN.

When overflow occurs (because a computed quantity exceeds OV in absolute value), the result is typically $\pm\infty$, or perhaps an error message. When underflow occurs (because a computed quantity is less than UN in absolute magnitude) the returned result may be either 0 or a tiny number less than UN in magnitude, with minimal exponent EMIN but with a leading zero ($0.d \cdots d$). Such tiny numbers are often called *denormalized* or *subnormal*, and floating point arithmetic which returns them instead of 0 is said to support *gradual underflow*.

The *relative machine precision* (or *machine epsilon*) of a basic operation $\odot \in \{+, -, *, /\}$ is defined as the smallest $EPS > 0$ satisfying

$$fl(a \odot b) = (a \odot b) * (1 + \delta) \text{ for some } |\delta| \leq EPS$$

for all arguments a and b that do not cause underflow, overflow, division by zero, or an invalid operation. When $fl(a \odot b)$ is a closest floating point number to the true result $a \odot b$ (with ties broken arbitrarily), then rounding is called "proper" and $EPS = .5 * BASE^{1-T}$. Otherwise typically $EPS = BASE^{1-T}$, although it can sometimes be worse if arithmetic is not implemented carefully. We further say that rounding is "IEEE style" if ties are broken by rounding to the nearest number whose least significant digit is even (i.e. whose bottom bit is 0).

The function FPINFO returns the above floating point parameters, among others, to help the user understand the accuracy to which results are computed. FPINFO can return the values for either single precision or double precision. The way the precision is specified is language dependent, as is the choice of floating point parameter to return, and described in section 2.7. The names single and double may have different meanings on different machines: We have long been accustomed to single precision meaning 32-bits on all IEEE and most other machines [7], except for Cray and its emulators where single is 64-bits. And there are historical examples of 60-bit formats on some old CDC machines, etc. Nonetheless, we all agree on single precision as a phrase with a certain system-dependent meaning, and double precision too, meaning at least twice as many significant digits as single.

[3] We ignore implementation details like "hidden bits", as well as unusual representations like logarithmic arithmetic and double-double.

The values returned by FPINFO are as follows, including the values returned for IEEE single and IEEE double, the most common cases. The floating point parameters in column 1 have analogous meanings as the like-named character arguments of the LAPACK subroutine xLAMCH.[4]

Floating point parameter	Description	Value in IEEE single	Value in IEEE double
BASE	base of the machine	2	2
T	number of digits	24	53
RND	1 when proper rounding occurs in addition 0 otherwise	1	1
IEEE	1 when rounding in addition is IEEE style 0 otherwise	1	1
EMIN	minimum exponent before (gradual) underflow	-126	-1022
EMAX	maximum exponent before overflow	127	1023
EPS	machine epsilon = .5*BASE^{1-T} if RND=1 = BASE^{1-T} if RND=0	$2^{-24} \approx 5 \times 10^{-8}$	$2^{-53} \approx 10^{-16}$
PREC	EPS*BASE	2^{-23}	2^{-52}
UN	underflow threshold = BASE^{EMIN}	$2^{-126} \approx 10^{-38}$	$2^{-1022} \approx 10^{-308}$
OV	overflow threshold = BASE^{EMAX+1} $*$ (1$-$EPS)	$\sim 2^{128} \approx 10^{38}$	$\sim 2^{1024} \approx 10^{308}$
SFMIN	safe minimum, such that 1/SFMIN does not overflow = UN if 1/OV<UN, else (1+EPS)/OV	$2^{-126} \approx 10^{-38}$	$2^{-1022} \approx 10^{-308}$

Table 1.9: Values returned by FPINFO

Chapter 4 defines an additional FPINFO-like function to supplement this one with additional information needed for error bounds.

1.7 Language Bindings

Each specification of a routine corresponds to an operation outlined in the functionality tables. Operations are organized analogous to the order in which they are presented in the functionality tables. The specification has the form:

NAME (*multi-word description of operation*) $<$ *mathematical representation* $>$

[4]Here are the differences: In xLAMCH, UN was called RMIN and OV was called RMAX. The value of IEEE was computed by xLAMCH but not returned. xLAMCH returned EMIN+1 and EMAX+1 instead of EMIN and EMAX, respectively (this corresponds to a different choice of where to put the "point" in $d.d \cdots d * BASE^E$).

Optional brief textual description of the functionality including any restrictions that apply to all language bindings.

- Fortran 95 binding
- Fortran 77 binding
- C binding

Alternative language bindings for C++ and Java were also discussed during the meetings of the forum, but the specifications for these bindings were postponed for a future series of meetings.

1.8 Error Handling

This document supports two types of error-handling capabilities: an error handler and error return codes. Each chapter of this document, and thus each flavor of BLAS, has the choice of using either capability, whichever is more appropriate. Chapters 2 and 4 rely on an error handler, and Chapter 3 provides error return codes.

One error handler, BLAS_ERROR, is defined. A series of error return codes are also defined. Each function in this document determines when and if an error-handling mechanism is called, and its function specification must document the conditions (if any) which trigger the error handling mechanism.

1.8.1 Return Codes

Routines in the Sparse BLAS chapter utilize return codes since many of the operations need to be recoverable. In Fortran 95 and 77, the error return code of a BLAS routine is returned in the parameter `istat`, usually the last argument in the parameter list. In C, the error code is the return value of the function. In either case, the value of the error code is the integer 0 if the operation was successful. In the event of an error detection, a nonzero value is returned and control returns back to the calling program, as usual. The application is not aborted or halted, and it is the responsibility of the caller to check error status of these BLAS operations.

1.8.2 Error Handlers

The error handler defines some minimal scalar input argument checking.

> *Advice to implementors.* A BLAS supplier is free to provide multiple interfaces to the libraries, so that a second interface may perform no error checking. (*End of advice to implementors.*)

Additional error checking may be performed (for instance, checking that there are no zeros on the diagonal of a triangular solve), but these kinds of tests are too implementation-constraining to be mandated by the standard. Any additional error checking must not abort execution.

When any of the mandated scalar input argument checks fail, if the BLAS error handler is used, it must use the API given below. The default behavior of the BLAS-compliant error handler is to print an informative error message and abort execution. However, the API of this error handler is mandated by this document specifically so that a user can override the default error handler with a user-defined routine, so that this behavior can be changed. It is therefore necessary that the implementor not assume that the error handler stops execution, but rather must return explicitly before altering the routine's operands in the event of an error.

The following are defined as errors by this standard. All Fortran 95, Fortran 77, and C routines must perform the following error check.

- Any value of the operator arguments whose meaning is not specified in section A.3 is invalid.

Additionally, all Fortran 77 and C routines must perform the following error checks, unless otherwise noted in the specification of the routine.

- Any problem dimension or bandwidth (eg., m, n, k, kl, ku) less than zero
- Any vector increment (eg., incw, incx, incy, incz) equal to zero
- Any leading dimension (eg. lda, ldb, ldc, ldt) less than one
- Any leading dimension (eg. lda, ldb, ldc, ldt) less than the relevant dimension of the problem. The relevant dimension of the problem is:
  - n, for a square, symmetric, or triangular matrix
  - m, for a m $\times$ n general, non-transposed matrix
  - n, for a m $\times$ n general, transposed matrix
  - kl + ku + 1 for a m $\times$ n general band matrix
  - k + 1 for a n $\times$ n symmetric or triangular band matrix with k super- or subdiagonals

Each language binding possesses its own unique error handler. However, all error handlers minimally pass three pieces of information:

1. `RNAME`, the name of the routine in which the error occurred.
2. `IFLAG`, an integer flag which, if negative, means that parameter number `-IFLAG` caused the error, and if set to nonnegative, is an implementation-specific error code
3. `IVAL`, the value of parameter number `-IFLAG`.

Each language's BLAS error handler should print an informative error message describing the error, and halt execution. The API of the error handler is explicitly spelled out in each section, so that if this behavior is not desired by the user or higher level library provider, it may be changed by the BLAS user, overriding the BLAS's error handler with one which performs as required.

The API for each language binding is mandated in the following sections; as an advice to the implementor, an example of a BLAS-2000 compliant error handler is included as well.

F95 error handler

The Fortran 95 BLAS do not need to test the option arguments, since these are derived types and hence invalid arguments are flagged by the compiler. The only case where array dimensions are arguments to the Fortran 95 BLAS are the nonsymmetric band routines where m and kl are passed as arguments. The other array dimensions can be determined in the BLAS routines using the intrinsic function SIZE, and arrays should be checked for conformance according to the operation being performed. For example in the operation AB the second dimension of A must equal the first dimension of B. Note that, for consistency, m is included in all of the nonsymmetric band routines although in some cases it is redundant; in those cases it should be tested against the relevant array dimension.

The mandated API of the routine is:

```
MODULE blas_error_handler
  INTERFACE blas_error
    SUBROUTINE blas_error(rname,iflag,ival)
      INTEGER, INTENT (IN) :: iflag
      INTEGER, OPTIONAL, INTENT (IN) :: ival
      CHARACTER (*), INTENT (IN) :: rname
    END SUBROUTINE blas_error
  END INTERFACE
END MODULE blas_error_handler
```

A possible implementation would be:

```
    SUBROUTINE blas_error(rname,iflag,ival)
      ! .. Scalar Arguments ..
      ! The optional argument ival must be present when iflag is in (-98,-1)
      INTEGER, INTENT (IN) :: iflag
      INTEGER, OPTIONAL, INTENT (IN) :: ival
      CHARACTER (*), INTENT (IN) :: rname
      ! ..
      SELECT CASE (iflag)
      CASE (-99)
        WRITE (*,1000) rname
      CASE (-98:-1)
        WRITE (*,2000) rname, -iflag, ival
      CASE DEFAULT
        WRITE (*,3000) iflag, rname
      END SELECT

      STOP

1000  FORMAT ('On entry to ',A, &
       ' two or more array argument sizes do not conform')
2000  FORMAT ('On entry to ',A,' argument number',I3, &
       ' had the illegal value of ',I5)
3000  FORMAT ('Unknown error code ',I5,' raised by routine ',A)

    END SUBROUTINE blas_error
```

F77 error handler

The mandated API of the routine is:

```
      SUBROUTINE BLAS_ERROR( RNAME, IFLAG, IVAL )
      CHARACTER*(*) RNAME
      INTEGER IFLAG, IVAL
```

A possible implementation would be:

```
      SUBROUTINE BLAS_ERROR( RNAME, IFLAG, IVAL )
      CHARACTER*(*) RNAME
```

```
      INTEGER IFLAG, IVAL

      IF( IFLAG.LT.0 ) THEN
         WRITE(*,1000) RNAME, -IFLAG, IVAL
      ELSE
         WRITE(*,2000) IFLAG, RNAME
      END IF
      STOP

1000  FORMAT('On entry to ',A, ' parameter number', I3,
     $       ' had the illegal value of', I)
2000  FORMAT('Unknown error code ',I,' raised by routine',A)
      END
```

C error handler

The mandated API of the routine is:

```
void BLAS_error(char *rname, int iflag, int ival, char *form, ...)
```

A possible implementation would be:

```
#include <stdio.h>
#include <stdarg.h>
void BLAS_error(char *rname, int iflag, int ival, char *form, ...)
{
   va_list argptr;

   va_start(argptr, form);
   fprintf(stderr, "Error #%d from routine %s:\n", iflag, rname);
   if (form) vfprintf(stderr, form, argptr);
   else if (iflag < 0)
      fprintf(stderr,
         "   Parameter number %d to routine %s had the illegal value %d\n",
              -iflag, rname, ival);
   else fprintf(stderr, "   Unknown error code %d from routine %s\n",
                iflag, rname);
   exit(iflag);
}
```

Chapter 2

Dense and Banded BLAS

2.1 Overview and Functionality

This chapter defines the functionality and language bindings for the dense and banded BLAS routines, addressing mathematical operations with scalars, vectors and dense, banded, and triangular matrices but not sparse data structures.

The chapter is organized as follows. Sections 2.1.1, 2.1.2, and 2.1.3 list in tabular form the functionality of the proposed routines. Unless otherwise specified, the operations apply to both real and complex arguments. For the sake of compactness the complex operators are omitted, so that whenever a transpose operation is given the conjugate transpose should also be assumed for the complex case. Section 2.2 defines the matrix storage schemes. Section 2.3 discusses general interface issues, and sections 2.4, 2.5, and 2.6 detail the interface issues for the respective language bindings – Fortran 95, Fortran 77, and C. Section 2.7 discusses issues concerning the numerical accuracy of the BLAS. And lastly, sections 2.8.2 – 2.8.10 present the language bindings for the proposed routines.

2.1.1 Scalar and Vector Operations

This section lists scalar and vector operations. The functionality tables are organized as follows. Table 2.1 lists the scalar and vector reduction operations, table 2.2 lists the rotation operations, table 2.3 lists the vector operations, and table 2.4 lists vector operations involving only data movement. Notation in the tables is defined in section 1.4, and details of the data structures are discussed in section 2.2. Vector norms are defined in Appendix A.1. The language bindings are presented in sections 2.8.2, 2.8.4, and 2.8.5.

2.1.2 Matrix-Vector Operations

This section lists the matrix-vectors operations in functionality table 2.5. Unless otherwise specified, the operations apply to both real and complex arguments. For the sake of compactness the complex operators are omitted, so that whenever a transpose operation is given both the conjugate and conjugate transpose should also be assumed for the complex case.

The matrix T represents an upper or lower triangular matrix, which can be unit or non-unit triangular. D represents a diagonal matrix. Notation in the tables is defined in section 1.4, and details of the data structures are discussed in section 2.2. The language bindings are presented in section 2.8.6.

Dot product	$r \leftarrow \beta r + \alpha x^T y$	DOT
Vector norms	$r \leftarrow \|\|x\|\|_1, r \leftarrow \|\|x\|\|_{1R}$, $r \leftarrow \|\|x\|\|_2$, $r \leftarrow \|\|x\|\|_\infty, r \leftarrow \|\|x\|\|_{\infty R}$	NORM
Sum	$r \leftarrow \sum_i x_i$	SUM
Min value & location	$k, x_k, ; k = \arg\min_i x_i$	MIN_VAL
Min abs value & location	$k, x_k, k = \arg\min_i(\|Re(x_i)\| + \|Im(x_i)\|)$	AMIN_VAL
Max value & location	$k, x_k, ; k = \arg\max_i x_i$	MAX_VAL
Max abs value & location	$k, x_k, k = \arg\max_i(\|Re(x_i)\| + \|Im(x_i)\|)$	AMAX_VAL
Sum of squares	$(ssq, scl) \leftarrow \sum x_i^2$, $ssq \cdot scl^2 = \sum x_i^2$	SUMSQ

Table 2.1: Reduction Operations

Generate Givens rotation	$(c, s, r) \leftarrow \mathrm{rot}(a, b)$	GEN_GROT
Generate Jacobi rotation	$(a, b, c, s) \leftarrow \mathrm{jrot}(x, y, z)$	GEN_JROT
Generate Householder transform	$(\alpha, x, \tau) \leftarrow \mathrm{house}(\alpha, x)$, $H = I - \alpha u u^T$	GEN_HOUSE

Table 2.2: Generate Transformations

Reciprocal Scale	$x \leftarrow x/\alpha$	RSCALE
Scaled vector accumulation	$y \leftarrow \alpha x + \beta y$,	AXPBY
Scaled vector addition	$w \leftarrow \alpha x + \beta y$	WAXPBY
Combined axpy & dot product	$\begin{cases} \hat{w} \leftarrow w - \alpha v \\ r \leftarrow \hat{w}^T u \end{cases}$	AXPY_DOT
Apply plane rotation	$\begin{pmatrix} x & y \end{pmatrix} \leftarrow \begin{pmatrix} x & y \end{pmatrix} R$	APPLY_GROT

Table 2.3: Vector Operations

Copy	$y \leftarrow x$	COPY
Swap	$y \leftrightarrow x$	SWAP
Sort vector	$x \leftarrow \mathrm{sort}(x)$	SORT
Sort vector & return index vector	$(p, x) \leftarrow \mathrm{sort}(x)$	SORTV
Permute vector	$x \leftarrow Px$	PERMUTE

Table 2.4: Data Movement with Vectors

2.1.3 Matrix Operations

This section lists single matrix operations, matrix-matrix operations, and matrix operations involving data movement. The functionality tables are organized as follows. Table 2.6 lists single matrix operations and matrix operations that involve $O(n^2)$ floating point operations, Table 2.7 lists the $O(n^3)$ matrix-matrix floating point operations and Table 2.8 lists those matrix floating point operations that involve only data movement. Unless otherwise specified, the operations apply to both real and complex arguments. For the sake of compactness the complex operators are omit-

Matrix vector product	$y \leftarrow \alpha Ax + \beta y$	GE,GB,SY,HE, SB,HB,SP,HP	MV
	$y \leftarrow \alpha A^T x + \beta y$	GE,GB	MV
	$x \leftarrow \alpha Tx,\ x \leftarrow \alpha T^T x$	TR,TB,TP	MV
Summed matrix vector multiplies	$y \leftarrow \alpha Ax + \beta Bx$	GE	SUM_MV
Multiple matrix vector multiplies	$\begin{cases} x \leftarrow T^T y \\ w \leftarrow Tz \end{cases}$	TR	MVT
	$\begin{cases} x \leftarrow \beta A^T y + z \\ w \leftarrow \alpha Ax \end{cases}$	GE	MVT
Multiple mv mults & low rank updates	$\begin{cases} \hat{A} \leftarrow A + u_1 v_1^T + u_2 v_2^T \\ x \leftarrow \beta \hat{A}^T y + z \\ w \leftarrow \alpha \hat{A} x \end{cases}$	GE	MVER
Triangular solve	$x \leftarrow \alpha T^{-1} x,\ x \leftarrow \alpha T^{-T} x$	TR,TB,TP	SV
Rank one updates	$A \leftarrow \alpha x y^T + \beta A$	GE	R
and symmetric ($A = A^T$)	$A \leftarrow \alpha x x^T + \beta A$	SY,HE,SP,HP	R
rank one & two updates	$A \leftarrow (\alpha x) y^T + y(\alpha x)^T + \beta A$	SY,HE,SP,HP	R2

Table 2.5: Matrix-Vector Operations

ted, so that whenever a transpose operation is given both the conjugate and conjugate transpose should also be assumed for the complex case. The matrix T represents an upper or lower triangular matrix, which can be unit or non-unit triangular. D, D_L, and D_R represent diagonal matrices, and J is a symmetric tridiagonal matrix. Notation in the tables is defined in section 1.4, and details of the data structures are discussed in section 2.2. Matrix norms are defined in Appendix A.2. The language bindings are listed in sections 2.8.6, 2.8.7, 2.8.8, and 2.8.9.

Matrix norms	$r \leftarrow \|\|A\|\|_1, r \leftarrow \|\|A\|\|_{1R}, r \leftarrow \|\|A\|\|_F,$ $r \leftarrow \|\|A\|\|_\infty, r \leftarrow \|\|A\|\|_{\infty R},$ $r \leftarrow \|\|A\|\|_{max}, r \leftarrow \|\|A\|\|_{maxR}$	GE,GB,SY,HE,SB,HB, SP,HP,TR,TB,TP	_NORM
Diagonal scaling	$A \leftarrow DA,\ A \leftarrow AD$	GE,GB	_DIAG_SCALE
	$A \leftarrow D_L A D_R$	GE,GB	_LRSCALE
	$A \leftarrow DAD$	SY,HE,SB,HB,SP,HP	_LRSCALE
	$A \leftarrow A + BD$	GE,GB	_DIAG_SCALE_ACC
Matrix acc and scale	$B \leftarrow \alpha A + \beta B,\ B \leftarrow \alpha A^T + \beta B$	GE,GB,SY,SB, SP,TR,TB,TP	_ACC
Matrix add and scale	$C \leftarrow \alpha A + \beta B$	GE,GB,SY,SB, SP,TR,TB,TP	_ADD

Table 2.6: Matrix Operations – $O(n^2)$ floating point operations

Matrix matrix product	$C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha A^T B + \beta C$ $C \leftarrow \alpha AB^T + \beta C,\ C \leftarrow \alpha A^T B^T + \beta C$	GE	MM
	$C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha BA + \beta C$	SY,HE	MM
Triangular multiply	$B \leftarrow \alpha TB,\ B \leftarrow \alpha BT$ $B \leftarrow \alpha T^T B,\ B \leftarrow \alpha BT^T$	TR	MM
Triangular solve	$B \leftarrow \alpha T^{-1} B,\ B \leftarrow \alpha BT^{-1}$ $B \leftarrow \alpha T^{-T} B,\ B \leftarrow \alpha BT^{-T}$	TR	SM
Symmetric rank k & $2k$ updates ($C = C^T$)	$C \leftarrow \alpha AA^T + \beta C,\ C \leftarrow \alpha A^T A + \beta C$	SY,HE	RK
	$C \leftarrow \alpha AJA^T + \beta C,\ C \leftarrow \alpha A^T JA + \beta C$	SY,HE	_TRIDIAG_RK
	$C \leftarrow (\alpha A)B^T + B(\alpha A)^T + \beta C,$ $C \leftarrow (\alpha A)^T B + B^T(\alpha A) + \beta C$	SY,HE	R2K
	$C \leftarrow (\alpha AJ)B^T + B(\alpha AJ)^T + \beta C,$ $C \leftarrow (\alpha AJ)^T B + B^T(\alpha AJ) + \beta C$	SY,HE	_TRIDIAG_R2K

Table 2.7: Matrix-Matrix Operations – $O(n^3)$ floating point operations

Matrix copy	$B \leftarrow A$	GE,GB,SY,HE,SB,HB,SP,HP,TR,TB,TP	_COPY
	$B \leftarrow A^T$	GE,GB	_COPY
Matrix transpose	$A \leftarrow A^T$	GE	_TRANS
Permute Matrix	$A \leftarrow PA,\ A \leftarrow AP$	GE	_PERMUTE

Table 2.8: Data Movement with Matrices

2.2 Matrix Storage Schemes

The following matrix storage schemes are used:

- column-based and row-based storage in a contiguous array;
- packed storage for symmetric, Hermitian or triangular matrices;
- band storage for band matrices;

In the examples below, $*$ indicates an array element that need not be set and is not referenced by the BLAS routines. Elements that "need not be set" are never read, written to, or otherwise accessed by the BLAS routines. The examples illustrate only the relevant part of the arrays; array arguments may of course have additional rows or columns, according to the usual rules for passing array arguments in C or Fortran.

2.2.1 Conventional Storage

The default scheme for storing matrices in the Fortran 95 and Fortran 77 interfaces is the one described in subsection 2.5.3: a matrix A is stored in a two-dimensional array A, with matrix element a_{ij} stored in array element $\mathrm{A}(i,j)$, assuming one-based indexing.

For the C language interfaces, matrices may be stored column-wise or row-wise as described in subsection 2.6.6: a matrix A is stored in a one-dimensional array A, with matrix element a_{ij} stored column-wise in array element $\mathrm{A}(i + j * lda)$ or row-wise in array element $\mathrm{A}(j + i * lda)$, assuming zero-based indexing.

If a matrix is **triangular** (upper or lower, as specified by the argument `uplo`), only the elements of the relevant triangle are accessed. The remaining elements of the array need not be set. Such elements are indicated by $*$ in the examples below. For example, assuming zero-based indexing and $n = 3$:

`order`	`uplo`	Triangular matrix A	Storage in array A
blas_colmajor	blas_upper	$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ & a_{11} & a_{12} \\ & & a_{22} \end{pmatrix}$	$a_{00}\ *\ *\ a_{01}\ a_{11}\ *\ a_{02}\ a_{12}\ a_{22}$
blas_rowmajor	blas_upper	$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ & a_{11} & a_{12} \\ & & a_{22} \end{pmatrix}$	$a_{00}\ a_{01}\ a_{02}\ *\ a_{11}\ a_{12}\ *\ *\ a_{22}$
blas_colmajor	blas_lower	$\begin{pmatrix} a_{00} & & \\ a_{10} & a_{11} & \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$	$a_{00}\ a_{10}\ a_{20}\ *\ a_{11}\ a_{21}\ *\ *\ a_{22}$
blas_rowmajor	blas_lower	$\begin{pmatrix} a_{00} & & \\ a_{10} & a_{11} & \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$	$a_{00}\ *\ *\ a_{10}\ a_{11}\ *\ a_{20}\ a_{21}\ a_{22}$

Routines that handle **symmetric** or **Hermitian** matrices allow for either the upper or lower triangle of the matrix (as specified by `uplo`) to be stored in the corresponding elements of the array; the remaining elements of the array need not be set. For example, when $n = 3$:

`order`	`uplo`	Hermitian matrix A	Storage in array A
blas_colmajor	blas_upper	$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ \bar{a}_{01} & a_{11} & a_{12} \\ \bar{a}_{02} & \bar{a}_{12} & a_{22} \end{pmatrix}$	$a_{00}\ *\ *\ a_{01}\ a_{11}\ *\ a_{02}\ a_{12}\ a_{22}$
blas_rowmajor	blas_upper	$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ \bar{a}_{01} & a_{11} & a_{12} \\ \bar{a}_{02} & \bar{a}_{12} & a_{22} \end{pmatrix}$	$a_{00}\ a_{01}\ a_{02}\ *\ a_{11}\ a_{12}\ *\ *\ a_{22}$
blas_colmajor	blas_lower	$\begin{pmatrix} a_{00} & \bar{a}_{10} & \bar{a}_{20} \\ a_{10} & a_{11} & \bar{a}_{21} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$	$a_{00}\ a_{10}\ a_{20}\ *\ a_{11}\ a_{21}\ *\ *\ a_{22}$
blas_rowmajor	blas_lower	$\begin{pmatrix} a_{00} & \bar{a}_{10} & \bar{a}_{20} \\ a_{10} & a_{11} & \bar{a}_{21} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$	$a_{00}\ *\ *\ a_{10}\ a_{11}\ *\ a_{20}\ a_{21}\ a_{22}$

2.2.2 Packed Storage

Symmetric, Hermitian or triangular matrices may be stored more compactly, if the relevant triangle (again as specified by `uplo`) is packed **by columns or rows** in a one-dimensional array. In the BLAS, arrays that hold matrices in packed storage, have names ending in 'P'. So, in the case of zero-based addressing as in C, we have the following formulas (For one-based addressing, as in Fortran, replace i by $i-1$ and j by $j-1$ in these formulas).

- if `uplo = blas_upper` then
  - if `order = blas_colmajor`, a_{ij} is stored in $\mathrm{AP}(i + j(j+1)/2)$ for $i \le j$;
  - if `order = blas_rowmajor`, a_{ij} is stored in $\mathrm{AP}(j + i(2n-i-1)/2)$ for $i \le j$;
- if `uplo = blas_lower` then
  - if `order = blas_colmajor`, a_{ij} is stored in $\mathrm{AP}(i + j(2n-j-1)/2)$ for $j \le i$.
  - if `order = blas_rowmajor`, a_{ij} is stored in $\mathrm{AP}(j + i(i+1)/2)$ for $j \le i$.

For example, assuming zero-based indexing:

`order`	`uplo`	Triangular matrix A	Packed storage in array ap
blas_colmajor	blas_upper	$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ & a_{11} & a_{12} \\ & & a_{22} \end{pmatrix}$	$a_{00}\ \underbrace{a_{01}\ a_{11}}\ \underbrace{a_{02}\ a_{12}\ a_{22}}$
blas_rowmajor	blas_upper	$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ & a_{11} & a_{12} \\ & & a_{22} \end{pmatrix}$	$\underbrace{a_{00}\ a_{01}\ a_{02}}\ \underbrace{a_{11}\ a_{12}}\ a_{22}$
blas_colmajor	blas_lower	$\begin{pmatrix} a_{00} & & \\ a_{10} & a_{11} & \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$	$\underbrace{a_{00}\ a_{10}\ a_{20}}\ \underbrace{a_{11}\ a_{21}}\ a_{22}$
blas_rowmajor	blas_lower	$\begin{pmatrix} a_{00} & & \\ a_{10} & a_{11} & \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$	$a_{00}\ \underbrace{a_{10}\ a_{11}}\ \underbrace{a_{20}\ a_{21}\ a_{22}}$

Note that for real or complex symmetric matrices, packing the upper triangle by columns is equivalent to packing the lower triangle by rows; packing the lower triangle by columns is equivalent to packing the upper triangle by rows. For complex Hermitian matrices, packing the upper triangle by columns is equivalent to packing the conjugate of the lower triangle by rows; packing the lower triangle by columns is equivalent to packing the conjugate of the upper triangle by rows.

2.2.3 Band Storage

For Fortran (column-major storage), an m-by-n band matrix with kl subdiagonals and ku superdiagonals may be stored compactly in a two-dimensional array with $kl+ku+1$ rows and n columns. Columns of the matrix are stored in corresponding columns (contiguous storage dimension) of the array, and diagonals of the matrix are stored in rows (non-contiguous or strided dimension) of the array. This storage scheme should be used in practice only if $kl, ku \ll \min(m,n)$, although BLAS routines work correctly for all values of kl and ku. In the BLAS, arrays that hold matrices in band storage have names ending in 'B'.

To be precise, for column-major storage, a_{ij} is stored in $\mathrm{AB}(ku+i-j, j)$ for $\max(0, j-ku) \le i \le \min(m-1, j+kl)$. For row-major storage, a_{ij} is stored in $\mathrm{AB}(i, kl+j-i)$ for $\max(0, j-ku) \le i \le \min(n-1, j+kl)$. For example, assuming column-major storage, when $m = n = 5$, $kl = 2$ and $ku = 1$:

Band matrix A	Band storage in array AB
$\begin{pmatrix} a_{00} & a_{01} & & & \\ a_{10} & a_{11} & a_{12} & & \\ a_{20} & a_{21} & a_{22} & a_{23} & \\ & a_{31} & a_{32} & a_{33} & a_{34} \\ & & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\begin{matrix} * & a_{01} & a_{12} & a_{23} & a_{34} \\ a_{00} & a_{11} & a_{22} & a_{33} & a_{44} \\ a_{10} & a_{21} & a_{32} & a_{43} & * \\ a_{20} & a_{31} & a_{42} & * & * \end{matrix}$

The elements marked $*$ in the upper left and lower right corners of the array AB need not be set, and are not referenced by BLAS routines.

For C (row-major storage), `order = blas_rowmajor`, the rows of the matrix are stored in corresponding rows (contiguous storage dimension) of the array, and diagonals of the matrix are stored in columns (non-contiguous or strided dimension) of the array. The m-by-n band matrix with kl subdiagonals and ku superdiagonals is stored in a one-dimensional array with n rows and $kl+ku+1$ columns, strided by lda. The padding with elements marked $*$ is now shifted to ensure that rows of the matrix are stored contiguously. Refer to section B.2.12 for full details.

Triangular band matrices are stored in the same format, with either $kl = 0$ if upper triangular, or $ku = 0$ if lower triangular.

For Fortran 77, and symmetric or Hermitian band matrices with kd subdiagonals or superdiagonals, only the upper or lower triangle (as specified by `uplo`) need be stored:

- if `uplo = blas_upper`, a_{ij} is stored in $\mathrm{AB}(kd+i-j, j)$ for $\max(0, j-kd) \le i \le j$;
- if `uplo = blas_lower`, a_{ij} is stored in $\mathrm{AB}(i-j, j)$ for $j \le i \le \min(n-1, j+kd)$.

For example, assuming zero-based indexing and $n = 5$ and $kd = 2$:

`uplo`	Hermitian band matrix A	Band storage in array AB
blas_upper	$\begin{pmatrix} a_{00} & a_{01} & a_{02} & & \\ \bar{a}_{01} & a_{11} & a_{12} & a_{13} & \\ \bar{a}_{02} & \bar{a}_{12} & a_{22} & a_{23} & a_{24} \\ & \bar{a}_{13} & \bar{a}_{23} & a_{33} & a_{34} \\ & & \bar{a}_{24} & \bar{a}_{34} & a_{44} \end{pmatrix}$	$\begin{matrix} * & * & a_{02} & a_{13} & a_{24} \\ * & a_{01} & a_{12} & a_{23} & a_{34} \\ a_{00} & a_{11} & a_{22} & a_{33} & a_{44} \end{matrix}$
blas_lower	$\begin{pmatrix} a_{00} & \bar{a}_{10} & \bar{a}_{20} & & \\ a_{10} & a_{11} & \bar{a}_{21} & \bar{a}_{31} & \\ a_{20} & a_{21} & a_{22} & \bar{a}_{32} & \bar{a}_{42} \\ & a_{31} & a_{32} & a_{33} & \bar{a}_{43} \\ & & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\begin{matrix} a_{00} & a_{11} & a_{22} & a_{33} & a_{44} \\ a_{10} & a_{21} & a_{32} & a_{43} & * \\ a_{20} & a_{31} & a_{42} & * & * \end{matrix}$

Similarly, for C (row-major storage), `order = blas_rowmajor`, the contiguous dimension (rows) of the matrix is stored in the contiguous dimension (rows) of the array, strided by lda. And pictorially, the one-dimensional array is the transpose of the AB storage as depicted above. The padding with elements marked $*$ is now shifted to ensure that rows of the matrix are stored contiguously. Refer to section B.2.12 for full details.

2.2.4 Unit Triangular Matrices

Some BLAS routines have an option to handle unit triangular matrices (that is, triangular matrices with diagonal elements = 1). This option is specified by an argument `diag`. If `diag = blas_unit_diag` (Unit triangular)), the array elements corresponding to the diagonal elements of the matrix are not referenced by the BLAS routines. The storage scheme for the matrix (whether conventional, packed or band) remains unchanged, as described in subsection 2.2.1.

2.2.5 Representation of a Householder Matrix

An elementary reflector (or elementary **Householder matrix**) H of order n is a unitary matrix of the form

$$H = I - \tau v v^H \tag{2.1}$$

where τ is a scalar, and v is an n-vector, with $|\tau|^2 ||v||_2^2 = 2\mathrm{Re}(\tau)$; v is often referred to as the **Householder vector**. Often v has several leading or trailing zero elements, but for the purpose of this discussion assume that H has no such special structure.

This representation agrees with what is used in LAPACK [6] (which differs from those used in LINPACK [23] or EISPACK [48, 32]) sets $v_1 = 1$; hence v_1 need not be stored. In real arithmetic, $1 \leq \tau \leq 2$, except that $\tau = 0$ implies $H = I$.

In complex arithmetic, τ may be complex, and satisfies $1 \leq \mathrm{Re}(\tau) \leq 2$ and $|\tau - 1| \leq 1$. Thus a complex H is not Hermitian (as it is in other representations), but it is unitary, which is the important property. The advantage of allowing τ to be complex is that, given an arbitrary complex vector x, H can be computed so that

$$H^H x = \beta(1, 0, \ldots, 0)^T$$

with *real* β. This is useful, for example, when reducing a complex Hermitian matrix to real symmetric tridiagonal matrix, or a complex rectangular matrix to real bidiagonal form.

2.2.6 Representation of a Permutation Matrix

An n-by-n permutation matrix P is represented as a product of at most n interchange permutations. An interchange permutation E is a permutation obtained by swapping two rows of the identity matrix. An efficient way to represent a general permutation matrix P is with an integer vector p of length n. In other words, $P = E_n \ldots E_1$ and each E_i is the identity with rows i and p_i interchanged.

Do $\mathsf{i} = 0$ to $\mathsf{n} - 1$	or	Do $\mathsf{i} = \mathsf{n} - 1$ to 0
$\quad \mathsf{x(i))} \leftrightarrow \mathsf{x(\ p(i))\ ))}$		$\quad \mathsf{x(i))} \leftrightarrow \mathsf{x(\ p(i))\ ))}$
End do		End do

2.3 Interface Issues

2.3.1 Naming Conventions

The naming conventions adopted for the routines are as defined in section 1.4.6.

2.3.2 Argument Aliasing

Correctness is only guaranteed if output arguments are not aliased with any other arguments.

2.4 Interface Issues for Fortran 95

Some of the functions in the tables of this chapter can be replaced by simple array expressions and assignments in Fortran 95, without loss of convenience or performance (assuming a reasonable degree of optimization by the compiler). Fortran 95 also allows groups of related functions to be merged together, each group being covered by a single interface.

The following sections discuss the indexing base for vector and matrix operands, the features of the Fortran 95 language that are used, the matrix storage schemes that are supported, and error handling.

We strongly recommend that optional arguments be supplied by keyword, not by position, since the order in which they are described may differ from the order in which they appear in the argument list.

2.4.1 Fortran 95 Modules

Refer to Appendix A.4 for the Fortran 95 module `blas_dense`. The module `blas_operator_arguments` contains the derived type values, and separate modules are supplied with explicit interfaces to the routines. If the module `blas_dense` is accessed by a `USE` statement in any program which makes calls to these BLAS routines, then those calls can be checked by the compiler for errors in the numbers or types of arguments.

2.4.2 Indexing

The Fortran 95 interface returns indices in the range $1 \leq I \leq N$ (where N is the number of entries in the dimension in question, and I is the index). This allows functions returning indices to be directly used to index standard arrays.

Likewise, for routines returning an index within a vector or matrix operand, this reference point is indexed starting at one.

2.4.3 Design of the Fortran 95 Interfaces

The proposed design utilizes the following features of the Fortran 95 language.

Generic interfaces: all procedures are accessed through *generic* interfaces. A single generic name covers several specific instances whose arguments may differ in the following properties:

data type (real or complex).

precision (or equivalently, kind type parameter "kind-value"). However, all real or complex arguments must have the same precision. We allow both single and double precision.

rank Some arguments may either have rank 2 (to store a matrix) or rank 1 (to store a vector). In other cases an argument may be either a rank 1 array or a scalar.

different argument lists Some of the arguments are optional. If one of these arguments does not appear in the calling sequence, a predefined value or a predefined action is assumed. Table 2.9 contains the predefined value or action for these arguments.

Assumed-shape arrays: all array arguments are *assumed-shape* arrays, which must have the exact shape required to store the corresponding matrix or vector. Hence arguments to specify array-dimensions or problem-dimensions are not required. The procedures assume that the supplied arrays have valid and consistent shapes. Zero dimensions (implying empty arrays) are allowed.

This means that, for a vector operand, the offset and stride are not needed as arguments. The actual argument corresponding to a n-length vector dummy argument could be:

actual argument	comments
x(ix:ix+(n-1)*incx)	ix$\neq$ 1 and incx$\neq$ 1
x(1:1+(n-1)*incx)	ix= 1 and incx$\neq$ 1
x(0:(n-1)*incx)	ix= 0 and incx$\neq$ 1
x(ix:ix+n-1)	ix$\neq$ 1 and incx= 1
x(1:n)	ix= 1 and incx= 1
x	if x is declared with shape (n), i.e. x(n)
x(ix)	where ix is an integer vector of n elements containing valid indices of x
a(:,j)	column j of a two-dimension array assuming that it has n rows (SIZE(a,1) $= n$)
a(i,:)	row i of a two-dimension array assuming that it has n columns (SIZE(a,2) $= n$)

Derived types: In the Fortran 95 bindings, we use dummy arguments whose actual argument must be a named constant of a derived type, which is defined within the BLAS module (and accessible via the BLAS module).

2.4.4 Matrix Storage Schemes

The matrix storage schemes for the Fortran 95 interfaces are as described in section 2.2. As with the Fortran 77 interfaces, only column-major storage is permitted. However, assumed-shape arrays are used instead of assumed-size arrays.

For a general banded matrix, a, three arguments a, m and kl are used to define the matrix since ku is defined from the shape of the matrix and kl ($ku = SIZE(a,1) - kl - 1$). For a symmetric banded matrix, a Hermitian banded matrix or triangular banded matrix, a, only a is used as an argument to define the matrix as the band width is defined from the shape of the matrix and is equal to $SIZE(a,1) - 1$ and $m = n$.

2.4.5 Format of the Fortran 95 bindings

Each interface is summarized in the form of a SUBROUTINE statement (or in few cases a FUNCTION statement), in which all of the potential arguments appear. Arguments which need not be supplied are grouped after the mandatory arguments and enclosed in square brackets, for example:

```
SUBROUTINE axpby( x, y [, alpha] [, beta] )
  <type>(<wp>), INTENT (IN) :: x(:)
  <type>(<wp>), INTENT (INOUT) :: y(:)
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
```

The default value for β is 1.0 or (1.0,0.0).

As generic interfaces are used, floating point variables that can be REAL or COMPLEX are denoted by the keyword <type> which designates the data type for the operand

```
<type> ::= REAL | COMPLEX
```

In some routines, however, some of the floating point arguments must be of a specific data type. If this is the case, then the argument type REAL or COMPLEX is used.

The precision of the floating point variable is denoted by <wp> (i.e., "working precision") where

```
<wp> ::= KIND(1.0) | KIND(1.0D0)
```

and KIND(1.0) and KIND(1.0D0) represent single precision and double precision, respectively.

Some arguments may either have rank 2 (to store a matrix) or rank 1 (to store a vector). In this case, the following notation is used:

```
<bb> ::= b(:,:)  | b(:)
```

The same notation is used in the case of an argument that may either have rank 1 or is a scalar.

```
<bb> ::= b(:)  | b
```

Fortran 95 bindings use assumed shape arrays. The actual arguments must have the correct dimension. For all the procedures that contain array arguments the shape of the array arguments is given in detail after the specification. For example the specification of the SUBROUTINE axpby given above is followed by:

x and y have shape (n)

which indicates that both arrays x and y must be rank 1 with the same number of elements.

The calling sequence may be followed by a table which lists the different variants of the operation, depending either on the ranks of some of the arguments or on the optional arguments. The scalar values alpha and beta take the defaults given in the following table:

Argument	default value in real case	default value in complex case
alpha	1.0	(1.0,0.0)
beta	0.0 OR 1.0	(0.0,0.0) OR (1.0,0.0)

Procedures that contain the optional scalar `beta` state the default value for `beta` only if it is 1.0 or (1.0,0.0), otherwise the default is assumed to be 0.0 or (0.0,0.0).

The following table shows the notation that is used for the values of optional arguments (since `alpha` and `beta` are also optional, for example):

Dummy argument	Notation in table	Named constant	Default value
`norm`	1-norm	`blas_one_norm`	`blas_one_norm`
	1R-norm	`blas_real_one_norm`	
	2-norm	`blas_two_norm`	
	Frobenius-norm	`blas_frobenius_norm`	
	inf-norm	`blas_inf_norm`	
	real-inf-norm	`blas_real_inf_norm`	
	max-norm	`blas_max_norm`	
	real-max-norm	`blas_real_max_norm`	
`sort`	sort in decreasing order	`blas_decreasing_order`	`blas_increasing_order`
	sort in increasing order	`blas_increasing_order`	
`side`	L	`blas_left_side`	`blas_left`
	R	`blas_right_side`	
`uplo`	U	`blas_upper`	`blas_upper`
	L	`blas_lower`	
`trans`*x*	N	`blas_no_trans`	`blas_no_trans`
	T	`blas_trans`	
	H	`blas_conj_trans`	
`conj`		`blas_no_conj`	`blas_no_conj`
		`blas_conj`	
`diag`	N	`blas_non_unit_diag`	`blas_non_unit_diag`
	U	`blas_unit_diag`	
`jrot`	inner rotation	`blas_jrot_inner`	`blas_jrot_inner`
	outer rotation	`blas_jrot_outer`	
	sorted rotation	`blas_jrot_sorted`	

Table 2.9: Default values of Operator Arguments

2.4.6 Error Handling

The Fortran 95 interface must supply an error-handling routine `blas_error`. The API for this error-handling routine is defined in section 1.8. By default, this routine will print an error message and stop execution. The user may modify the action performed by the error-handling routine, and this modification must be documented.

The following values of arguments are invalid and will be flagged by the error-handling routine:

- Any value of the operator arguments whose meaning is not specified in the language-dependent section is invalid;

Routine-specific error conditions are listed in the respective language bindings.

2.5 Interface Issues for Fortran 77

Unless explicitly stated, the Fortran 77 binding is consistent with ANSI standard Fortran 77. There are several points where this standard diverges from the ANSI Fortran 77 standard. In particular:

- Subroutine names are not limited to six significant characters.
- Subroutine names contain an underscore.
- Subroutines may use the INCLUDE statement for include files.

Section 2.5.2 discusses the indexing of vector and matrix operands. Section A.5 defines the operator arguments, section 2.5.3 defines array arguments, and section 2.2 lists the matrix storage schemes that are supported. Section 2.5.5 details the format of the language binding, and section 2.5.6 discusses error handling.

2.5.1 Fortran 77 Include File

Refer to Appendix A.5 for details of the Fortran 77 include file `blas_namedconstants.h`.

2.5.2 Indexing

The Fortran 77 interface returns indices in the range $1 \leq I \leq N$ (where N is the number of entries in the dimension in question, and I is the index). This allows functions returning indices to be directly used to index standard arrays.

Likewise, for routines returning an index within a vector or matrix operand, this reference point is indexed starting at one.

2.5.3 Array Arguments

Vector arguments are permitted to have a storage spacing between elements. This spacing is specified by an increment argument. For example, suppose a vector x having components x_i, $i = 1, \ldots, N$, is stored in an array $X()$ with increment argument $INCX$. If $INCX > 0$ then x_i is stored in $X(1 + (i-1) * INCX)$. If $INCX < 0$ then x_i is stored in $X(1 + (N-i) * |INCX|)$. This method of indexing when $INCX < 0$ avoids negative indices in the array $X()$ and thus permits the subprograms to be written in Fortran 77. $INCX = 0$ is an illegal value.

Each two-dimensional array argument is immediately followed in the argument list by its leading dimension, whose name has the form LD<array-name>. If a two-dimensional array A of dimension (LDA,N) holds an m-by-n matrix A, then A(i, j) holds a_{ij} for $i = 1, \ldots, m$ and $j = 1, \ldots, n$ (LDA must be at least m). See Section 2.2 for more about storage of matrices.

Note that array arguments are usually declared in the software as assumed-size arrays (last dimension *), for example:

```
REAL A( LDA, * )
```

although the documentation gives the dimensions as (LDA,N). The latter form is more informative since it specifies the required minimum value of the last dimension. However an assumed-size array declaration has been used in the software in order to overcome some limitations in the Fortran 77 standard. In particular it allows the routine to be called when the relevant dimension (N, in this case) is zero. However actual array dimensions in the calling program must be at least 1 (LDA in this example).

2.5.4 Matrix Storage Schemes

The matrix storage schemes for the Fortran 77 interfaces are as described in section 2.2. Only column-major storage is permitted, and all two-dimensional arrays are assumed-size arrays.

2.5.5 Format of the Fortran 77 bindings

Each interface is summarized in the form of a `SUBROUTINE` statement (or a `FUNCTION` statement). The declarations of the arguments are listed in alphabetical order. For example,

```
SUBROUTINE BLAS_xAXPBY( N, ALPHA, X, INCX, BETA, Y, INCY )
INTEGER            INCX, INCY, N
<type>             ALPHA, BETA
<type>             X( * ), Y( * )
```

Floating point variables are denoted by the keyword `<type>` which designates the data type for the operand (`REAL`, `DOUBLE PRECISION`, `COMPLEX`, or `COMPLEX*16`). This data type will agree with the **x** letter in the naming convention of the routine. In some routines, however, not all floating point variables will be of the same data type. If this is the case, then a variable may be denoted by the keyword `<ctype>` to restrict the data type to `COMPLEX` or `COMPLEX*16`, or `<rtype>` to restrict the data type to `REAL` or `DOUBLE PRECISION`.

The language binding will be followed by any restrictions dictated for this interface.

2.5.6 Error Handling

The Fortran 77 interface supplies an error-handling routine `BLAS_ERROR`, as defined in section 1.8. By default, this routine will print an error message and stop execution. The user may modify the action performed by the error-handling routine, and this modification must be documented.

The following values of arguments are invalid and will be flagged by the error-handling routine:

- Any value of the operator arguments whose meaning is not specified in the language-dependent section is invalid;
- incw=0 or incx=0 or incy=0 or incz=0;
- lda, ldb, ldc, or ldt < 1;
- lda $<$ m if the matrix is an $m \times n$ general matrix and trans = blas_no_trans;
- lda $<$ n if the matrix is an $m \times n$ general matrix and trans = blas_trans;
- lda $<$ n if the matrix is an $n \times n$ square, symmetric, or triangular matrix;
- lda $<$ kl + ku + 1, if the matrix is an $m \times n$ general band matrix;
- lda $<$ k+1, if the matrix is an $n \times n$ symmetric or triangular band matrix with k super- or subdiagonals;

Routine-specific error conditions are listed in the respective language bindings.

2.6 Interface Issues for C

The interface is expressed in terms of ANSI/ISO C. Most platforms provide ANSI/ISO C compilers, and if this is not the case, free ANSI/ISO C compilers are available (eg., `gcc`).

Section 2.6.2 discusses the indexing of vector and matrix operands. Section A.6 defines the operator arguments, section 2.6.3 discusses the handling of complex data types, section 2.6.4 defines return values of complex functions, and section 2.6.5 provides the rule for argument aliasing. Section 2.6.6 defines array arguments, and section 2.6.7 lists the matrix storage schemes that are supported. Section 2.6.8 details the format of the language binding, and section 2.6.9 discusses error handling.

2.6.1 C Include File

The C interface to the BLAS has a standard include file, called `blas_dense.h`, which minimally contains the values of the enumerated types and ANSI/ISO C prototypes for all BLAS routines. Refer to Appendix A.6 for details of the C include files pertaining to Chapters 2 – 4.

> *Advice to implementors.* Note that the vendor is not constrained to using precisely this include file; only the enumerated type definitions are fully specified. The implementor is free to make any other changes which are not apparent to the user. For instance, all matrix dimensions might be accepted as `size_t` instead of `int`, or the implementor might choose to make some routines in-line. (*End of advice to implementors.*)

2.6.2 Indexing

The C interface returns indices in the range $0 \leq I \leq N-1$ (where N is the number of entries in the dimension in question, and I is the index). This allows functions returning indices to be directly used to index standard arrays.

Likewise, for routines returning an index within a vector or matrix operand, this reference point is indexed starting at zero.

2.6.3 Handling of complex data types

All complex arguments are accepted as `void *`. A complex element consists of two consecutive memory locations of the underlying data type (i.e., `float` or `double`), where the first location contains the real component, and the second contains the imaginary component.

An ISO/IEC committee (WG14/X3J11) [38] is presently working on an extension to ANSI/ISO C which defines complex data types. This extension is one of several additions to the C language, commonly referred to as the C9X standard. The definition of a complex element is the same as given above, and so the handling of complex types by this interface will not need to be changed when ANSI/ISO C standard is extended.

2.6.4 Return values of complex functions

BLAS routines which return complex values in Fortran 77 are instead recast as subroutines in the C interface, with the return value being an output parameter added to the end of the argument list. This allows the output parameter to be accepted as a void pointer, as discussed above.

2.6.5 Aliasing of arguments

Unless specified otherwise, only input-only arguments (specified with the `const` qualifier), may be legally aliased on a call to the C interface to the BLAS.

2.6.6 Array arguments

Arrays are constrained to being contiguous in memory. They are accepted as pointers, not as arrays of pointers. Note that this means that two-dimensional array arguments in C are not permitted.

All BLAS routines which take one or more two dimensional arrays as arguments receive one extra parameter as their first argument. This argument is an enumerated type (see Appendix A). If this parameter is set to `blas_rowmajor`, it is assumed that elements within a row of the array(s) are contiguous in memory, while elements within array columns are separated by a constant stride given in the `stride` parameter (this parameter corresponds to the leading dimension [e.g. `LDA`] in the Fortran 77 interface).

If the order is given as `blas_colmajor`, elements within array columns are assumed to be contiguous, with elements within array rows separated by `stride` memory elements.

Note that there is only one `blas_order_type` parameter to a given routine: all array operands are required to use the same ordering.

2.6.7 Matrix Storage Schemes

The matrix storage schemes for the C interfaces are as described in section 2.2. Column-major storage and row-major storage in a contiguous array are permitted.

2.6.8 Format of the C bindings

Each routine is summarized in the form of an ANSI/ISO C prototype. For example:

```
void BLAS_xaxpby( int n, SCALAR_IN alpha, const ARRAY x, int incx,
                  SCALAR_IN beta, ARRAY y, int incy );
```

Floating point variables are denoted by the keywords `SCALAR` and `ARRAY` to denote scalar arguments and array arguments respectively.

`SCALAR_IN`	`ARRAY` or `SCALAR_INOUT`	operation
`float` or `double`	`float *` or `double *`	real arithmetic
`const void *`	`void *`	complex arithmetic

This data type will agree with the `x` letter in the naming convention of the routine. In some routines, however, not all floating point variables will be of the same data type. If this is the case, then a variable may be denoted by the keyword `RSCALAR_INOUT`, `CSCALAR_INOUT`, `RARRAY`, or `CARRAY`, to restrict the data type to real or complex arithmetic, respectively.

The language binding will be followed by any restrictions dictated for this interface.

2.6.9 Error Handling

The C interface must supply an error-handling routine `BLAS_error`. This error-handling routine will accept as input a character string, specifying the name of the routine where the error occurred.

By default, this routine will print an error message and stop execution. The user may modify the action performed by the error-handling routine, and this modification must be documented.

The following values of arguments are invalid and will be flagged by the error-handling routine:

- Any value of the operator arguments whose meaning is not specified in the language-dependent section is invalid;
- incw=0 or incx=0 or incy=0 or incz=0;
- lda, ldb, ldc, or ldt < 1;
- lda < m if the matrix is an $m \times n$ general matrix;
- lda < n if the matrix is an $n \times n$ square, symmetric, or triangular matrix;
- lda < kl + ku + 1, if the matrix is an $m \times n$ general band matrix;
- lda < k+1, if the matrix is an $n \times n$ symmetric or triangular band matrix with k super- or subdiagonals;

Routine-specific error conditions are listed in the respective language bindings.

2.7 Numerical Accuracy and Environmental Enquiry

With a few exceptions that are explicitly described below, no particular computational order is mandated by the function specifications. In other words, any algorithm that produces results "close enough" to the usual algorithms presented in a standard book on matrix computations [33, 19, 35] is acceptable. For example, Strassen's algorithm may be used for matrix multiplication, even though it can be significantly less accurate than conventional matrix multiplication for some pairs of matrices [35]. Also, matrix multiplication may be implemented either as $C = (\alpha \cdot A) \cdot B + (\beta \cdot C)$ or $C = \alpha \cdot (A \cdot B) + (\beta \cdot C)$ or $C = A \cdot (\alpha \cdot B) + (\beta \cdot C)$, whichever is convenient.

To use the error bounds in [33, 19, 35] and elsewhere, certain machine parameters are needed to describe the accuracy of the arithmetic.

These are described in detail in section 1.6, and returned by function `xFPINFO`. Its calling sequence in C or Fortran 77 is

```
result = xFPINFO( CMACH )
```

where `x=S` for single precision and `x=D` for double precision. In Fortran 95, its calling sequence is

```
result = FPINFO( CMACH, float )
```

where the "kind" of float (single or double) is used to determine the kind of the result. The argument CMACH can take on the following named constant values (the exact representations are language dependent, with CMACH available as a derived type in Fortran 95, named integer constants in Fortran 77, and an enumerated type in C). The named constant values are defined in sections A.4, A.5, and A.6. CMACH has the analogous meaning (see footnote 4 in section 1.6 for a discussion) as the like-named character argument of the LAPACK auxiliary routine xLAMCH:

Value of CMACH	Name of floating point parameter (see Table 1.9 in section 1.6 for details)
blas_base	BASE
blas_t	T
blas_rnd	RND
blas_ieee	IEEE
blas_emin	EMIN
blas_emax	EMAX
blas_eps	EPS
blas_prec	PREC
blas_underflow	UN
blas_overflow	OV
blas_sfmin	SFMIN

Here are the exceptional routines where we ask for particularly careful implementations to avoid unnecessary over/underflows, that could make the output unnecessarily inaccurate or unreliable. The details of each routine are described with the language dependent calling sequences. Model implementations that avoid unnecessary over/underflows are based on corresponding LAPACK auxiliary routines, NAG routines, or cited reports.

1. Reduction Operations (Section 2.8.2)
   - NORM (Vector norms)
   - SUMSQ (Sum of squares)
2. Generate Transformations (Section 2.8.3)
   - GEN_GROT (Generate Givens rotation)
   - GEN_JROT (Generate Jacobi rotation)
   - GEN_HOUSE (Generate Householder transform)
3. Vector Operations (Section 2.8.4)
   - RSCALE (Reciprocal scale)
4. Matrix Operations (Section 2.8.7)
   - {GE,GB,SY,HE,SB,SP,HP,TR,TB,TP}_NORM (Matrix norms)

2.8 Language Bindings

Each specification of a routine will correspond to an operation outlined in the functionality tables. Operations are organized analogous to the order in which they are presented in the functionality tables. The specification will have the form:

NAME (*multi-word description of operation*) < *mathematical representation* >

Optional brief textual description of the functionality including any restrictions that apply to all language bindings.

- Fortran 95 binding
- Fortran 77 binding
- C binding

2.8.1 Overview

- Reduction Operations (section 2.8.2)
  - DOT (Dot product)
  - NORM (Vector norms)
  - SUM (Sum)
  - MIN_VAL (Min value & location)
  - AMIN_VAL (Min absolute value & location)
  - MAX_VAL (Max value & location)
  - AMAX_VAL (Max absolute value & location)
  - SUMSQ (Sum of squares)
- Generate Transformations (section 2.8.3)
  - GEN_GROT (Generate Givens rotation)
  - GEN_JROT (Generate Jacobi rotation)
  - GEN_HOUSE (Generate Householder transform)
- Vector Operations (section 2.8.4)
  - RSCALE (Reciprocal Scale)
  - AXPBY (Scaled vector accumulation)
  - WAXPBY (Scaled vector addition)
  - AXPY_DOT (Combined AXPY and DOT)
  - APPLY_GROT (Apply plane rotation)
- Data Movement with Vectors (section 2.8.5)
  - COPY (Vector copy)
  - SWAP (Swap)
  - SORT (Sort vector)
  - SORTV (Sort vector & return index vector)
  - PERMUTE (Permute vector)
- Matrix-Vector Operations (section 2.8.6)
  - {GE,GB}MV (Matrix vector product)
  - {SY,SB,SP}MV (Symmetric matrix vector product)
  - {HE,HB,HP}MV (Hermitian matrix vector product)

  - {TR,TB,TP}MV (Triangular matrix vector product)
  - GE_SUM_MV (Summed matrix vector multiplies)
  - GEMVT (Combined matrix vector product)
  - TRMVT (Combined triangular matrix vector product)
  - GEMVER (Combined matrix vector product with a rank 2 update)
  - {TR,TB,TP}SV (Triangular solve)
  - GER (Rank one update)
  - {SY,SP}R (Symmetric rank one update)
  - {HE,HP}R (Hermitian rank one update)
  - {SY,SP}R2 (Symmetric rank two update)
  - {HE,HP}R2 (Hermitian rank two update)

- Matrix Operations (section 2.8.7)
  - {GE,GB,SY,HE,SB,HB,SP,HP,TR,TB,TP}_NORM (Matrix norms)
  - {GE,GB}_DIAG_SCALE (Diagonal scaling)
  - {GE,GB}_LRSCALE (Two-sided diagonal scaling)
  - {SY,SB,SP}_LRSCALE (Two-sided diagonal scaling of a symmetric matrix)
  - {HE,HB,HP}_LRSCALE (Two-sided diagonal scaling of a Hermitian matrix)
  - {GE,GB}_DIAG_SCALE_ACC (Diagonal scaling and accumulation)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_ACC (Matrix accumulation and scale)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_ADD (Matrix add and scale)

- Matrix-Matrix Operations (section 2.8.8)
  - GEMM (General Matrix Matrix product)
  - SYMM (Symmetric matrix matrix product)
  - HEMM (Hermitian matrix matrix product)
  - TRMM (Triangular matrix matrix multiply)
  - TRSM (Triangular solve)
  - SYRK (Symmetric rank-k update)
  - HERK (Hermitian rank-k update)
  - SY_TRIDIAG_RK (Symmetric rank-k update with tridiagonal matrix)
  - HE_TRIDIAG_RK (Hermitian rank-k update with tridiagonal matrix)
  - SYR2K (Symmetric rank-2k update)
  - HER2K (Hermitian rank-2k update)
  - SY_TRIDIAG_R2K (Symmetric rank-2k update with tridiagonal matrix)
  - HE_TRIDIAG_R2K (Hermitian rank-2k update with tridiagonal matrix)

- Data Movement with Matrices (section 2.8.9)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_COPY (Matrix copy)

- {HE,HB,HP}_COPY (Matrix copy)
- {GE}_TRANS (Matrix transposition)
- {GE}_PERMUTE (Permute matrix)

- Environmental Enquiry (section 2.8.10)
  - FPINFO (Environmental enquiry)

2.8.2 Reduction Operations

DOT (Dot Product) $$x, y \in \mathbb{R}^n, r \leftarrow \beta r + \alpha x^T y = \beta r + \alpha \sum_{i=0}^{n-1} x_i y_i$$

$$x, y \in \mathbb{C}^n, r \leftarrow \beta r + \alpha x^T y = \beta r + \alpha \sum_{i=0}^{n-1} x_i y_i \text{ or } r \leftarrow \beta r + \alpha x^H y = \beta r + \alpha \sum_{i=0}^{n-1} \bar{x}_i y_i$$

The routine DOT adds the scaled dot product of two vectors x and y into a scaled scalar r. The routine returns immediately if n is less than zero, or, if beta is equal to one and either alpha or n is equal to zero. If alpha is equal to zero then x and y are not read. Similarly, if beta is equal to zero, r is not read. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler.

When x and y are complex vectors, the vector components x_i are used unconjugated or conjugated as specified by the operator argument conj. If x and y are real vectors, the operator argument conj has no effect.

- Fortran 95 binding:

```
SUBROUTINE dot( x, y, r [, conj] [, alpha] [, beta] )
  <type>(<wp>), INTENT (IN) :: x(:), y(:)
  <type>(<wp>), INTENT (INOUT) :: r
  TYPE (blas_conj_type), INTENT(IN), OPTIONAL :: conj
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
where
  x and y have shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xDOT( CONJ, N, ALPHA, X, INCX, BETA, Y, INCY, R )
INTEGER           CONJ, INCX, INCY, N
<type>            ALPHA, BETA, R
<type>            X( * ), Y( * )
```

- C binding:

```
void BLAS_xdot( enum blas_conj_type conj, int n, SCALAR_IN alpha,
                const ARRAY x, int incx, SCALAR_IN beta, const ARRAY y,
                int incy, SCALAR_INOUT r );
```

NORM (Vector norms) $r \leftarrow ||x||_1,\ ||x||_{1R},\ ||x||_2, ||x||_\infty,\ or\ ||x||_{\infty R}$

The routine NORM computes the $||\cdot||_1$, $||\cdot||_{1R}$, $||\cdot||_2$, $||\cdot||_\infty$, or $||\cdot||_{\infty R}$ of a vector x depending on the value passed as the norm operator argument.

If `norm = blas_frobenius_norm`, an error flag is not raised, and the two-norm is returned to the user. If n is less than or equal to zero, this routine returns immediately with the output scalar r set to zero. The resulting scalar r is always real and its value is as defined in section 2.1.1. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
REAL(<wp>) FUNCTION norm( x [, norm] )
  <type>(<wp>), INTENT (IN) :: x(:)
  TYPE (blas_norm_type), INTENT (IN), OPTIONAL :: norm
where
  x has shape (n)
```

- Fortran 77 binding:

```
<rtype> FUNCTION BLAS_xNORM( NORM, N, X, INCX )
INTEGER            INCX, N, NORM
<type>             X( * )
```

- C binding:

```
void BLAS_xnorm( enum blas_norm_type norm, int n, const ARRAY x,
                 int incx, RSCALAR_INOUT r );
```

SUM (Sum) $r \leftarrow \sum_{i=0}^{n-1} x_i$

The routine SUM computes the sum of the entries of a vector x. If n is less than or equal to zero, this routine returns immediately with the output scalar r set to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
<type>(<wp>) FUNCTION sum( x )
  <type>(<wp>), INTENT (IN) :: x(:)
where
  x has shape (n)
```

This is the same as the Fortran 95 intrinsic function SUM.

- Fortran 77 binding:

```
<type> FUNCTION BLAS_xSUM( N, X, INCX )
INTEGER            INCX, N
<type>             X( * )
```

- C binding:

```
void BLAS_xsum( int n, const ARRAY x, int incx, SCALAR_INOUT sum );
```

MIN_VAL (Min value & location) k, x_k such that $k = \arg\min_{0\le i<n} x_i$

The routine MIN_VAL finds the smallest component of a real vector x and determines the smallest offset or index k such that $x_k = \min_{0\le i<n} x_i$. This value x_k is returned by the routine and denoted by $\arg\min_{0\le i<\mathsf{n}} x_i$ below. When the value of the n argument is less than or equal to zero, the routine should initialize the output scalars k to the largest invalid index or offset value (negative one or zero) and r to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

Advice to users. The routine MIN_VAL strictly operates on real vectors. This routine is not defined for complex vectors. (*End of advice to users.*)

- Fortran 95 binding:

```
SUBROUTINE min_val( x, k, r )
  REAL(<wp>), INTENT (IN) :: x(:)
  INTEGER, INTENT (OUT) :: k
  REAL(<wp>), INTENT (OUT) :: r
where
  x has shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xMIN_VAL( N, X, INCX, K, R )
INTEGER            INCX, K, N
<rtype>            R
<rtype>            X( * )
```

- C binding:

```
void BLAS_xmin_val( int n, const RARRAY x, int incx, int k,
                    RSCALAR_INOUT r );
```

AMIN_VAL (Min absolute value & location) k, x_k such that $k = \arg\min_{0\le i<n} (|Re(x_i)| + |Im(x_i)|)$

The routine AMIN_VAL finds the offset or index of the smallest component of a vector x and also returns the smallest component of the vector x with respect to the absolute value. When the value of the n argument is less than or equal to zero, the routine should initialize the output scalars k to the largest invalid index or offset value (negative one or zero) and r to zero. The resulting scalar r is always real. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE amin_val( x, k, r )
  <type>(<wp>), INTENT (IN) :: x(:)
  INTEGER, INTENT (OUT) :: k
  REAL(<wp>), INTENT (OUT) :: r
where
  x has shape (n)
```

  A Fortran 95 interface was defined for this routine since it would have been too expensive using Fortran 95 intrinsics.

- Fortran 77 binding:

```
SUBROUTINE BLAS_xAMIN_VAL( N, X, INCX, K, R )
INTEGER            INCX, K, N
<rtype>            R
<type>             X( * )
```

- C binding:

```
void BLAS_xamin_val( int n, const ARRAY x, int incx, int k,
                     RSCALAR_INOUT r );
```

MAX_VAL (Max value & location) k, x_k such that $k = \arg\max_{0 \le i < n} x_i$

The routine MAX_VAL finds the largest component of a real vector x and determines the smallest offset or index k such that $x_k = \max_{0 \le i < \mathsf{n}} x_i$. This value x_k is returned by the routine and denoted by $\arg\max_{0 \le i < \mathsf{n}} x_i$ below. When the value of the n argument is less than or equal to zero, the routine should initialize the output scalars k to the largest invalid index or offset value (negative one or zero) and r to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

Advice to users. The routine MAX_VAL strictly operates on real vectors. This routine is not defined for complex vectors. (*End of advice to users.*)

- Fortran 95 binding:

```
SUBROUTINE max_val( x, k, r )
  REAL(<wp>), INTENT (IN) :: x(:)
  INTEGER, INTENT (OUT) :: k
  REAL(<wp>), INTENT (OUT) :: r
where
  x has shape (n)
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xMAX_VAL( N, X, INCX, K, R )
      INTEGER            INCX, K, N
      <rtype>            R
      <rtype>            X( * )
```

- C binding:

```
void BLAS_xmax_val( int n, const RARRAY x, int incx, int k,
                    RSCALAR_INOUT r );
```

AMAX_VAL (Max absolute value & location) k, x_k such that $k = \arg\max_{0 \leq i < n} (|Re(x_i)| + |Im(x_i)|)$

The routine AMAX_VAL finds the offset or index of the largest component of a vector x and also returns the largest component of the vector x with respect to the absolute value. When the value of the n argument is less than or equal to zero, the routine should initialize the output scalars k to the largest invalid index or offset value (negative one or zero) and r to zero. The resulting scalar r is always real. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
      SUBROUTINE amax_val( x, k, r )
        <type>(<wp>), INTENT (IN) :: x(:)
        INTEGER, INTENT (OUT) :: k
        REAL(<wp>), INTENT (OUT) :: r
      where
        x has shape (n)
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xAMAX_VAL( N, X, INCX, K, R )
      INTEGER            INCX, K, N
      <rtype>            R
      <type>             X( * )
```

- C binding:

```
void BLAS_xamax_val( int n, const ARRAY x, int incx, int k,
                     RSCALAR_INOUT r );
```

SUMSQ (Sum of squares) $(scl, ssq) \leftarrow \sum x_i^2,$

The routine SUMSQ returns the values scl and ssq such that

$$scl^2 * ssq = scale^2 * sumsq + \sum_{i=0}^{n-1} (Re(x_i)^2 + Im(x_i)^2),$$

The value of *sumsq* is assumed to be at least unity and the value of *ssq* will then satisfy $1.0 \le ssq \le (sumsq + n)$ when x is a real vector and $1.0 \le ssq \le (sumsq + 2n)$ when x is a complex vector. *scale* is assumed to be non-negative and *scl* returns the value

$$scl = \max_{0 \le i < n} (scale, abs(Re(x_i)), abs(Im(x_i))).$$

scale and *sumsq* must be supplied on entry in scl and ssq respectively. scl and ssq are overwritten by *scl* and *ssq* respectively. The arguments scl and ssq are therefore always real scalars. If scl is less than zero or ssq is less than one, an error flag is set and passed to the error handler. If n is less than or equal to zero, this routine returns immediately with scl and ssq unchanged. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

> *Advice to implementors.* High-quality implementations of this routine SUMSQ should be accurate. The subroutine SLASSQ of the LAPACK [6] software library is an example of such an accurate implementation. High-quality implementations should document the accuracy of the algorithms used in this routine so as to alleviate the portability problems this represents. (*End of advice to implementors.*)

- Fortran 95 binding:

```
SUBROUTINE sumsq( x, ssq, scl )
  <type>(<wp>), INTENT (IN) :: x(:)
  REAL(<wp>), INTENT (INOUT) :: ssq, scl
where
  x has shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xSUMSQ( N, X, INCX, SSQ, SCL )
INTEGER             INCX, N
<rtype>             SCL, SSQ
<type>              X( * )
```

- C binding:

```
void BLAS_xsumsq( int n, const ARRAY x, int incx, RSCALAR_INOUT ssq,
                  RSCALAR_INOUT scl );
```

2.8.3 Generate Transformations

GEN_GROT (Generate Givens rotation) $(c, s, r) \leftarrow \mathrm{rot}(a, b)$

The routine GEN_GROT constructs a Givens plane rotation so that

$$\begin{pmatrix} c & s \\ -\bar{s} & c \end{pmatrix} \cdot \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix},$$

where c is always a real scalar and $c^2 + |s|^2$ is equal to one. The scalars a and b are unchanged on exit. c, s and r are defined precisely as follows, where we use the function

$$\text{sign}(x) \equiv \begin{cases} x/|x| & \text{if } x \neq 0 \\ 1 & \text{if } x = 0 \end{cases}$$

Defining Givens rotations:

if $b = 0$ (includes the case $a = b = 0$)
  $c = 1$
  $s = 0$
  $r = a$
elseif $a = 0$ (b must be nonzero)
  $c = 0$
  $s = \text{sign}(\bar{b})$
  $r = |b|$
else (a and b both nonzero)
  $c = |a|/\sqrt{|a|^2 + |b|^2}$
  $s = \text{sign}(a)\bar{b}/\sqrt{|a|^2 + |b|^2}$
  $r = \text{sign}(a)\sqrt{|a|^2 + |b|^2}$
endif

When a and b are real, $\bar{b}$ may be replaced by b.

> *Advice to implementors.* High-quality implementations of this routine GEN_GROT should be accurate. We recommend one of the implementations described in [9]. We note that the above definition of Givens rotations matches the one in the subroutine CLARTG of the LAPACK [6] software library, but differs slightly from the definitions used in LAPACK routines SLARTG, SLARGV and CLARGV. LAPACK routines using these slightly different Givens rotations continue to function correctly [9]. (*End of advice to implementors.*)

- Fortran 95 binding:

```
SUBROUTINE gen_grot( a, b, c, s, r )
  <type>(<wp>), INTENT (IN) :: a, b
  REAL(<wp>), INTENT (OUT) :: c
  <type>(<wp>), INTENT (OUT) :: s, r
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xGEN_GROT( A, B, C, S, R )
<rtype>             C
<type>              A, B, R, S
```

- C binding:

```
void BLAS_xgen_grot( SCALAR_IN a, SCALAR_IN b, RSCALAR_INOUT c,
                     SCALAR_INOUT s, SCALAR_INOUT r );
```

GEN_JROT (Generate Jacobi rotation) $(a, b, c, s) \leftarrow \text{jrot}(x, y, z)$

The routine GEN_JROT constructs a Jacobi rotation so that

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} = \begin{pmatrix} c & \bar{s} \\ -s & c \end{pmatrix} \cdot \begin{pmatrix} x & y \\ \bar{y} & z \end{pmatrix} \cdot \begin{pmatrix} c & -\bar{s} \\ s & c \end{pmatrix},$$

If JROT = blas_inner_rotation, then the rotation is chosen so that $c \geq \frac{1}{\sqrt{2}}$.
If JROT = blas_outer_rotation, then the rotation is chosen so that $0 \leq c \leq \frac{1}{\sqrt{2}}$.
If JROT = blas_sorted_rotation, then the rotation is chosen so that $abs(a) \geq abs(b)$.

On entry, the argument x contains the value x, and on exit it contains the value a. On entry, the argument y contains the value y. On entry, the argument z contains the value z, and on exit it contains the value b. The arguments x and z are real scalars, and argument c is always a real scalar and $c^2 + |s|^2$ is equal to one.

Advice to implementors. High-quality implementations of this routine GEN_JROT should document the accuracy of the algorithms used in those functions so as to alleviate the portability problems this represents. (See NAG routine F06BEF). (*End of advice to implementors.*)

- Fortran 95 binding:

```
SUBROUTINE gen_jrot( x, y, z, c, s [, jrot]  )
  REAL(<wp>), INTENT (INOUT) :: x, z
  <type>(<wp>), INTENT (IN) :: y
  REAL(<wp>), INTENT (OUT) :: c
  <type>(<wp>), INTENT (OUT) :: s
  TYPE (blas_jrot_type), INTENT(IN), OPTIONAL :: jrot
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xGEN_JROT( JROT, X, Y, Z, C, S )
INTEGER            JROT
<rtype>            C, X, Z
<type>             S, Y
```

- C binding:

```
void BLAS_xgen_jrot( enum blas_jrot_type jrot, RSCALAR_INOUT x,
                     SCALAR_IN y, RSCALAR_INOUT z, RSCALAR_INOUT c,
                     SCALAR_INOUT s );
```

GEN_HOUSE (Generate Householder transform) $(\alpha, x, \tau) \leftarrow \text{house}(\alpha, x)$,

The routine GEN_HOUSE generates an elementary reflector H of order n, such that

$$H\begin{pmatrix} \alpha \\ x \end{pmatrix} = \begin{pmatrix} \beta \\ 0 \end{pmatrix} \text{ and } H^H H = I,$$

where α and β are scalars, and x is an $(n-1)$-element vector. β is always a real scalar. H is represented in the form

$$H = I - \tau \begin{pmatrix} 1 \\ v \end{pmatrix} \begin{pmatrix} 1 & v^T \end{pmatrix},$$

where τ is a scalar and v is a $(n-1)$-element vector. τ is called the Householder scalar and $\begin{pmatrix} 1 \\ v \end{pmatrix}$ the Householder vector. Note that when x is a complex vector, H is not Hermitian. If the elements of x are zero, and α is real, then τ is equal to zero and H is taken to be the unit matrix. Otherwise, the real part of τ is greater than or equal to one and less than or equal to two. Moreover, the absolute value of the quantity $\tau - 1$ is less than or equal to one.

On exit, the scalar argument alpha is overwritten with the value of the scalar β. Similarly, the vector argument x is overwritten with the vector v. If n is less than or equal to zero, this function returns immediately with the output scalar tau set to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

> *Advice to implementors.* High-quality implementations of this routine GEN_HOUSE should be accurate. The subroutines SLARFG and CLARFG of the LAPACK [6] software library are examples of such an accurate implementation. High-quality implementations should document the accuracy of the algorithms used in those functions so as to alleviate the portability problems this represents. (*End of advice to implementors.*)

> *Advice to users.* Routines to apply Householder transformations are not provided. The subroutines xORMyy of the LAPACK [6] software library are examples of such implementations. (*End of advice to users.*)

- Fortran 95 binding:

```
SUBROUTINE gen_house( alpha, x, tau )
  <type>(<wp>), INTENT (INOUT) :: alpha
  <type>(<wp>), INTENT (INOUT) :: x(:)
  <type>(<wp>), INTENT (OUT) :: tau
where
  x has shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xGEN_HOUSE( N, ALPHA, X, INCX, TAU )
INTEGER            INCX, N
<type>             ALPHA, TAU
<type>             X( * )
```

- C binding:

```
void BLAS_xgen_house( int n, SCALAR_INOUT alpha, ARRAY x, int incx,
                      SCALAR_INOUT tau );
```

2.8.4 Vector Operations

RSCALE (Reciprocal Scale) $x \leftarrow x/\alpha$

The routine RSCALE scales the entries of a vector x by the real scalar $1/\alpha$. The scalar α is always real and supposed to be nonzero. This should be done without overflow or underflow as long as the final result x/α does not overflow or underflow. If n is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero or if alpha is equal to zero, an error flag is set and passed to the error handler.

> *Advice to implementors.* High-quality implementations of this routine RSCALE should be accurate. The subroutine xRSCL of the LAPACK [6] software library is an example of such an accurate implementation. High-quality implementations should document the accuracy of the algorithms used in those functions so as to alleviate the portability problems this represents. (*End of advice to implementors.*)

- Fortran 95 binding:

```
SUBROUTINE rscale( alpha, x )
  REAL(<wp>), INTENT (IN) :: alpha
  <type>(<wp>), INTENT (INOUT) :: x(:)
where
  x has shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xRSCALE( N, ALPHA, X, INCX )
INTEGER            INCX, N
<rtype>            ALPHA
<type>             X( * )
```

- C binding:

```
void BLAS_xrscale( int n, RSCALAR_IN alpha, ARRAY x, int incx );
```

AXPBY (Scaled vector accumulation) $y \leftarrow \alpha x + \beta y$

The routine AXPBY scales the vector x by α and the vector y by β, adds these two vectors to one another and stores the result in the vector y. If n is less than or equal to zero, or if α is equal to zero and β is equal to one, this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE axpby( x, y [, alpha] [, beta] )
  <type>(<wp>), INTENT (IN) :: x(:)
```

```
    <type>(<wp>), INTENT (INOUT) :: y(:)
    <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
  where
    x and y have shape (n)
```

The default value for β is 1.0 or (1.0,0.0).

- Fortran 77 binding:

```
  SUBROUTINE BLAS_xAXPBY( N, ALPHA, X, INCX, BETA, Y, INCY )
  INTEGER           INCX, INCY, N
  <type>            ALPHA, BETA
  <type>            X( * ), Y( * )
```

- C binding:

```
void BLAS_xaxpby( int n, SCALAR_IN alpha, const ARRAY x, int incx,
                  SCALAR_IN beta, ARRAY y, int incy );
```

WAXPBY (Scaled vector addition) $w \leftarrow \alpha x + \beta y$

The routine WAXPBY scales the vector x by α and the vector y by β, adds these two vectors to one another and stores the result in the vector w. If n is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy or incw less than zero is permitted. However, if either incx or incy or incw is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
  SUBROUTINE waxpby( x, y, w [, alpha] [, beta] )
    <type>(<wp>), INTENT (IN) :: x(:), y(:)
    <type>(<wp>), INTENT (OUT) :: w(:)
    <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
  where
    x, y and w have shape (n)
```

The default value for β is 1.0 or (1.0,0.0).

- Fortran 77 binding:

```
  SUBROUTINE BLAS_xWAXPBY( N, ALPHA, X, INCX, BETA, Y, INCY, W, INCW )
  INTEGER           INCW, INCX, INCY, N
  <type>            ALPHA, BETA
  <type>            W( * ), X( * ), Y( * )
```

- C binding:

```
void BLAS_xwaxpby( int n, SCALAR_IN alpha, const ARRAY x, int incx,
                   SCALAR_IN beta, const ARRAY y, int incy, ARRAY w,
                   int incw );
```

AXPY_DOT (Combined AXPY and DOT) $\hat{w} \leftarrow w - \alpha v, r \leftarrow \hat{w}^T u$

The routine combines an axpy and a dot product. w is decremented by a multiple of v. A dot product is then computed with the decremented w.

Advice to implementors. Note that $\hat{w}$ may be used to update r before it is written back to memory. This optimization, which accelerates algorithms like modified Gram-Schmidt orthogonalization, is the justification for AXPY_DOT, which could otherwise be implemented by calls to AXPBY and DOT. (*End of advice to implementors.*)

If n is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incw or incv or incu less than zero is permitted. However, if either incw or incv or incu is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE axpy_dot( w, v, u, r [, alpha] )
  <type>(<wp>), INTENT (IN) :: v(:), u(:)
  <type>(<wp>), INTENT (INOUT) :: w(:)
  <type>(<wp>), INTENT (OUT) :: r
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha
where
  u, v and w have shape (n)
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xAXPY_DOT( N, ALPHA, W, INCW, V, INCV, U, INCU,
     $                           R )
      INTEGER            INCW, INCV, INCU, N
      <type>             ALPHA, R
      <type>             W( * ), V( * ), U( * )
```

- C binding:

```
void BLAS_xaxpy_dot( int n, SCALAR_IN alpha, ARRAY w, int incw,
                     const ARRAY v, int incv, const ARRAY u, int incu,
                     SCALAR_INOUT r );
```

APPLY_GROT (Apply plane rotation) $\begin{pmatrix} x & y \end{pmatrix} \leftarrow \begin{pmatrix} x & y \end{pmatrix} R$

The routine APPLY_GROT applies a plane rotation to the vectors x and y. When the vectors x and y are real vectors, the scalars c and s are real scalars. When the vectors x and y are complex vectors, c is a real scalar and s is a complex scalar. This routine computes

$$\forall\, i \in [0 \ldots n-1], \begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} c & s \\ -\bar{s} & c \end{pmatrix} \cdot \begin{pmatrix} x_i \\ y_i \end{pmatrix}.$$

If n is less than or equal to zero or if c is one and s is zero, the routine APPLY_GROT returns immediately. As described in section 2.5.3, the value of incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE apply_grot( c, s, x, y )
  REAL(<wp>), INTENT (IN) :: c
  <type>(<wp>), INTENT (IN) :: s
  <type>(<wp>), INTENT (INOUT) :: x(:), y(:)
where
  x and y have shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xAPPLY_GROT( N, C, S, X, INCX, Y, INCY )
INTEGER            INCX, INCY, N
<rtype>            C
<type>             S
<type>             X( * ), Y( * )
```

- C binding:

```
void BLAS_xapply_grot( int n, RSCALAR_IN c, SCALAR_IN s, ARRAY x, int incx,
                       ARRAY y, int incy );
```

2.8.5 Data Movement with Vectors

COPY (Vector copy) $y \leftarrow x$

The routine COPY copies the vector x into the vector y. If n is less than or equal to zero, the routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE copy( x, y )
  <type>(<wp>), INTENT (IN) :: x(:)
  <type>(<wp>), INTENT (OUT) :: y(:)
where
  x and y have shape (n)
```

This is similar to the Fortran 95 assignment y=x.

- Fortran 77 binding:

```
SUBROUTINE BLAS_xCOPY( N, X, INCX, Y, INCY )
INTEGER            INCX, INCY, N
<type>             X( * ), Y( * )
```

- C binding:

```
void BLAS_xcopy( int n, const ARRAY x, int incx, ARRAY y, int incy );
```

SWAP (Swap) $y \leftrightarrow x$

The routine SWAP interchanges the vectors x and y. If n is less than or equal to zero, the routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE swap( x, y )
  <type>(<wp>), INTENT (INOUT) :: x(:), y(:)
where
  x and y have shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xSWAP( N, X, INCX, Y, INCY )
INTEGER            INCX, INCY, N
<type>             X( * ), Y( * )
```

- C binding:

```
void BLAS_xswap( int n, ARRAY x, int incx, ARRAY y, int incy );
```

SORT (Sort vector) $x \leftarrow \text{sort}(x)$

The routine SORT sorts the entries of a real vector x in increasing or decreasing order and overwrites this vector x with the sorted vector. If n is less than or equal to zero, the function returns immediately. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

Advice to users. The routine SORT strictly operates on real vectors. This routine is not defined for complex vectors. (*End of advice to users.*)

Advice to implementors. The subroutine xLASRT of the LAPACK [6] software library is an example of such a routine. (*End of advice to implementors.*)

- Fortran 95 binding: *Refer to SORTV specification*
- Fortran 77 binding:

```
SUBROUTINE BLAS_xSORT( SORT, N, X, INCX )
INTEGER            INCX, N, SORT
<rtype>            X( * )
```

- C binding:

```
void BLAS_xsort( enum blas_sort_type sort, int n, RARRAY x, int incx );
```

SORTV (Sort vector & return index vector) $(p, x) \leftarrow \text{sort}(x)$

The routine SORTV sorts the entries of a real vector x in increasing or decreasing order and overwrites this vector x with the sorted vector $(x = P * x)$. If n is less than or equal to zero, the routine returns immediately. As described in section 2.5.3, the value incx or incp less than zero is permitted. However, if either incx or incp is equal to zero, an error flag is set and passed to the error handler.

The representation of the permutation vector p is described in section 2.2.6.

Advice to users. The routine SORTV strictly operates on real vectors. This routine is not defined for complex vectors. (*End of advice to users.*)

- Fortran 95 binding:

```
SUBROUTINE sortv( x [, sort] [, p] )
  REAL(<wp>), INTENT (INOUT) :: x(:)
  TYPE (blas_sort_type), INTENT (IN), OPTIONAL :: sort
  INTEGER, INTENT (OUT), OPTIONAL :: p(:)
where
  x and p have shape (n)
```

The functionality of SORT is covered by SORTV.

- Fortran 77 binding:

```
SUBROUTINE BLAS_xSORTV( SORT, N, X, INCX, P, INCP )
INTEGER            INCP, INCX, N, SORT
INTEGER            P( * )
<rtype>            X( * )
```

- C binding:

```
void BLAS_xsortv( enum blas_sort_type sort, int n, RARRAY x, int incx,
                  int *p, int incp );
```

PERMUTE (Permute vector) $x \leftarrow Px$

The routine PERMUTE permutes the entries of a vector x according to the permutation vector p. If n is less than or equal to zero, the routine returns immediately. As described in section 2.5.3, the value incx or incp less than zero is permitted. However, if either incx or incp is equal to zero, an error flag is set and passed to the error handler.

The encoding of the permutation P in the vector p follows the same conventions as the ones described above for the routine SORTV. Refer to section 2.2.6 for complete details.

- Fortran 95 binding:

```
SUBROUTINE permute( x, p )
  <type>(<wp>), INTENT (INOUT) :: x(:)
  INTEGER, INTENT (IN) :: p(:)
where
  x and p have shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xPERMUTE( N, P, INCP, X, INCX )
INTEGER             INCP, INCX, N
INTEGER             P( * )
<type>              X( * )
```

  The value of INCP may be positive or negative. A negative value of INCP applies the permutation in the opposite direction.

- C binding:

```
void BLAS_xpermute( int n, const int *p, int incp, ARRAY x, int incx );
```

  The value of incp may be positive or negative. A negative value of incp applies the permutation in the opposite direction.

2.8.6 Matrix-Vector Operations

In the following section, $op(X)$ denotes X, or X^T or X^H where X is a matrix.

{GE,GB}MV (Matrix vector product) $y \leftarrow \alpha op(A)x + \beta y$

The routines perform a matrix vector multiply $y \leftarrow \alpha op(A)x + \beta y$ where α and β are scalars, and A is a general (or general band) matrix. If m or n is less than or equal to zero or if beta is equal to one and alpha is equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. For the routine GEMV, if lda is less than one, or $trans = blas\_no\_trans$ and lda is less than m, or $trans = blas\_trans$ and lda is less than n, an error flag is set and passed to the error handler. For the C bindings of GEMV, if `order = blas_rowmajor` and if lda is less than one or lda is less than n, an error flag is set and passed to the error handler; if `order = blas_colmajor` and if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine GBMV, if kl or ku is less than zero, or if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE gbmv( a, m, kl, x, y [, trans] [, alpha] [, beta] )
  <type>(<wp>), INTENT(IN) :: a(:,:), x(:)
  INTEGER, INTENT(IN) :: m, kl
  <type>(<wp>), INTENT(INOUT) :: y(:)
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
where
  if trans = blas_no_trans then
     x has shape (n)
     y has shape (m)
  else if trans =/ blas_no_trans then
     x has shape (m)
     y has shape (n)
  end if
```

  The functionality of gemv is covered by gemm.

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGEMV( TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA,
     $                       Y, INCY )
General Band:
      SUBROUTINE BLAS_xGBMV( TRANS, M, N, KL, KU, ALPHA, A, LDA, X,
     $                       INCX, BETA, Y, INCY )
all:
      INTEGER            INCX, INCY, KL, KU, LDA, M, N, TRANS
      <type>             ALPHA, BETA
      <type>             A( LDA, * ), X( * ), Y( * )
```

- C binding:

```
General:
void BLAS_xgemv( enum blas_order_type order, enum blas_trans_type trans,
                 int m, int n, SCALAR_IN alpha, const ARRAY a, int lda,
                 const ARRAY x, int incx, SCALAR_IN beta, ARRAY y, int incy );
General Band:
void BLAS_xgbmv( enum blas_order_type order, enum blas_trans_type trans,
                 int m, int n, int kl, int ku, SCALAR_IN alpha, const ARRAY a,
                 int lda, const ARRAY x, int incx, SCALAR_IN beta,
                 ARRAY y, int incy );
```

{SY,SB,SP}MV (Symmetric matrix vector product) $y \leftarrow \alpha Ax + \beta y$ with $A = A^T$

The routines multiply a vector x by a real or complex symmetric matrix A, scales the resulting vector and adds it to the scaled vector operand y. If n or k is less than or equal to zero or if beta is equal to one and alpha is equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. For the routine SYMV, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine SBMV, if lda is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
Symmetric Band:
    SUBROUTINE sbmv( a, x, y [, uplo] [, alpha] [, beta] )
Symmetric Packed:
    SUBROUTINE spmv( ap, x, y [, uplo] [, alpha] [, beta] )
all:
      <type>(<wp>), INTENT(IN) :: <aa>, x(:)
      <type>(<wp>), INTENT(INOUT) :: y(:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
      <aa>  ::= a(:,:) or ap(:)
    and
      SB  a has shape (k+1,n)
      SP  ap has shape (n*(n+1)/2)
      x and y have shape (n)
    (k=band width)
```

The funtionality of symv is covered by symm.

- Fortran 77 binding:

```
Symmetric:
      SUBROUTINE BLAS_xSYMV( UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y,
     $                       INCY )
Symmetric Band:
      SUBROUTINE BLAS_xSBMV( UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA, Y,
     $                       INCY )
Symmetric Packed:
      SUBROUTINE BLAS_xSPMV( UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY )
all:
      INTEGER            INCX, INCY, K, LDA, N, UPLO
      <type>             ALPHA, BETA
      <type>             A( LDA, * ) or AP( * ), X( * ), Y( * )
```

- C binding:

```
Symmetric:
void BLAS_xsymv( enum blas_order_type order, enum blas_uplo_type uplo,
                 int n, SCALAR_IN alpha, const ARRAY a, int lda,
                 const ARRAY x, int incx, SCALAR_IN beta, ARRAY y, int incy );
Symmetric Band:
void BLAS_xsbmv( enum blas_order_type order, enum blas_uplo_type uplo,
                 int n, int k, SCALAR_IN alpha, const ARRAY a, int lda,
                 const ARRAY x, int incx, SCALAR_IN beta, ARRAY y, int incy );
Symmetric Packed:
void BLAS_xspmv( enum blas_order_type order, enum blas_uplo_type uplo, int n,
                 SCALAR_IN alpha, const ARRAY ap, const ARRAY x, int incx,
                 SCALAR_IN beta, ARRAY y, int incy );
```

{HE,HB,HP}MV (Hermitian matrix vector product) $y \leftarrow \alpha A x + \beta y$ with $A = A^H$

The routines multiply a vector x by a Hermitian matrix A, scales the resulting vector and adds it to the scaled vector operand y. If n is less than or equal to zero or if beta is equal to one and alpha is equal to zero, this routine returns immediately. The imaginary part of the diagonal entries of the matrix operand are supposed to be zero and should not be referenced. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. For the routine HEMV, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine HBMV, if lda is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
Hermitian Band:
    SUBROUTINE hbmv( a, x, y  [, uplo] [, alpha] [, beta] )
Hermitian Packed:
    SUBROUTINE hpmv( ap, x, y [, uplo] [, alpha] [, beta] )
all:
      COMPLEX(<wp>), INTENT(IN) :: <aa>, x(:)
      COMPLEX(<wp>), INTENT(INOUT) :: y(:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      COMPLEX(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
      <aa>  ::= a(:,:) or ap(:)
    and
       HB  a has shape (k+1,n)
       HP  ap has shape (n*(n+1)/2)
       x and y have shape (n)
    (k=band width)
```

The funtionality of hemv is covered by hemm.

- Fortran 77 binding:

```
Hermitian:
      SUBROUTINE BLAS_xHEMV( UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y,
     $                       INCY )
Hermitian Band:
      SUBROUTINE BLAS_xHBMV( UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA,
     $                       Y, INCY )
Hermitian Packed:
      SUBROUTINE BLAS_xHPMV( UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY )
all:
      INTEGER            INCX, INCY, K, LDA, N, UPLO
      <ctype>            ALPHA, BETA
      <ctype>            A( LDA, * ) or AP( * ), X( * ), Y( * )
```

- C binding:

```
Hermitian:
void BLAS_xhemv( enum blas_order_type order, enum blas_uplo_type uplo,
                 int n, CSCALAR_IN alpha, const CARRAY a, int lda,
                 const CARRAY x, int incx, CSCALAR_IN beta, CARRAY y,
                 int incy );
Hermitian Band:
void BLAS_xhbmv( enum blas_order_type order, enum blas_uplo_type uplo,
                 int n, int k, CSCALAR_IN alpha, const CARRAY a, int lda,
                 const CARRAY x, int incx, CSCALAR_IN beta, CARRAY y,
                 int incy );
Hermitian Packed:
void BLAS_xhpmv( enum blas_order_type order, enum blas_uplo_type uplo,
                 int n, CSCALAR_IN alpha, const CARRAY ap, const CARRAY x,
                 int incx, CSCALAR_IN beta, CARRAY y, int incy );
```

{TR,TB,TP}MV (Triangular matrix vector product) $x \leftarrow \alpha T x$, $x \leftarrow \alpha T^T x$ or $x \leftarrow \alpha T^H x$

The routines multiply a vector x by a general triangular matrix T or its transpose, or its conjugate transpose, and copies the resulting vector in the vector operand x. If n is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler. For the routine TRMV, if ldt is less than one or ldt is less than n, an error flag is set and passed to the error handler. For the routine TBMV, if ldt is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
Triangular Band:
    SUBROUTINE tbmv( t, x  [, uplo] [, trans] [, diag] [, alpha] )
Triangular Packed:
    SUBROUTINE tpmv( tp, x [, uplo] [, trans] [, diag] [, alpha] )
all:
```

```
      <type>(<wp>), INTENT(IN) :: <tt>
      <type>(<wp>), INTENT(INOUT) ::  x(:)
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
      TYPE (blas_diag_type), INTENT(IN), OPTIONAL :: diag
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
    where
      <tt>  ::= t(:,:) or tp(:)
    and
       TB  t has shape (k+1,n)
       TP  tp has shape (n*(n+1)/2)
       x has shape (n)
    (k=band width)
```

The funtionality of trmv is covered by trmm.

- Fortran 77 binding:

```
Triangular:
       SUBROUTINE BLAS_xTRMV( UPLO, TRANS, DIAG, N, ALPHA, T, LDT, X,
      $                       INCX )
Triangular Band:
       SUBROUTINE BLAS_xTBMV( UPLO, TRANS, DIAG, N, K, ALPHA, T, LDT,
      $                       X, INCX )
Triangular Packed:
       SUBROUTINE BLAS_xTPMV( UPLO, TRANS, DIAG, N, ALPHA, TP, X, INCX )
all:
      INTEGER           DIAG, INCX, K, LDT, N, TRANS, UPLO
      <type>            ALPHA
      <type>            T( LDT, * ) or TP( * ), X( * )
```

- C binding:

```
Triangular:
void BLAS_xtrmv( enum blas_order_type order, enum blas_uplo_type uplo,
                 enum blas_trans_type trans, enum blas_diag_type diag, int n,
                 SCALAR_IN alpha, const ARRAY t, int ldt, ARRAY x, int incx );
Triangular Band:
void BLAS_xtbmv( enum blas_order_type order, enum blas_uplo_type uplo,
                 enum blas_trans_type trans, enum blas_diag_type diag, int n,
                 int k, SCALAR_IN alpha, const ARRAY t, int ldt, ARRAY x,
                 int incx );
Triangular Packed:
void BLAS_xtpmv( enum blas_order_type order, enum blas_uplo_type uplo,
                 enum blas_trans_type trans, enum blas_diag_type diag, int n,
                 SCALAR_IN alpha, const ARRAY tp, ARRAY x, int incx );
```

GE_SUM_MV (Summed matrix vector multiplies) $y \leftarrow \alpha Ax + \beta Bx$

This routine adds the product of two scaled matrix vector products. It can be used to compute the residual of an approximate eigenvector and eigenvalue of the generalized eigenvalue problem $A * x = \lambda * B * x$. If m or n is less than or equal to zero, then this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or lda is less than m, or ldb is less than one or ldb is less than m, an error flag is set and passed to the error handler. For the C bindings for GE_SUM_MV, if `order = blas_rowmajor` and if lda is less than one or lda is less than n, or if ldb is less than one or ldb is less than n, an error flag is set and passed to the error handler; if `order = blas_colmajor` and if lda is less than one or lda is less than m, or if ldb is less than one or ldb is less than m, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE ge_sum_mv( a, x, b, y [, alpha] [, beta] )
  <type>(<wp>), INTENT (IN) :: a(:,:), b(:,:)
  <type>(<wp>), INTENT (IN) :: x(:)
  <type>(<wp>), INTENT (OUT) :: y(:)
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
where
  x has shape (n);
  y has shape (m);
  a and b have shape (m,n) for general matrices
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xGE_SUM_MV( M, N, ALPHA, A, LDA, X, INCX, BETA,
     $                            B, LDB, Y, INCY )
      INTEGER            INCX, INCY, LDA, LDB, M, N
      <type>             ALPHA, BETA
      <type>             A( LDA, * ), B( LDB, * ), X( * ), Y( * )
```

- C binding:

```
void BLAS_xge_sum_mv( enum blas_order_type order, int m, int n,
                      SCALAR_IN alpha, const ARRAY a, int lda,
                      const ARRAY x, int incx, SCALAR_IN beta,
                      const ARRAY B, int ldb, ARRAY y, int incy );
```

GEMVT (Multiple matrix vector multiplies) $x \leftarrow \beta A^T y + z, w \leftarrow \alpha Ax$

The routine combines a matrix vector and a transposed matrix vector multiply. It multiplies a vector y by a general matrix A^T, scales the resulting vector and adds the result to z, storing the result in the vector operand x. It then multiplies the matrix A by x, scales the resulting vector and stores it in the vector operand w.

Advice to implementors. Note that x and w may be computed while passing A through the top of the memory just once. This optimization, which accelerates algorithms like reducing a symmetric matrix to tridiagonal form, is the justification for GEMVT, which could otherwise be implemented by two calls to GEMV. (*End of advice to implementors.*)

If m or n is less than or equal to zero, this function returns immediately. As described in section 2.5.3, the value incx or incy or incw or incz less than zero is permitted. However, if either incx, incy, incw, or incz is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the C bindings, if `order = blas_rowmajor` and if lda is less than one or lda is less than n, an error flag is set and passed to the error handler; if `order = blas_colmajor` and if lda is less than one or lda is less than m, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE gemvt( a, x, y, w, z [, alpha] [, beta] )
  <type>(<wp>), INTENT (IN) :: a(:,:)
  <type>(<wp>), INTENT (IN) :: y(:), z(:)
  <type>(<wp>), INTENT (OUT) :: x(:), w(:)
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
where
  w and y have shape (m);
  x and z have shape (n);
  a has shape (m,n) for general matrix
```

- Fortran 77 binding:

```
 SUBROUTINE BLAS_xGEMVT( M, N, ALPHA, A, LDA, X, INCX, Y, INCY,
$                        BETA, W, INCW, Z, INCZ )
 INTEGER            INCW, INCX, INCY, INCZ, LDA, M, N
 <type>             ALPHA, BETA
 <type>             A( LDA, * ), X( * ), Y( * ), W( * ), Z( * )
```

- C binding:

```
void BLAS_xgemvt( enum blas_order_type order, int m, int n, SCALAR_IN alpha,
                  const ARRAY a, int lda, ARRAY x, int incx, const ARRAY y,
                  int incy, SCALAR_IN beta, ARRAY w, int incw, const ARRAY z,
                  int incz );
```

TRMVT (Multiple triangular matrix vector product) $x \leftarrow T^T y$ and $w \leftarrow Tz$

The routine combines a matrix vector and a transposed matrix vector multiply. It multiplies a vector y by a triangular matrix T^T, storing the result as x. It also multiplies the matrix by the vector z, storing the result as w.

Advice to implementors. Note that x and w may be computed while passing T through the top of the memory just once. This optimization, which accelerates algorithms like reducing a symmetric matrix to tridiagonal form, is the justification for TRMVT, which could otherwise be implemented by two calls to TRMV. (*End of advice to implementors.*)

If n is less than or equal to zero, this function returns immediately. As described in section 2.5.3, the value incx or incy or incw or incz less than zero is permitted. However, if either incx, incy, incw, or incz is equal to zero, an error flag is set and passed to the error handler. If ldt is less than one or ldt is less than n, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE trmvt( t, x, y, w, z [, uplo] )
  <type>(<wp>), INTENT (IN) :: t(:,:)
  <type>(<wp>), INTENT (IN) :: y(:), z(:)
  <type>(<wp>), INTENT (OUT) :: x(:), w(:)
  TYPE (blas_uplo_type), INTENT (IN), OPTIONAL :: uplo
where
  w, x, y and z have shape (n);
  t has shape (n,n).
```

- Fortran 77 binding:

```
 SUBROUTINE BLAS_xTRMVT( UPLO, N, T, LDT, X, INCX, Y, INCY, W, INCW,
$                        Z, INCZ )
 INTEGER           INCW, INCX, INCY, INCZ, LDT, N, UPLO
 <type>            T( LDT, * ), W( * ), X( * ), Y( * ), Z( * )
```

- C binding:

```
void BLAS_xtrmvt( enum blas_order_type order, enum blas_uplo_type uplo,
                  int n, const ARRAY t, int ldt, ARRAY x, int incx,
                  const ARRAY y, int incy, ARRAY w, int incw, const ARRAY z,
                  int incz );
```

GEMVER (Multiple matrix vector multiply with a rank 2 update)

$$\hat{A} \leftarrow A + u_1 v_1^T + u_2 v_2^T \text{ and } x \leftarrow \beta \hat{A}^T y + z \text{ and } w \leftarrow \alpha \hat{A} x$$

The routine precedes a combined matrix vector and a transposed matrix vector multiply by a rank two update. A matrix A is updated by $u_1 v_1^T$ and $u_2 v_2^T$. The transpose of the updated matrix is multiplied by a vector y. The resulting vector is scaled and added to the vector operand z, and stored in x . The operand x is multiplied by the updated matrix A. The resulting vector is scaled and stored as w.

> *Advice to implementors.* Note that $\hat{A}$, x and w may be computed while passing A through the top of the memory just once. This optimization, which accelerates algorithms like reducing a general matrix to bidiagonal form, is the justification for GEMVER, which could otherwise be implemented by calls to other BLAS routines. (*End of advice to implementors.*)

If m or n is less than or equal to zero, this function returns immediately. As described in section 2.5.3, the value incx or incy or incw or incz less than zero is permitted. However, if either incx, incy, incw, or incz is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the C bindings, if `order = blas_rowmajor` and if lda is less than one or lda is less than n, an error flag is set and passed to the error handler; if `order = blas_colmajor` and if lda is less than one or lda is less than m, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE gemver( a, u1, v1, u2, v2, x, y, z, w [, alpha] [, beta] )
  <type>(<wp>), INTENT (IN) :: u1(:), u2(:), v1(:), v2(:), y(:), z(:)
  <type>(<wp>), INTENT (INOUT) :: a(:,:), x(:)
  <type>(<wp>), INTENT (OUT) :: w(:)
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
where
  u1, u2, w and y have shape (m);
  v1, v2, x and z have shape (n);
  a has shape (m,n).
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGEMVER( M, N, A, LDA, U1, V1, U2, V2, ALPHA, X,
     $                         INCX, Y, INCY, BETA, W, INCW, Z, INCZ )
      INTEGER            INCW, INCX, INCY, INCZ, LDA, M, N
      <type>             ALPHA, BETA
      <type>             U1( * ), V1( * ), U2( * ), V2( * )
      <type>             A( LDA, * ), W( * ), X( * ), Y( * ), Z( * )
```

- C binding:

```
General:
void BLAS_xgemver( enum blas_order_type order, int m, int n, ARRAY a,
                   int lda, const ARRAY u1, const ARRAY v1,
                   const ARRAY u2, const ARRAY v2, SCALAR_IN alpha,
                   ARRAY x, int incx, const ARRAY y, int incy, ARRAY w,
                   int incw, SCALAR_IN beta, const ARRAY z, int incz );
```

{TR,TB,TP}SV (Triangular solve) $x \leftarrow \alpha T^{-1}x$, $x \leftarrow \alpha T^{-T}x$

These routines solve one of the systems of equations $x \leftarrow \alpha T^{-1}x$ or $x \leftarrow \alpha T^{-T}x$, where x is a vector and the matrix T is a unit, non-unit, upper or lower triangular (or triangular banded or triangular packed) matrix. If n is less than or equal to zero, this function returns immediately. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler. For TRSV, if ldt is less than one or ldt is less than n, an error flag is set and passed to the error handler. For TBSV, if ldt is less than one or ldt is less than k plus one, an error flag is set and passed to the error handler.

Advice to implementors. Note that no check for singularity, or near singularity is specified for these triangular equation-solving routines. The requirements for such a test depend on the application, and so we felt that this should not be included, but should instead be performed before calling the triangular solver. (*End of advice to implementors.*)

- Fortran 95 binding:

```
Triangular Band:
    SUBROUTINE tbsv( t, x [, uplo] [, trans] [, diag] [, alpha] )
Triangular Packed:
    SUBROUTINE tpsv( tp, x [, uplo] [, trans] [, diag] [, alpha] )
all:
      <type>(<wp>), INTENT(IN) :: <tt>
      <type>(<wp>), INTENT(INOUT) :: x(:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
      TYPE (blas_diag_type), INTENT(IN), OPTIONAL :: diag
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
    where
      <tt>  ::= t(:,:) or tp(:)
    and
      x has shape (n)
      TB  t has shape (k+1,n)
      TP  tp has shape (n*(n+1)/2)
    (k=band width)
```

The funtionality of trsv is covered by trsm.

- Fortran 77 binding:

```
Triangular:
       SUBROUTINE BLAS_xTRSV( UPLO, TRANS, DIAG, N, ALPHA, T, LDT, X,
      $                       INCX )
Triangular Band:
       SUBROUTINE BLAS_xTBSV( UPLO, TRANS, DIAG, N, K, ALPHA, T, LDT,
      $                       X, INCX )
Triangular Packed:
       SUBROUTINE BLAS_xTPSV( UPLO, TRANS, DIAG, N, ALPHA, TP, X, INCX )
all:
      INTEGER            DIAG, INCX, K, LDT, N, TRANS, UPLO
      <type>             ALPHA
      <type>             T( LDT, * ) or TP( * ), X( * )
```

- C binding:

```
Triangular:
void BLAS_xtrsv( enum blas_order_type order, enum blas_uplo_type uplo,
                 enum blas_trans_type trans, enum blas_diag_type diag,
                 int n, SCALAR_IN alpha, const ARRAY t, int ldt,
```

```
                  ARRAY x, int incx );
Triangular Band:
void BLAS_xtbsv( enum blas_order_type order, enum blas_uplo_type uplo,
                 enum blas_trans_type trans, enum blas_diag_type diag,
                 int n, int k, SCALAR_IN alpha, const ARRAY t, int ldt,
                 ARRAY x, int incx );
Triangular Packed:
void BLAS_xtpsv( enum blas_order_type order, enum blas_uplo_type uplo,
                 enum blas_trans_type trans, enum blas_diag_type diag,
                 int n, SCALAR_IN alpha, const ARRAY tp, ARRAY x,
                 int incx );
```

GER (Rank one update) $A \in \mathbb{R}^{n^2}, A \leftarrow \alpha x y^T + \beta A$ $A \in \mathbb{C}^{n^2}, A \leftarrow \alpha x y^T + \beta A$ or $A \leftarrow \alpha x y^H + \beta A$

This routine performs the rank 1 operation $A \leftarrow \alpha x y^T + \beta A$ where α and β are scalars, x and y are vectors, and and A is a matrix. If m or n is less than or equal to zero or if beta is equal to one and alpha is equal to zero, this function returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the C bindings, if `order = blas_rowmajor` and if lda is less than one or lda is less than n, an error flag is set and passed to the error handler; if `order = blas_colmajor` and if lda is less than one or lda is less than m, an error flag is set and passed to the error handler.

The operator argument conj is only referenced when x and y are complex vectors. When x and y are complex vectors, the vector components y_i are used unconjugated or conjugated as specified by the operator argument conj.

- Fortran 95 binding: *Refer to GEMM specification*
- Fortran 77 binding:

```
      SUBROUTINE BLAS_xGER( CONJ, M, N, ALPHA, X, INCX, Y, INCY, BETA,
     $                      A, LDA )
      INTEGER            CONJ, INCX, INCY, LDA, M, N
      <type>             ALPHA, BETA
      <type>             A( LDA, * ), X( * ), Y( * )
```

- C binding:

```
void BLAS_xger( enum blas_order_type order, enum blas_conj_type conj,
                int m, int n, SCALAR_IN alpha, const ARRAY x, int incx,
                const ARRAY y, int incy, SCALAR_IN beta, ARRAY a, int lda );
```

{SY,SP}R (Symmetric Rank One Update) $A \leftarrow \alpha x x^T + \beta A$ with $A = A^T$

The routine performs the symmetric rank-1 update $A = \alpha x x^T + \beta A$, where α and β are scalars, x is a vector and A is a symmetric (symmetric packed) matrix. This routine returns immediately if n is less than or equal to zero or if beta is equal to one and alpha is equal to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or lda is less than n, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routines xSYR and xSPR with added functionality for complex symmetric matrices.

- Fortran 95 binding:

```
Symmetric Packed:
    SUBROUTINE spr( x, ap [, uplo] [, trans] [, alpha] [, beta] )
      <type>(<wp>), INTENT(IN) :: x(:)
      <type>(<wp>), INTENT(INOUT) :: ap(:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
     x has shape (n)
     ap has shape (n*(n+1)/2)
```

  The functionality of syr is covered by syrk.

- Fortran 77 binding:

```
Symmetric:
      SUBROUTINE BLAS_xSYR( UPLO, N, ALPHA, X, INCX, BETA, A, LDA )
Symmetric Packed:
      SUBROUTINE BLAS_xSPR( UPLO, N, ALPHA, X, INCX, BETA, AP )
all:
      INTEGER             INCX, LDA, N, UPLO
      <type>              ALPHA, BETA
      <type>              A( LDA, * ) or AP( * ), X( * )
```

- C binding:

```
Symmetric:
void BLAS_xsyr( enum blas_order_type order, enum blas_uplo_type uplo,
                int n, SCALAR_IN alpha, const ARRAY x, int incx,
                SCALAR_IN beta, ARRAY a, int lda );
Symmetric Packed:
void BLAS_xspr( enum blas_order_type order, enum blas_uplo_type uplo,
                int n, SCALAR_IN alpha, const ARRAY x, int incx,
                SCALAR_IN beta, ARRAY ap );
```

{HE,HP}R (Hermitian Rank One Update) $A \leftarrow \alpha x x^H + \beta A$ with $A = A^H$

The routine performs the Hermitian rank-1 update $A = \alpha x x^H + \beta A$, where α and β are real scalars, x is a complex vector and A is a Hermitian (Hermitian packed) matrix. This routine returns immediately if n is less than or equal to zero or if beta is equal to one and alpha is equal to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or lda is less than n, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
Hermitian Packed:
    SUBROUTINE hpr( x, ap [, uplo] [, trans] [, alpha] [, beta] )
      COMPLEX(<wp>), INTENT(IN) :: x(:)
      COMPLEX(<wp>), INTENT(INOUT) :: ap(:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
      REAL(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
     x has shape (n)
     ap has shape (n*(n+1)/2)
```

  The functionality of her is covered by herk.

- Fortran 77 binding:

```
Hermitian:
      SUBROUTINE BLAS_xHER( UPLO, N, ALPHA, X, INCX, BETA, A, LDA )
Hermitian Packed:
      SUBROUTINE BLAS_xHPR( UPLO, N, ALPHA, X, INCX, BETA, AP )
all:
      INTEGER            INCX, LDA, N, UPLO
      <rtype>            ALPHA, BETA
      <ctype>            A( LDA, * ) or AP( * ), X( * )
```

- C binding:

```
Hermitian:
void BLAS_xher( enum blas_order_type order, enum blas_uplo_type uplo,
                int n, RSCALAR_IN alpha, const CARRAY x, int incx,
                RSCALAR_IN beta, CARRAY a, int lda );
Hermitian Packed:
void BLAS_xhpr( enum blas_order_type order, enum blas_uplo_type uplo,
                int n, RSCALAR_IN alpha, const CARRAY x, int incx,
                RSCALAR_IN beta, CARRAY ap );
```

{SY,SP}R2 (Symmetric Rank two update) $A \leftarrow \alpha x y^T + \alpha y x^T + \beta A$ with $A = A^T$

The routine performs the symmetric rank-2 update $A = \alpha x y^T + \alpha y x^T + \beta A$, where α and β are scalars, x is a vector and A is a symmetric (symmetric packed) matrix. This routine returns immediately if n is less than or equal to zero or if beta is equal to one and alpha is equal to zero. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or lda is less than n, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routines xSYR2 and xSPR2 with added functionality for complex symmetric matrices.

- Fortran 95 binding:

```
Symmetric Packed:
    SUBROUTINE spr2( x, y, ap [, uplo] [, trans] [, alpha] [, beta] )
      <type>(<wp>), INTENT(IN) :: x(:), y(:)
      <type>(<wp>), INTENT(INOUT) :: ap(:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
     x and y have shape (n)
     ap has shape (n*(n+1)/2)
```

The functionality of syr2 is covered by syr2k.

- Fortran 77 binding:

```
Symmetric:
      SUBROUTINE BLAS_xSYR2( UPLO, N, ALPHA, X, INCX, Y, INCY, BETA, A,
     $                       LDA )
Symmetric Packed:
      SUBROUTINE BLAS_xSPR2( UPLO, N, ALPHA, X, INCX, Y, INCY, BETA,
     $                       AP )
all:
      INTEGER            INCX, LDA, N, UPLO
      <type>             ALPHA, BETA
      <type>             A( LDA, * ) or AP( * ), X( * ), Y( * )
```

- C binding:

```
Symmetric:
void BLAS_xsyr2( enum blas_order_type order, enum blas_uplo_type uplo,
                 int n, SCALAR_IN alpha, const ARRAY x, int incx,
                 const ARRAY y, int incy, SCALAR_IN beta, ARRAY a, int lda );
Symmetric Packed:
void BLAS_xspr2( enum blas_order_type order, enum blas_uplo_type uplo,
                 int n, SCALAR_IN alpha, const ARRAY x, int incx,
                 const ARRAY y, int incy, SCALAR_IN beta, ARRAY ap );
```

{HE,HP}R2 (Hermitian Rank two update) $A \leftarrow \alpha x y^H + \bar{\alpha} y x^H + \beta A$ with $A = A^H$

The routine performs the Hermitian rank-2 update $A = \alpha x y^H + \bar{\alpha} y x^H + \beta A$, where α is a complex scalar and and β is a real scalar, x and y are complex vectors and A is a Hermitian

(Hermitian packed) matrix. This routine returns immediately if n is less than or equal to zero or if beta is equal to one and alpha is equal to zero. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or lda is less than n, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
Hermitian Packed:
    SUBROUTINE hpr2( x, y, ap [, uplo] [, trans] [, alpha] [, beta] )
      COMPLEX(<wp>), INTENT(IN) :: x(:), y(:)
      COMPLEX(<wp>), INTENT(INOUT) :: ap(:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
      COMPLEX(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
     x and y have shape (n)
     ap has shape (n*(n+1)/2)
```

  The functionality of her2 is covered by her2k.

- Fortran 77 binding:

```
Hermitian:
      SUBROUTINE BLAS_xHER2( UPLO, N, ALPHA, X, INCX, Y, INCY, BETA, A,
     $                    LDA )
Hermitian Packed:
      SUBROUTINE BLAS_xHPR2( UPLO, N, ALPHA, X, INCX, Y, INCY, BETA,
     $                       AP )
all:
      INTEGER           INCX, LDA, N, UPLO
      <ctype>           ALPHA
      <rtype>           BETA
      <ctype>           A( LDA, * ) or AP( * ), X( * ), Y( * )
```

- C binding:

```
Hermitian:
void BLAS_xher2( enum blas_order_type order, enum blas_uplo_type uplo,
                 int n, CSCALAR_IN alpha, const CARRAY x, int incx,
                 const CARRAY y, int incy, RSCALAR_IN beta, CARRAY a,
                 int lda );
Hermitian Packed:
void BLAS_xhpr2( enum blas_order_type order, enum blas_uplo_type uplo,
                 int n, CSCALAR_IN alpha, const CARRAY x, int incx,
                 const CARRAY y, int incy, RSCALAR_IN beta, CARRAY ap );
```

2.8.7 Matrix Operations

{GE,GB,SY,HE,SB,HB,SP,HP,TR,TB,TP}_NORM (Matrix norms)

$$r \leftarrow ||A||_1,\ ||A||_{1R},\ ||A||_F,\ ||A||_\infty,\ ||A||_{\infty R},\ ||A||_{max},\ or\ ||A||_{maxR}$$

These routines compute the one-norm, real one-norm, Frobenius-norm, infinity-norm, real infinity-norm, max-norm, or real max-norm of a general matrix A depending on the value passed as the norm operator argument. This routine returns immediately with the output scalar r set to zero if m (for nonsymmetric matrices) or n or kl or ku (for band matrices) or k (for symmetric band matrices) is less than or equal to zero. The resulting scalar r is always real and as defined in section 2.1.3. If `norm = blas_two_norm`, requesting the two-norm of a matrix, an error flag is set and passed to the error handler. The only exception to this rule is if the matrix is a single column or a single row, whereby the Frobenius-norm is returned since the two-norm and Frobenius-norm of a vector are identical. For the routine GE_NORM, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the C bindings of GE_NORM, if `order = blas_rowmajor` and if lda is less than one or lda is less than n, an error flag is set and passed to the error handler; if `order = blas_colmajor` and if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine GB_NORM, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler. For the routines SY_NORM, HE_NORM, and TR_NORM, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routines SB_NORM, HB_NORM, and TB_NORM, if lda is less than k plus one, an error flag is set and passed to the error handler.

> *Advice to implementors.* High-quality implementations of these routines should be accurate. The subroutines SLANGB, SLANGE, SLANGT, SLANHS, SLANSB, SLANSP, SLANST, SLANSY, SLANTB, SLANTP, and SLANTR, of the LAPACK [6] software library are examples of accurate implementations. High-quality implementations should document the accuracy of the algorithms used in this routine so as to alleviate the portability problems this represents. (*End of advice to implementors.*)

- Fortran 95 binding:

```
General:
    REAL(<wp>) FUNCTION ge_norm( a [, norm] )
General Band:
    REAL(<wp>) FUNCTION gb_norm( a, m, kl [, norm] )
Symmetric:
    REAL(<wp>) FUNCTION sy_norm( a [, norm] [, uplo] )
Hermitian:
    REAL(<wp>) FUNCTION he_norm( a [, norm] [, uplo] )
Symmetric Band:
    REAL(<wp>) FUNCTION sb_norm( a [, norm] [, uplo] )
Hermitian Band:
    REAL(<wp>) FUNCTION hb_norm( a [, norm] [, uplo] )
Symmetric Packed:
    REAL(<wp>) FUNCTION sp_norm( ap [, norm] [, uplo] )
Hermitian Packed:
    REAL(<wp>) FUNCTION hp_norm( ap [, norm] [, uplo] )
```

```
Triangular:
    REAL(<wp>) FUNCTION tr_norm( a [, norm] [, uplo] [, diag] )
Triangular Band:
    REAL(<wp>) FUNCTION tb_norm( a [, norm] [, uplo] [, diag] )
Triangular Packed:
    REAL(<wp>) FUNCTION tp_norm( ap [, norm] [, uplo] [, diag] )
all:
      <type>(<wp>), INTENT (IN) :: a(:,:) | ap(:)
      INTEGER, INTENT (IN) :: m, kl
      TYPE (blas_norm_type), INTENT (IN), OPTIONAL :: norm
      TYPE (blas_uplo_type), INTENT (IN), OPTIONAL :: uplo
      TYPE (blas_diag_type), INTENT (IN), OPTIONAL :: diag
    where
      a has shape (m,n) for general matrix
                  (l,n) for general banded matrix (l > kl)
                  (n,n) for symmetric, Hermitian or triangular
                  (k+1,n) for symmetric banded, Hermitian banded
                          or triangular banded (k=band width)
      ap has shape (n*(n+1)/2).
```

- Fortran 77 binding:

```
General:
      <rtype> FUNCTION BLAS_xGE_NORM( NORM, M, N, A, LDA )
General Band:
      <rtype> FUNCTION BLAS_xGB_NORM( NORM, M, N, KL, KU, A, LDA )
Symmetric:
      <rtype> FUNCTION BLAS_xSY_NORM( NORM, UPLO, N, A, LDA )
Hermitian:
      <rtype> FUNCTION BLAS_xHE_NORM( NORM, UPLO, N, A, LDA )
Symmetric Band:
      <rtype> FUNCTION BLAS_xSB_NORM( NORM, UPLO, N, K, A, LDA )
Hermitian Band:
      <rtype> FUNCTION BLAS_xHB_NORM( NORM, UPLO, N, K, A, LDA )
Symmetric Packed:
      <rtype> FUNCTION BLAS_xSP_NORM( NORM, UPLO, N, AP )
Hermitian Packed:
      <rtype> FUNCTION BLAS_xHP_NORM( NORM, UPLO, N, AP )
Triangular:
      <rtype> FUNCTION BLAS_xTR_NORM( NORM, UPLO, DIAG, N, A, LDA )
Triangular Band:
      <rtype> FUNCTION BLAS_xTB_NORM( NORM, UPLO, DIAG, N, K, A, LDA )
Triangular Packed:
      <rtype> FUNCTION BLAS_xTP_NORM( NORM, UPLO, DIAG, N, AP )
all:
      INTEGER            DIAG, K, KL, KU, LDA, M, N, NORM, UPLO
      <type>             A( LDA, * ) or AP( * )
```

- C binding:

```
General:
void BLAS_xge_norm( enum blas_order_type order, enum blas_norm_type norm,
                    int m, int n, const ARRAY a, int lda, RSCALAR_INOUT r );
General Band:
void BLAS_xgb_norm( enum blas_order_type order, enum blas_norm_type norm,
                    int m, int n, int kl, int ku, const ARRAY a, int lda,
                    RSCALAR_INOUT r );
Symmetric:
void BLAS_xsy_norm( enum blas_order_type order, enum blas_norm_type norm,
                    enum blas_uplo_type uplo, int n, const ARRAY a,
                    int lda, RSCALAR_INOUT r );
Hermitian:
void BLAS_xhe_norm( enum blas_order_type order, enum blas_norm_type norm,
                    enum blas_uplo_type uplo, int n, const CARRAY a,
                    int lda, RSCALAR_INOUT r );
Symmetric Band:
void BLAS_xsb_norm( enum blas_order_type order, enum blas_norm_type norm,
                    enum blas_uplo_type uplo, int n, int k, const ARRAY a,
                    int lda, RSCALAR_INOUT r );
Hermitian Band:
void BLAS_xhb_norm( enum blas_order_type order, enum blas_norm_type norm,
                    enum blas_uplo_type uplo, int n, int k, const CARRAY a,
                    int lda, RSCALAR_INOUT r );
Symmetric Packed:
void BLAS_xsp_norm( enum blas_order_type order, enum blas_norm_type norm,
                    enum blas_uplo_type uplo, int n, const ARRAY ap,
                    RSCALAR_INOUT r );
Hermitian Packed:
void BLAS_xhp_norm( enum blas_order_type order, enum blas_norm_type norm,
                    enum blas_uplo_type uplo, int n, const CARRAY ap,
                    RSCALAR_INOUT r );
Triangular:
void BLAS_xtr_norm( enum blas_order_type order, enum blas_norm_type norm,
                    enum blas_uplo_type uplo, enum blas_diag_type diag,
                    int n, const ARRAY a, int lda, RSCALAR_INOUT r );
Triangular Band:
void BLAS_xtb_norm( enum blas_order_type order, enum blas_norm_type norm,
                    enum blas_uplo_type uplo, enum blas_diag_type diag,
                    int n, int k, const ARRAY a, int lda, RSCALAR_INOUT r );
Triangular Packed:
void BLAS_xtp_norm( enum blas_order_type order, enum blas_norm_type norm,
                    enum blas_uplo_type uplo, enum blas_diag_type diag,
                    int n, const ARRAY ap, RSCALAR_INOUT r );
```

{GE,GB}_DIAG_SCALE (Diagonal scaling) $A \leftarrow DA, AD$ with D diagonal

These routines scale a general (or banded) matrix A on the left side or the right side by a diagonal matrix D. This routine returns immediately if m or n or kl or ku (for band matrices) is less than or equal to zero. As described in section 2.5.3, the value incd less than zero is permitted. However, if incd is equal to zero, an error flag is set and passed to the error handler. For the routine GE_DIAG_SCALE, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the C bindings of GE_DIAG_SCALE, if `order = blas_rowmajor` and if lda is less than one or lda is less than n, an error flag is set and passed to the error handler; if `order = blas_colmajor` and if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine GB_DIAG_SCALE, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
    SUBROUTINE ge_diag_scale( d, a [, side] )
General Band:
    SUBROUTINE gb_diag_scale( d, a, m, kl [, side] )
all:
      <type>(<wp>), INTENT (IN) :: d(:)
      <type>(<wp>), INTENT (INOUT) :: a(:,:)
      INTEGER, INTENT (IN) :: m, kl
      TYPE (blas_side_type), INTENT (IN), OPTIONAL :: side
    where
      a has shape (m,n) for general matrix
                  (l,n) for general banded matrix (l > kl)
      d has shape (p) where p = m if side = blas_left_side
                            p = n if side = blas_right_side
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_DIAG_SCALE( SIDE, M, N, D, INCD, A, LDA )
General Band:
      SUBROUTINE BLAS_xGB_DIAG_SCALE( SIDE, M, N, KL, KU, D, INCD, A,
     $                                LDA )
all:
      INTEGER            INCD, KL, KU, LDA, M, N, SIDE
      <type>             A( LDA, * ), D( * )
```

- C binding:

```
General:
void BLAS_xge_diag_scale( enum blas_order_type order,
                          enum blas_side_type side, int m, int n,
                          const ARRAY d, int incd, ARRAY a, int lda );
General Band:
```

```
void BLAS_xgb_diag_scale( enum blas_order_type order,
                          enum blas_side_type side, int m, int n, int kl,
                          int ku, const ARRAY d, int incd, ARRAY a, int lda );
```

{GE,GB}_LRSCALE (Two-sided diagonal scaling) $A \leftarrow D_L A D_R$

These routines scale a general (or banded) matrix A on the left side by a diagonal matrix D_L and on the right side by a diagonal matrix D_R. This routine returns immediately if m or n or kl or ku (for band matrices) is less than or equal to zero. As described in section 2.5.3, the value incdl or incdr less than zero is permitted. However, if either incdl or incdr is equal to zero, an error flag is set and passed to the error handler. For the routine GE_LRSCALE, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the C bindings of GE_LRSCALE, if `order = blas_rowmajor` and if lda is less than one or lda is less than n, an error flag is set and passed to the error handler; if `order = blas_colmajor` and if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine GB_LRSCALE, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
    SUBROUTINE ge_lrscale( dl, dr, a )
General Band:
    SUBROUTINE gb_lrscale( dl, dr, a, m, kl )
all:
      <type>(<wp>), INTENT (IN) :: dl(:), dr(:)
      <type>(<wp>), INTENT (INOUT) :: a(:,:)
      INTEGER, INTENT (IN) :: m, kl
    where
      a has shape (m,n) for general matrix
                  (l,n) for general banded matrix (l > kl)
      dl has shape (m)
      dr has shape (n)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_LRSCALE( M, N, DL, INCDL, DR, INCDR, A, LDA )
General Band:
      SUBROUTINE BLAS_xGB_LRSCALE( M, N, KL, KU, DL, INCDL, DR, INCDR,
     $                             A, LDA )
all:
      INTEGER            INCDL, INCDR, KL, KU, LDA, M, N
      <type>             A( LDA, * ), DL( * ), DR( * )
```

- C binding:

```
General:
void BLAS_xge_lrscale( enum blas_order_type order, int m, int n,
                       const ARRAY dl, int incdl, const ARRAY dr,
                       int incdr, ARRAY a, int lda );
General Band:
void BLAS_xgb_lrscale( enum blas_order_type order, int m, int n, int kl,
                       int ku, const ARRAY dl, int incdl, const ARRAY dr,
                       int incdr, ARRAY a, int lda );
```

{SY,SB,SP}_LRSCALE (Two-sided diagonal scaling of a symmetric matrix)

$$A \leftarrow DAD \text{ with } A = A^T$$

These routines perform a two-sided scaling of a symmetric (or symmetric banded or symmetric packed) matrix A by a diagonal matrix D. This routine returns immediately if n or k (for symmetric band matrices) is less than or equal to zero. As described in section 2.5.3, the value incd less than zero is permitted. However, if incd is equal to zero, an error flag is set and passed to the error handler. For the routines SY_LRSCALE and SP_LRSCALE, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine SB_LRSCALE, if lda is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
Symmetric:
    SUBROUTINE sy_lrscale( d, a [, uplo] )
Symmetric Band:
    SUBROUTINE sb_lrscale( d, a [, uplo] )
Symmetric Packed:
    SUBROUTINE sp_lrscale( d, ap [, uplo] )
all:
      <type>(<wp>), INTENT (IN) :: d(:)
      <type>(<wp>), INTENT (INOUT) :: a(:,:) | ap(:)
      TYPE (blas_uplo_type), INTENT (IN), OPTIONAL :: uplo
    where
      a has shape (n,n) for symmetric
                  (k+1,n) for symmetric banded (k=band width)
      ap has shape (n*(n+1)/2).
      d has shape (n)
```

- Fortran 77 binding:

```
Symmetric:
      SUBROUTINE BLAS_xSY_LRSCALE( UPLO, N, D, INCD, A, LDA )
Symmetric Band:
      SUBROUTINE BLAS_xSB_LRSCALE( UPLO, N, K, D, INCD, A, LDA )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_LRSCALE( UPLO, N, D, INCD, AP )
```

```
all:
     INTEGER           INCD, K, LDA, N, UPLO
     <type>            A( LDA, * ) or AP( * ), D( * )
```

- C binding:

```
Symmetric:
void BLAS_xsy_lrscale( enum blas_order_type order, enum blas_uplo_type uplo,
                       int n, const ARRAY d, int incd, ARRAY a, int lda );
Symmetric Band:
void BLAS_xsb_lrscale( enum blas_order_type order, enum blas_uplo_type uplo,
                       int n, int k, const ARRAY d, int incd, ARRAY a,
                       int lda );
Symmetric Packed:
void BLAS_xsp_lrscale( enum blas_order_type order, enum blas_uplo_type uplo,
                       int n, const ARRAY d, int incd, ARRAY ap );
```

{HE,HB,HP}_LRSCALE (Two-sided diagonal scaling of a Hermitian matrix)

$$A \leftarrow DAD^H \text{ with } A = A^H$$

These routines perform a two-sided scaling of a Hermitian (or Hermitian banded or Hermitian packed) matrix A by a diagonal matrix D. This routine returns immediately if n or k (for Hermitian band matrices) is less than or equal to zero. As described in section 2.5.3, the value incd less than zero is permitted. However, if incd is equal to zero, an error flag is set and passed to the error handler. For the routines HE_LRSCALE, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine HB_LRSCALE, if lda is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
Hermitian:
    SUBROUTINE he_lrscale( d, a [, uplo] )
Hermitian Band:
    SUBROUTINE hb_lrscale( d, a [, uplo] )
Hermitian Packed:
    SUBROUTINE hp_lrscale( d, ap [, uplo] )
all:
      COMPLEX(<wp>), INTENT (IN) :: d(:)
      COMPLEX(<wp>), INTENT (INOUT) :: a(:,:) | ap(:)
      TYPE (blas_uplo_type), INTENT (IN), OPTIONAL :: uplo
    where
      a has shape (n,n) for Hermitian
                  (k+1,n) for Hermitian banded (k=band width)
      ap has shape (n*(n+1)/2).
      d has shape (n)
```

- Fortran 77 binding:

```
Hermitian:
      SUBROUTINE BLAS_xHE_LRSCALE( UPLO, N, D, INCD, A, LDA )
Hermitian Band:
      SUBROUTINE BLAS_xHB_LRSCALE( UPLO, N, K, D, INCD, A, LDA )
Hermitian Packed:
      SUBROUTINE BLAS_xHP_LRSCALE( UPLO, N, D, INCD, AP )
all:
      INTEGER            INCD, K, LDA, N, UPLO
      <ctype>             A( LDA, * ) or AP( * ), D( * )
```

- C binding:

```
Hermitian:
void BLAS_xhe_lrscale( enum blas_order_type order, enum blas_uplo_type uplo,
                       int n, const ARRAY d, int incd, ARRAY a, int lda );
Hermitian Band:
void BLAS_xhb_lrscale( enum blas_order_type order, enum blas_uplo_type uplo,
                       int n, int k, const ARRAY d, int incd, ARRAY a,
                       int lda );
Hermitian Packed:
void BLAS_xhp_lrscale( enum blas_order_type order, enum blas_uplo_type uplo,
                       int n, const ARRAY d, int incd, ARRAY ap );
```

{GE,GB}_DIAG_SCALE_ACC (Diagonal scaling and accumulation) $A \leftarrow A + BD$

These routines perform the diagonal scaling of a general (or banded) matrix B and accumulate the result in the matrix A. This routine returns immediately if m or n or kl or ku (for band matrices) is less than or equal to zero. As described in section 2.5.3, the value incd less than zero is permitted. However, if incd is equal to zero, an error flag is set and passed to the error handler. For the routine GE_DIAG_SCALE_ACC, if lda or ldb is less than one or lda or ldb is less than m, an error flag is set and passed to the error handler. For the C bindings of GE_DIAG_SCALE_ACC, if `order = blas_rowmajor` and if lda or ldb is less than one or lda or ldb is less than n, an error flag is set and passed to the error handler; if `order = blas_colmajor` and if lda or ldb is less than one or lda or ldb is less than m, an error flag is set and passed to the error handler. For the routine GB_DIAG_SCALE_ACC, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
    SUBROUTINE ge_diag_scale_acc( b, d, a )
Band:
    SUBROUTINE gb_diag_scale_acc( b, m, kl, d, a )
all:
      <type>(<wp>), INTENT (IN) :: b(:,:), d(:)
      <type>(<wp>), INTENT (INOUT) :: a(:,:)
      INTEGER, INTENT (IN) :: m, kl
```

```
    where
      a has shape (m,n)
      b has shape (m,n) for general matrix
                  (l,n) for general banded matrix (l > kl)
      d has shape (n)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_DIAG_SCALE_ACC( M, N, B, LDB, D, INCD, A,
     $                                    LDA )
Band:
      SUBROUTINE BLAS_xGB_DIAG_SCALE_ACC( M, N, KL, KU, B, LDB, D, INCD,
     $                                    A, LDA )
all:
      INTEGER            INCD, KL, KU, LDA, LDB, M, N
      <type>             A( LDA, * ), B( LDB, * ), D( * )
```

- C binding:

```
General:
void BLAS_xge_diag_scale_acc( enum blas_order_type order, int m, int n,
                              const ARRAY b, int ldb, const ARRAY d,
                              int incd, ARRAY a, int lda );
General Band:
void BLAS_xgb_diag_scale_acc( enum blas_order_type order, int m, int n,
                              int kl, int ku, const ARRAY b, int ldb,
                              const ARRAY d, int incd, ARRAY a, int lda );
```

{GE,SY,SB,SP}_ACC (Matrix accumulation and scale) $B \leftarrow \alpha A + \beta B$, $B \leftarrow \alpha A^T + \beta B$

These routines scale a matrix A (or its transpose) and scale a matrix B and accumulate the result in the matrix B. Matrices A and B have the same storage format. These routines return immediately if alpha is equal to zero and beta is equal to one, or if m (for nonsymmetric matrices) or n or k (for symmetric band matrices) is less than or equal to zero. As described in section 2.5.3, for the routine GE_ACC, if lda or ldb is less than one or lda or ldb is less than m, an error flag is set and passed to the error handler. For the C bindings for GE_ACC, if `order = blas_rowmajor` and if lda or ldb is less than one or lda or ldb is less than n, an error flag is set and passed to the error handler; if `order = blas_colmajor` and if lda or ldb is less than one or lda or ldb is less than m, an error flag is set and passed to the error handler. For the routine SY_ACC, if lda or ldb is less than one or lda or ldb is less than n, an error flag is set and passed to the error handler. For the routine SB_ACC, if lda or ldb is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
    SUBROUTINE ge_acc( a, b [, trans] [, alpha] [, beta] )
```

```
Symmetric:
    SUBROUTINE sy_acc( a, b [, uplo] [, trans] [, alpha] [, beta] )
Symmetric Band:
    SUBROUTINE sb_acc( a, b [, uplo] [, trans] [, alpha] [, beta] )
Symmetric Packed:
    SUBROUTINE sp_acc( ap, bp [, uplo] [, trans] [, alpha] [, beta] )
all:
      <type>(<wp>), INTENT(IN) :: a(:,:) | ap(:)
      <type>(<wp>), INTENT(INOUT) :: b(:,:) | bp(:)
      TYPE (blas_uplo_type), INTENT (IN), OPTIONAL :: uplo
      TYPE (blas_trans_type), INTENT (IN), OPTIONAL :: trans
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
```

The default value for β is 1.0 or (1.0,0.0).

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_ACC( TRANS, M, N, ALPHA, A, LDA, BETA, B,
     $                         LDB )
Symmetric:
      SUBROUTINE BLAS_xSY_ACC( UPLO, TRANS, N, ALPHA, A, LDA, BETA, B,
     $                         LDB )
Symmetric Band:
      SUBROUTINE BLAS_xSB_ACC( UPLO, TRANS, N, K, ALPHA, A, LDA, BETA,
     $                         B, LDB )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_ACC( UPLO, TRANS, N, ALPHA, AP, BETA, BP )
all:
      INTEGER            K, LDA, LDB, M, N, TRANS, UPLO
      <type>             ALPHA, BETA
      <type>             A( LDA, * ) or AP( * ), B( LDB, * ) or BP( * )
```

- C binding:

```
General:
void BLAS_xge_acc( enum blas_order_type order, enum blas_trans_type trans,
                   int m, int n, SCALAR_IN alpha, const ARRAY a, int lda,
                   SCALAR_IN beta, ARRAY b, int ldb );
Symmetric:
void BLAS_xsy_acc( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, int n, SCALAR_IN alpha,
                   const ARRAY a, int lda, SCALAR_IN beta, ARRAY b, int ldb );
Symmetric Band:
void BLAS_xsb_acc( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, int n, int k, SCALAR_IN alpha,
                   const ARRAY a, int lda, SCALAR_IN beta, ARRAY b, int ldb );
Symmetric Packed:
```

```
void BLAS_xsp_acc( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, int n, SCALAR_IN alpha,
                   const ARRAY ap, SCALAR_IN beta, ARRAY bp );
```

{GB,TR,TB,TP}_ACC (Matrix accumulation and scale) $B \leftarrow \alpha A + \beta B$

These routines scale matrices A and B and accumulate the result in the matrix B. Matrices A and B have the same storage format. These routines return immediately if alpha is equal to zero and beta is equal to one, or if m or kl or ku (for general band matrices) or n or k (for triangular band matrices) is less than or equal to zero. For the routine GB_ACC, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler. For the routines TR_ACC and TP_ACC, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine TB_ACC, if lda is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General Band:
    SUBROUTINE gb_acc( a, m, kl, b [, alpha] [, beta] )
Triangular:
    SUBROUTINE tr_acc( a, b [, uplo] [, diag] [, alpha] [, beta] )
Triangular Band:
    SUBROUTINE tb_acc( a, b [, uplo] [, diag] [, alpha] [, beta] )
Triangular Packed:
    SUBROUTINE tp_acc( ap, bp [, uplo] [, diag] [, alpha] [, beta] )
all:
      <type>(<wp>), INTENT(IN) :: a(:,:) | ap(:)
      INTEGER, INTENT (IN) :: m, kl
      <type>(<wp>), INTENT(INOUT) :: b(:,:) | bp(:)
      TYPE (blas_uplo_type), INTENT (IN), OPTIONAL :: uplo
      TYPE (blas_diag_type), INTENT (IN), OPTIONAL :: diag
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
```

The default value for β is 1.0 or (1.0,0.0).

- Fortran 77 binding:

```
General Band:
      SUBROUTINE BLAS_xGB_ACC( M, N, KL, KU, ALPHA, A, LDA, BETA, B,
     $                         LDB )
Triangular:
      SUBROUTINE BLAS_xTR_ACC( UPLO, DIAG, N, ALPHA, A, LDA, BETA, B,
     $                         LDB )
Triangular Band:
      SUBROUTINE BLAS_xTB_ACC( UPLO, DIAG, N, K, ALPHA, A, LDA, BETA, B,
     $                         LDB )
Triangular Packed:
      SUBROUTINE BLAS_xTP_ACC( UPLO, DIAG, N, ALPHA, AP, BETA, BP )
```

```
all:
      INTEGER            DIAG, K, KL, KU, LDA, LDB, M, N, UPLO
      <type>             ALPHA, BETA
      <type>             A( LDA, * ) or AP( * ), B( LDB, * ) or BP( * )
```

- C binding:

```
General Band:
void BLAS_xgb_acc( enum blas_order_type order, int m, int n, int kl, int ku,
                   SCALAR_IN alpha, const ARRAY a, int lda, SCALAR_IN beta,
                   ARRAY b, int ldb );
Triangular:
void BLAS_xtr_acc( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_diag_type diag, int n, SCALAR_IN alpha,
                   const ARRAY a, int lda, SCALAR_IN beta, ARRAY b, int ldb );
Triangular Band:
void BLAS_xtb_acc( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_diag_type diag, int n, int k, SCALAR_IN alpha,
                   const ARRAY a, int lda, SCALAR_IN beta, ARRAY b, int ldb );
Triangular Packed:
void BLAS_xtp_acc( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_diag_type diag, int n, SCALAR_IN alpha,
                   const ARRAY ap, SCALAR_IN beta, ARRAY bp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_ADD (Matrix add and scale) $C \leftarrow \alpha A + \beta B$

This routine scales two matrices A and B and stores their sum in a matrix C. Matrices A, B, and C have the same storage format. This routine returns immediately if m or kl or ku (for general band matrices) or n or k (for symmetric or triangular band matrices) is less than or equal to zero. For the routine GE_ADD, if lda or ldb is less than one or less than m, an error flag is set and passed to the error handler. For the C bindings for GE_ADD, if `order = blas_rowmajor` and if lda or ldb is less than one or lda or ldb is less than n, an error flag is set and passed to the error handler; if `order = blas_colmajor` and if lda or ldb is less than one or lda or ldb is less than m, an error flag is set and passed to the error handler. For the routine GB_ADD, if lda or ldb is less than kl plus ku plus one, an error flag is set and passed to the error handler. For the routines SY_ADD, TR_ADD, SP_ADD, and TP_ADD, if lda or ldb is less than one or lda or ldb is less than n, an error flag is set and passed to the error handler. For the routines SB_ADD and TB_ADD, if lda or ldb is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
    SUBROUTINE ge_add( a, b, c [, alpha] [, beta] )
General Band:
    SUBROUTINE gb_add( a, m, kl, b, c [, alpha] [, beta] )
Symmetric:
    SUBROUTINE sy_add( a, b, c [, uplo] [, alpha] [, beta] )
```

```
Symmetric Band:
    SUBROUTINE sb_add( a, b, c [, uplo] [, alpha] [, beta] )
Symmetric Packed:
    SUBROUTINE sp_add( ap, bp, cp [, uplo] [, alpha] [, beta] )
Triangular:
    SUBROUTINE tr_add( a, b, c [, uplo] [, diag] [, alpha] [, beta] )
Triangular Band:
    SUBROUTINE tb_add( a, b, c [, uplo] [, diag] [, alpha] [, beta] )
Triangular Packed:
    SUBROUTINE tp_add( ap, bp, cp [, uplo] [, diag] [, alpha] [, beta] )
all:
      <type>(<wp>), INTENT(IN) :: a(:,:) | ap(:)
      INTEGER, INTENT (IN) :: m, kl
      <type>(<wp>), INTENT(IN) :: b(:,:) | bp(:)
      <type>(<wp>), INTENT(OUT) :: c(:,:) | cp(:)
      TYPE (blas_uplo_type), INTENT (IN), OPTIONAL :: uplo
      TYPE (blas_diag_type), INTENT (IN), OPTIONAL :: diag
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
     assuming A, B and C all the same (general, banded or packed) with
     the same size.
      a, b and c have shape (m,n) for general matrix
                            (l,n) for general banded matrix (l > kl)
                            (n,n) for symmetric or triangular
                            (k+1,n) for symmetric banded or triangular
                                    banded (k=band width)
      ap, bp and cp have shape (n*(n+1)/2).
```

The default value for β is 1.0 or (1.0,0.0).

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_ADD( M, N, ALPHA, A, LDA, BETA, B, LDB, C,
     $                         LDC )
General Band:
      SUBROUTINE BLAS_xGB_ADD( M, N, KL, KU, ALPHA, A, LDA, BETA, B,
     $                         LDB, C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_ADD( UPLO, N, ALPHA, A, LDA, BETA, B, LDB,
     $                         C, LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_ADD( UPLO, N, K, ALPHA, A, LDA, BETA, B, LDB,
     $                         C, LDC )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_ADD( UPLO, N, ALPHA, AP, BETA, BP, CP )
Triangular:
      SUBROUTINE BLAS_xTR_ADD( UPLO, DIAG, N, ALPHA, A, LDA, BETA, B,
```

```
     $                        LDB, C, LDC )
Triangular Band:
      SUBROUTINE BLAS_xTB_ADD( UPLO, DIAG, N, K, ALPHA, A, LDA, BETA,
     $                        B, LDB, C, LDC )
Triangular Packed:
      SUBROUTINE BLAS_xTP_ADD( UPLO, DIAG, N, ALPHA, AP, BETA, BP, CP )
all:
      INTEGER            DIAG, K, KL, KU, LDA, LDB, M, N, TRANS, UPLO
      <type>             ALPHA, BETA
      <type>             A( LDA, * ) or AP( * ), B( LDB, * ) or BP( * ),
      <type>             C( LDC, * ) or CP( * )
```

- C binding:

```
General:
void BLAS_xge_add( enum blas_order_type order, int m, int n, SCALAR_IN alpha,
                   const ARRAY a, int lda, SCALAR_IN beta, const ARRAY b,
                   int ldb, ARRAY c, int ldc );
General Band:
void BLAS_xgb_add( enum blas_order_type order, int m, int n, int kl, int ku,
                   SCALAR_IN alpha, const ARRAY a, int lda, SCALAR_IN beta,
                   const ARRAY b, int ldb, ARRAY c, int ldc );
Symmetric:
void BLAS_xsy_add( enum blas_order_type order, enum blas_uplo_type uplo, int n,
                   SCALAR_IN alpha, const ARRAY a, int lda, SCALAR_IN beta,
                   const ARRAY b, int ldb, ARRAY c, int ldc );
Symmetric Band:
void BLAS_xsb_add( enum blas_order_type order, enum blas_uplo_type uplo,
                   int n, int k, SCALAR_IN alpha, const ARRAY a, int lda,
                   SCALAR_IN beta, const ARRAY b, int ldb, ARRAY c, int ldc );
Symmetric Packed:
void BLAS_xsp_add( enum blas_order_type order, enum blas_uplo_type uplo,
                   int n, SCALAR_IN alpha, const ARRAY ap, SCALAR_IN beta,
                   const ARRAY bp, ARRAY cp );
Triangular:
void BLAS_xtr_add( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_diag_type diag, int n, SCALAR_IN alpha,
                   const ARRAY a, int lda, SCALAR_IN beta, const ARRAY b,
                   int ldb, ARRAY c, int ldc );
Triangular Band:
void BLAS_xtb_add( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_diag_type diag, int n, int k, SCALAR_IN alpha,
                   const ARRAY a, int lda, SCALAR_IN beta, const ARRAY b,
                   int ldb, ARRAY c, int ldc );
Triangular Packed:
void BLAS_xtp_add( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_diag_type diag, int n, SCALAR_IN alpha,
                   const ARRAY ap, SCALAR_IN beta, const ARRAY bp,
```

```
                    ARRAY cp );
```

2.8.8 Matrix-Matrix Operations

In the following section, $op(X)$ denotes X, or X^T or X^H where X is a matrix.

GEMM (General Matrix Matrix Product) $C \leftarrow \alpha op(A)op(B) + \beta C$

The routine performs a general matrix matrix multiply $C \leftarrow \alpha op(A)op(B) + \beta C$ where α and β are scalars, and A, B, and C are general matrices. This routine returns immediately if alpha is equal to zero and beta is equal to one, or if m or n or k is less than or equal to zero. If lda is less than one, or $transa = blas\_no\_trans$ and lda is less than m, or $transa \neq blas\_no\_trans$ and lda is less than k, or ldb is less than one, or $transb = blas\_no\_trans$ and ldb is less than k, or $transb \neq blas\_no\_trans$ and ldb is less than n, or ldc is less than one or less than m, an error flag is set and passed to the error handler.

This interface encompasses the Legacy BLAS routine xGEMM.

- Fortran 95 binding:

```
SUBROUTINE gemm( a, b, c [, transa] [, transb] [, alpha] [, beta] )
  <type>(<wp>), INTENT(IN) :: <aa>, <bb>
  <type>(<wp>), INTENT(INOUT) :: <cc>
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transa, transb
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
where
  <aa>  ::= a(:,:) or a(:)
  <bb>  ::= b(:,:) or b(:)
  <cc>  ::= c(:,:) or c(:)
and
  c, rank 2, has shape (m,n)
       a has shape (m,k) if transa = blas_no_trans (the default)
                   (k,m) if transa /= blas_no_trans
                   (m) if rank 1
       b has shape (k,n) if transb = blas_no_trans (the default)
                   (n,k) if transb /= blas_no_trans
                   (n) if rank 1
  c, rank 1, has shape (m)
       a has shape (m,n) if transa = blas_no_trans (the default)
                   (n,m) if transa /= blas_no_trans
       b has shape (n)
```

Rank a	Rank b	Rank c	transa	transb	Operation	Arguments
2	2	2	N	N	$C \leftarrow \alpha AB + \beta C$	real or complex
2	2	2	N	T	$C \leftarrow \alpha AB^T + \beta C$	real or complex
2	2	2	N	H	$C \leftarrow \alpha AB^H + \beta C$	complex
2	2	2	T	N	$C \leftarrow \alpha A^T B + \beta C$	real or complex
2	2	2	T	T	$C \leftarrow \alpha A^T B^T + \beta C$	real or complex
2	2	2	H	N	$C \leftarrow \alpha A^H B + \beta C$	complex
2	2	2	H	H	$C \leftarrow \alpha A^H B^H + \beta C$	complex
2	1	1	N	-	$c \leftarrow \alpha Ab + \beta c$	real or complex
2	1	1	T	-	$c \leftarrow \alpha A^T b + \beta c$	real or complex
2	1	1	H	-	$c \leftarrow \alpha A^H b + \beta c$	complex
1	1	2	-	-	$C \leftarrow \alpha ab^T + \beta C$	real or complex
1	1	2	-	H	$C \leftarrow \alpha ab^H + \beta C$	complex

The functionality of xGEMV, xGER, xGERU, and xGERC are also covered by this generic procedure.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xGEMM( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA,
     $                       B, LDB, BETA, C, LDC )
      INTEGER            K, LDA, LDB, LDC, M, N, TRANSA, TRANSB
      <type>             ALPHA, BETA
      <type>             A( LDA, * ), B( LDB, * ), C( LDC, * )
```

- C binding:

```
void BLAS_xgemm( enum blas_order_type order, enum blas_trans_type transa,
                 enum blas_trans_type transb, int m, int n, int k,
                 SCALAR_IN alpha, const ARRAY a, int lda, const ARRAY b,
                 int ldb, SCALAR_IN beta, ARRAY c, int ldc );
```

SYMM (Symmetric Matrix Matrix Product) $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$

This routine performs one of the symmetric matrix matrix operations $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$ where α and β are scalars, A is a symmetric matrix, and B and C are general matrices. This routine returns immediately if alpha is equal to zero and beta is equal to one, or if m or n is less than or equal to zero. For side equal to blas_left_side, and if lda is less than one or less than m, or if ldb is less than one or less than m, or if ldc is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if lda is less than one or less than n, or if ldb is less than one or less than n, or if ldc is less than one or less than n, an error flag is set and passed to the error handler.

The interfaces encompass the Legacy BLAS routine xSYMM with added functionality for complex symmetric matrices.

- Fortran 95 binding:

```
    SUBROUTINE symm( a, b, c [, side] [, uplo] [, alpha] [, beta] )
      <type>(<wp>), INTENT(IN) :: a(:,:), <bb>
      <type>(<wp>), INTENT(INOUT) :: <cc>
      TYPE (blas_side_type), INTENT(IN), OPTIONAL :: side
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
      <bb>  ::= b(:,:) or b(:)
      <cc>  ::= c(:,:) or c(:)
    and
     c, rank 2, has shape (m,n), b same shape as c
       SY  a has shape (m,m) if side = blas_left_side (the default)
           a has shape (n,n) if side /= blas_left_side
     c, rank 1, has shape (m), b same shape as c
       SY  a has shape (m,m)
```

Rank b	Rank c	side	Operation
2	2	L	$C \leftarrow \alpha AB + \beta C$
2	2	R	$C \leftarrow \alpha BA + \beta C$
1	1	-	$c \leftarrow \alpha Ab + \beta c$

The functionality of xSYMV is covered by symm.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xSYMM( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB,
     $                       BETA, C, LDC )
      INTEGER            LDA, LDB, LDC, M, N, SIDE, UPLO
      <type>             ALPHA, BETA
      <type>             A( LDA, * ), B( LDB, * ), C( LDC, * )
```

- C binding:

```
void BLAS_xsymm( enum blas_order_type order, enum blas_side_type side,
                 enum blas_uplo_type uplo, int m, int n, SCALAR_IN alpha,
                 const ARRAY a, int lda, const ARRAY b, int ldb,
                 SCALAR_IN beta, ARRAY c, int ldc );
```

HEMM (Hermitian Matrix Matrix Product) $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$

This routine performs one of the Hermitian matrix matrix operations $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$ where α and β are scalars, A is a Hermitian matrix, and B and C are general matrices. This routine returns immediately if alpha is equal to zero and beta is equal to one, or if m or n is less than or equal to zero. For side equal to blas_left_side, and if lda is less than one or less than m, or if ldb is less than one or less than m, or if ldc is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if lda is less than one or less than n, or if ldb is less than one or less than n, or if ldc is less than one or less than n, an error flag is set and passed to the error handler.

The interfaces encompass the Legacy BLAS routine xHEMM.

- Fortran 95 binding:

```
Hermitian:
    SUBROUTINE hemm( a, b, c [, side] [, uplo] [, alpha] [, beta] )
      COMPLEX(<wp>), INTENT(IN) :: a(:,:), <bb>
      COMPLEX(<wp>), INTENT(INOUT) :: <cc>
      TYPE (blas_side_type), INTENT(IN), OPTIONAL :: side
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      COMPLEX(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
      <bb>  ::= b(:,:) or b(:)
      <cc>  ::= c(:,:) or c(:)
    and
     c, rank 2, has shape (m,n), b same shape as c
       HE  a has shape (m,m) if "side" = blas_left_side (the default)
           a has shape (n,n) if "side" /= blas_left_side
     c, rank 1, has shape (m), b same shape as c
       HE  a has shape (m,m)
```

Rank b	Rank c	side	Operation
2	2	L	$C \leftarrow \alpha AB + \beta C$
2	2	R	$C \leftarrow \alpha BA + \beta C$
1	1	-	$c \leftarrow \alpha Ab + \beta c$

The functionality of xHEMV is covered by hemm.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xHEMM( SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB,
     $                       BETA, C, LDC )
      INTEGER            LDA, LDB, LDC, M, N, SIDE, UPLO
      <ctype>            ALPHA, BETA
      <ctype>            A( LDA, * ), B( LDB, * ), C( LDC, * )
```

- C binding:

```
void BLAS_xhemm( enum blas_order_type order, enum blas_side_type side,
                 enum blas_uplo_type uplo, int m, int n, CSCALAR_IN alpha,
                 const CARRAY a, int lda, const CARRAY b, int ldb,
                 CSCALAR_IN beta, CARRAY c, int ldc );
```

TRMM (Triangular Matrix Matrix Multiply) $\qquad B \leftarrow \alpha op(T)B$ or $B \leftarrow \alpha Bop(T)$

These routines perform one of the matrix-matrix operations $B \leftarrow \alpha op(T)B$ or $B \leftarrow \alpha Bop(T)$ where α is a scalar, B is a general matrix, and T is a unit, or non-unit, upper or lower triangular (or triangular band) matrix. This routine returns immediately if m, n, or k (for triangular band matrices), is less than or equal to zero. For side equal to blas_left_side, and if ldt is less than one or less than m, or if ldb is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if ldt is less than one or less than n, or if ldb is less than one or less than m, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xTRMM.

- Fortran 95 binding:

```
SUBROUTINE trmm( t, b [, side] [, uplo] [, transt] [, diag] [, alpha] )
  <type>(<wp>), INTENT(IN) :: t(:,:)
  <type>(<wp>), INTENT(INOUT) :: <bb>
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
  TYPE (blas_diag_type), INTENT(IN), OPTIONAL :: diag
  TYPE (blas_side_type), INTENT(IN), OPTIONAL :: side
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transt
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
where
  <bb>  ::= b(:,:) or b(:)
and
 b, rank 2, has shape (m,n)
   TR  t has shape (m,m) if side = blas_left_side (the default)
       t has shape (n,n) if side /= blas_left_side
 b, rank 1, has shape (m)
   TR  t has shape (m,m)
```

Rank b	transt	side	Operation
2	N	L	$B \leftarrow \alpha TB$
2	T	L	$B \leftarrow \alpha T^T B$
2	H	L	$B \leftarrow \alpha T^H B$
2	N	R	$B \leftarrow \alpha BT$
2	T	R	$B \leftarrow \alpha BT^T$
2	H	R	$B \leftarrow \alpha BT^H$
1	N	-	$b \leftarrow \alpha Tb$
1	T	-	$b \leftarrow \alpha T^T b$
1	H	-	$b \leftarrow \alpha T^H b$

The functionality of xTRMV is covered by trmm.

- Fortran 77 binding:

```
  SUBROUTINE BLAS_xTRMM( SIDE, UPLO, TRANST, DIAG, M, N, ALPHA, T,
 $                       LDT, B, LDB )
  INTEGER            DIAG, LDB, LDT, M, N, SIDE, TRANST, UPLO
  <type>             ALPHA
  <type>             T( LDT, * ), B( LDB, * )
```

- C binding:

```
void BLAS_xtrmm( enum blas_order_type order, enum blas_side_type side,
                 enum blas_uplo_type uplo, enum blas_trans_type transt,
                 enum blas_diag_type diag, int m, int n, SCALAR_IN alpha,
                 const ARRAY t, int ldt, ARRAY b, int ldb );
```

TRSM (Triangular Solve) $B \leftarrow \alpha op(T^{-1})B$ or $B \leftarrow \alpha Bop(T^{-1})$

This routine solves one of the matrix equations $B \leftarrow \alpha op(T^{-1})B$ or $B \leftarrow \alpha Bop(T^{-1})$ where α is a scalar, B is a general matrix, and T is a unit, or non-unit, upper or lower triangular matrix. This routine returns immediately if m or n is less than or equal to zero. For side equal to blas_left_side, and if ldt is less than one or less than m, or if ldb is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if ldt is less than one or less than n, or if ldb is less than one or less than m, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xTRSM.

Advice to implementors. Note that no check for singularity, or near singularity is specified for these triangular equation-solving routines. The requirements for such a test depend on the application, and so we felt that this should not be included, but should instead be performed before calling the triangular solver. (*End of advice to implementors.*)

- Fortran 95 binding:

```
SUBROUTINE trsm( t, b [, side] [, uplo] [, transt] [, diag] [, alpha] )
  <type>(<wp>), INTENT(IN) :: t(:,:)
  <type>(<wp>), INTENT(INOUT) :: <bb>
  TYPE (blas_side_type), INTENT(IN), OPTIONAL :: side
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transt
  TYPE (blas_diag_type), INTENT(IN), OPTIONAL :: diag
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
where
  <bb>  ::= b(:,:) or b(:)
and
 b, rank 2, has shape (m,n)
   TR  t has shape (m,m) if side = blas_left_side (the default)
       t has shape (n,n) if side /= blas_left_side
 b, rank 1, has shape (m)
   TR  t has shape (m,m)
```

Rank b	transt	side	Operation
2	N	L	$B \leftarrow \alpha T^{-1} B$
2	T	L	$B \leftarrow \alpha T^{-T} B$
2	H	L	$B \leftarrow \alpha T^{-H} B$
2	N	R	$B \leftarrow \alpha B T^{-1}$
2	T	R	$B \leftarrow \alpha B T^{-T}$
2	H	R	$B \leftarrow \alpha B T^{-H}$
1	N	-	$b \leftarrow \alpha T^{-1} b$
1	T	-	$b \leftarrow \alpha T^{-T} b$
1	H	-	$b \leftarrow \alpha T^{-H} b$

The functionality of xTRSV is covered by trsm.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xTRSM( SIDE, UPLO, TRANST, DIAG, M, N, ALPHA,
     $                       T, LDT, B, LDB )
      INTEGER            DIAG, LDB, LDT, M, N, SIDE, TRANST, UPLO
      <type>             ALPHA
      <type>             T( LDT, * ), B( LDB, * )
```

- C binding:

```
void BLAS_xtrsm( enum blas_order_type order, enum blas_side_type side,
                 enum blas_uplo_type uplo, enum blas_trans_type transt,
                 enum blas_diag_type diag, int m, int n, SCALAR_IN alpha,
                 const ARRAY t, int ldt, ARRAY b, int ldb );
```

SYRK (Symmetric Rank K update) $C \leftarrow \alpha AA^T + \beta C$, $C \leftarrow \alpha A^T A + \beta C$

This routine performs one of the symmetric rank k operations $C \leftarrow \alpha AA^T + \beta C$ or $C \leftarrow \alpha A^T A + \beta C$ where α and β are scalars, C is a symmetric matrix, and A is a general matrix. This routine returns immediately if alpha is equal to zero and beta is equal to one, or if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xSYRK with added functionality for complex symmetric matrices.

- Fortran 95 binding:

```
SUBROUTINE syrk( a, c [, uplo] [, trans] [, alpha] [, beta] )
  <type>(<wp>), INTENT(IN) :: <aa>
  <type>(<wp>), INTENT(INOUT) :: c(:,:)
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
where
  <aa>  ::= a(:,:) or a(:)
and
 c has shape (n,n)
 a has shape (n,k) if trans = blas_no_trans (the default)
             (k,n) if trans /= blas_no_trans
             (n) if rank 1
```

Rank a	trans	Operation
2	N	$C \leftarrow \alpha AA^T + \beta C$
2	T	$C \leftarrow \alpha A^T A + \beta C$
1	-	$C \leftarrow \alpha aa^T + \beta C$

The functionality of xSYR is covered by syrk.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xSYRK( UPLO, TRANS, N, K, ALPHA, A, LDA, BETA,
     $                       C, LDC )
      INTEGER            K, LDA, LDC, N, TRANS, UPLO
      <type>             ALPHA, BETA
      <type>             A( LDA, * ), C( LDC, * )
```

- C binding:

```
void BLAS_xsyrk( enum blas_order_type order, enum blas_uplo_type uplo,
                 enum blas_trans_type trans, int n, int k, SCALAR_IN alpha,
                 const ARRAY a, int lda, SCALAR_IN beta, ARRAY c, int ldc );
```

HERK (Hermitian Rank K update) $C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C$

This routine performs one of the Hermitian rank k operations $C \leftarrow \alpha AA^H + \beta C$ or $C \leftarrow \alpha A^H A + \beta C$ where α and β are scalars, C is a Hermitian matrix, and A is a general matrix. This routine returns immediately if alpha is equal to zero and beta is equal to one, or if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xHERK.

- Fortran 95 binding:

```
    SUBROUTINE herk( a, c [, uplo] [, trans] [, alpha] [, beta] )
      COMPLEX(<wp>), INTENT(IN) :: <aa>
      COMPLEX(<wp>), INTENT(INOUT) :: c(:,:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
      REAL(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
      <aa>  ::= a(:,:) or a(:)
    and
     c has shape (n,n)
     a has shape (n,k) if trans = blas_no_trans (the default)
                 (k,n) if trans /= blas_no_trans
                 (n) if rank 1
```

Rank a	trans	Operation
2	N	$C \leftarrow \alpha AA^H + \beta C$
2	T	$C \leftarrow \alpha A^H A + \beta C$
1	-	$C \leftarrow \alpha aa^H + \beta C$

The functionality of xHER is covered by herk.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xHERK( UPLO, TRANS, N, K, ALPHA, A, LDA, BETA, C,
     $                       LDC )
      INTEGER            K, LDA, LDC, N, TRANS, UPLO
      <rtype>            ALPHA, BETA
      <ctype>            A( LDA, * ), C( LDC, * )
```

- C binding:

```
void BLAS_xherk( enum blas_order_type order, enum blas_uplo_type uplo,
                 enum blas_trans_type trans, int n, int k, RSCALAR_IN alpha,
                 const CARRAY a, int lda, RSCALAR_IN beta, CARRAY c, int ldc );
```

SY_TRIDIAG_RK (Symmetric Rank K update with symmetric tridiagonal matrix)

$$C \leftarrow \alpha AJA^T + \beta C, C \leftarrow \alpha A^T JA + \beta C$$

This routine performs one of the symmetric rank k operations $C \leftarrow \alpha AJA^T + \beta C$ or $C \leftarrow \alpha A^T JA + \beta C$ where α and β are scalars, C is a symmetric matrix, A is a general matrix, and J is a symmetric tridiagonal matrix. This routine returns immediately if alpha is equal to zero and beta is equal to one, or if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
    SUBROUTINE sy_tridiag_rk( a, d, e, c [, uplo] [, trans] [, alpha] &
                              [, beta] )
      <type>(<wp>), INTENT(IN) :: a(:,:)
      <type>(<wp>), INTENT(IN) :: d(:), e(:)
      <type>(<wp>), INTENT(INOUT) :: c(:,:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
     c has shape (n,n)
      if trans = blas_no_trans (the default)
         a has shape (n,k)
         d has shape (k)
         e has shape (k-1)
      if trans /= blas_no_trans
         a has shape (k,n)
         d has shape (n)
         e has shape (n-1)
```

Rank a	trans	Operation
2	N	$C \leftarrow \alpha AJA^T + \beta C$
2	T	$C \leftarrow \alpha A^T JA + \beta C$

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xSY_TRIDIAG_RK( UPLO, TRANS, N, K, ALPHA, A, LDA, D,
     $                                E, BETA, C, LDC )
      INTEGER            K, LDA, LDC, N, TRANS, UPLO
      <type>             ALPHA, BETA
      <type>             A( LDA, * ), C( LDC, * ), D( * ), E( * )
```

- C binding:

```
void BLAS_xsy_tridiag_rk( enum blas_order_type order, enum blas_uplo_type uplo,
                          enum blas_trans_type trans, int n, int k,
                          SCALAR_IN alpha, const ARRAY a, int lda,
                          const ARRAY d, const ARRAY e, SCALAR_IN beta,
                          ARRAY c, int ldc );
```

HE_TRIDIAG_RK (Hermitian Rank K update with symmetric tridiagonal matrix)

$$C \leftarrow \alpha AJA^H + \beta C, C \leftarrow \alpha A^H JA + \beta C$$

This routine performs one of the Hermitian rank k operations $C \leftarrow \alpha AJA^H + \beta C$ or $C \leftarrow \alpha A^H JA + \beta C$ where α and β are scalars, C is a Hermitian matrix, A is a general matrix, and J is a symmetric tridiagonal matrix. This routine returns immediately if alpha is equal to zero and beta is equal to one, or if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
    SUBROUTINE he_tridiag_rk( a, d, e, c [, uplo] [, trans] [, alpha] &
                              [, beta] )
      COMPLEX(<wp>), INTENT(IN) :: a(:,:)
      COMPLEX(<wp>), INTENT(IN) :: d(:), e(:)
      COMPLEX(<wp>), INTENT(INOUT) :: c(:,:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
      REAL(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
     c has shape (n,n)
     if trans = blas_no_trans (the default)
        a has shape (n,k)
        d has shape (k)
```

```
         e has shape (k-1)
      if trans /= blas_no_trans
         a has shape (k,n)
         d has shape (n)
         e has shape (n-1)
```

Rank a	trans	Operation
2	N	$C \leftarrow \alpha AJA^H + \beta C$
2	T	$C \leftarrow \alpha A^H JA + \beta C$

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xHE_TRIDIAG_RK( UPLO, TRANS, N, K, ALPHA, A, LDA,
     $                                D, E, BETA, C, LDC )
      INTEGER            K, LDA, LDC, N, TRANS, UPLO
      <rtype>            ALPHA, BETA
      <ctype>            A( LDA, * ), C( LDC, * ), D( * ), E( * )
```

- C binding:

```
void BLAS_xhe_tridiag_rk( enum blas_order_type order, enum blas_uplo_type uplo,
                          enum blas_trans_type trans, int n, int k,
                          RSCALAR_IN alpha, const CARRAY a, int lda,
                          const CARRAY d, const CARRAY e, RSCALAR_IN beta,
                          CARRAY c, int ldc );
```

SYR2K (Symmetric rank 2k update) $C \leftarrow (\alpha A)B^T + B(\alpha A)^T + \beta C$

$$C \leftarrow (\alpha A)^T B + B^T(\alpha A) + \beta C$$

These routines perform the symmetric rank 2k operation $C \leftarrow (\alpha A)B^T + B(\alpha A)^T + \beta C$ or $C \leftarrow (\alpha A)^T B + B^T(\alpha A) + \beta C$ where α and β are scalars, C is a symmetric matrix, and A and B are general matrices. This routine returns immediately if alpha is equal to zero and beta is equal to one, or if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, or if ldb is less than one or less than k, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xSYR2K with added functionality for complex symmetric matrices.

- Fortran 95 binding:

```
    SUBROUTINE syr2k( a, b, c [, uplo] [, trans] [, alpha] [, beta] )
      <type>(<wp>), INTENT(IN) :: <aa>, <bb>
      <type>(<wp>), INTENT(INOUT) :: c(:,:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
```

```
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
where
  <aa>  ::= a(:,:) or a(:)
  <bb>  ::= b(:,:) or b(:)
and
 c has shape (n,n)
 if trans = blas_no_trans (the default)
    a has shape (n,k)
    b has shape (n,k)
 if trans /= blas_no_trans
    a has shape (k,n)
    b has shape (k,n)
```

Rank a	Rank b	trans	Operation
2	2	N	$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$
2	2	T	$C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$
1	1	-	$C \leftarrow \alpha ab^T + \alpha ba^T + \beta C$

The functionality of xSYR2 is covered by syr2k.

- Fortran 77 binding:

```
  SUBROUTINE BLAS_xSYR2K( UPLO, TRANS, N, K, ALPHA, A, LDA, B, LDB,
 $                        BETA, C, LDC )
  INTEGER           K, LDA, LDB, LDC, N, TRANS, UPLO
  <type>            ALPHA, BETA
  <type>            A( LDA, * ), B( LDB, * ), C( LDC, * )
```

- C binding:

```
void BLAS_xsyr2k( enum blas_order_type order, enum blas_uplo_type uplo,
                  enum blas_trans_type trans, int n, int k, SCALAR_IN alpha,
                  const ARRAY a, int lda, const ARRAY b, int ldb,
                  SCALAR_IN beta, ARRAY c, int ldc );
```

HER2K (Hermitian rank 2k update) $C \leftarrow (\alpha A)B^H + B(\alpha A)^H + \beta C$

$$C \leftarrow (\alpha A)^H B + B^H(\alpha A) + \beta C$$

These routines perform the Hermitian rank 2k operation $C \leftarrow (\alpha A)B^H + B(\alpha A)^H + \beta C$ or $C \leftarrow (\alpha A)^H B + B^H(\alpha A) + \beta C$ where α and β are scalars, C is a Hermitian matrix, and A and B are general matrices. This routine returns immediately if alpha is equal to zero and beta is equal to one, or if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, or if ldb is less than one or less than k, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xHER2K.

- Fortran 95 binding:

```
SUBROUTINE her2k( a, b, c [, uplo] [, trans] [, alpha] [, beta] )
  COMPLEX(<wp>), INTENT(IN) :: <aa>, <bb>
  COMPLEX(<wp>), INTENT(INOUT) :: c(:,:)
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
  COMPLEX(<wp>), INTENT(IN), OPTIONAL :: alpha
  REAL(<wp>), INTENT(IN), OPTIONAL :: beta
where
  <aa>  ::= a(:,:) or a(:)
  <bb>  ::= b(:,:) or b(:)
and
 c has shape (n,n)
 a and b have shape (n,k) if trans = blas_no_trans (the default)
                    (k,n) if trans /= blas_no_trans
                    (n) if rank 1
```

Rank a	Rank b	trans	Operation
2	2	N	$C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C$
2	2	T	$C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C$
1	1	-	$C \leftarrow \alpha ab^H + \bar{\alpha} ba^H + \beta C$

The functionality of xHER2 is covered by her2k.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xHER2K( UPLO, TRANS, N, K, ALPHA, A, LDA, B, LDB,
     $                        BETA, C, LDC )
      INTEGER            K, LDA, LDB, LDC, N, TRANS, UPLO
      <ctype>            ALPHA
      <rtype>            BETA
      <ctype>            A( LDA, * ), B( LDB, * ), C( LDC, * )
```

- C binding:

```
void BLAS_xher2k( enum blas_order_type order, enum blas_uplo_type uplo,
                  enum blas_trans_type trans, int n, int k, CSCALAR_IN alpha,
                  const CARRAY A, int lda, const CARRAY b, int ldb,
                  RSCALAR_IN beta, CARRAY c, int ldc );
```

SY_TRIDIAG_R2K (Symmetric rank 2k update with symmetric tridiagonal matrix)

$$C \leftarrow (\alpha AJ)B^T + B(\alpha AJ)^T + \beta C$$
$$C \leftarrow (\alpha AJ)^T B + B^T(\alpha AJ) + \beta C$$

These routines perform the symmetric rank 2k operation $C \leftarrow (\alpha AJ)B^T + B(\alpha AJ)^T + \beta C$ or $C \leftarrow (\alpha AJ)^T B + B^T(\alpha AJ) + \beta C$ where α and β are scalars, C is a symmetric matrix, A and B are general matrices, and J is a symmetric tridiagonal matrix. This routine returns immediately if alpha is equal to zero and beta is equal to one, or if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, or if ldb is less than one or less than k, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE sy_tridiag_r2k( a, d, e, b, c [, uplo] [, trans] &
                           [, alpha] [, beta] )
  <type>(<wp>), INTENT(IN) :: a(:,:), b(:,:)
  <type>(<wp>), INTENT(IN) :: d(:), e(:)
  <type>(<wp>), INTENT(INOUT) :: c(:,:)
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
  COMPLEX(<wp>), INTENT(IN), OPTIONAL :: alpha
  <type>(<wp>), INTENT(IN), OPTIONAL :: beta
where
 c has shape (n,n)
  if trans = blas_no_trans (the default)
     a and b have shape (n,k)
     d has shape (k)
     e has shape (k-1)
  if trans /= blas_no_trans
     a and b have shape (k,n)
     d has shape (n)
     e has shape (n-1)
```

Rank a	Rank b	trans	Operation
2	2	N	$C \leftarrow (\alpha AJ)B^T + B(\alpha AJ)^T + \beta C$
2	2	T	$C \leftarrow (\alpha AJ)^T B + B^T(\alpha AJ) + \beta C$

- Fortran 77 binding:

```
 SUBROUTINE BLAS_xSY_TRIDIAG_R2K( UPLO, TRANS, N, K, ALPHA, A, LDA,
$                                 D, E, B, LDB, BETA, C, LDC )
 INTEGER           K, LDA, LDB, LDC, N, TRANS, UPLO
 <type>            ALPHA, BETA
 <type>            A( LDA, * ), B( LDB, * ), C( LDC, * ),
$                  D( * ), E( * )
```

- C binding:

```
void BLAS_xsy_tridiag_r2k( enum blas_order_type order,
                           enum blas_uplo_type uplo,
                           enum blas_trans_type trans, int n, int k,
                           SCALAR_IN alpha, const ARRAY a, int lda,
                           const ARRAY d, const ARRAY e, const ARRAY b,
                           int ldb, SCALAR_IN beta, ARRAY c, int ldc );
```

HE_TRIDIAG_R2K (Hermitian rank 2k update with symmetric tridiagonal matrix)

$$C \leftarrow (\alpha AJ)B^H + B(\alpha AJ)^H + \beta C$$
$$C \leftarrow (\alpha AJ)^H B + B^H(\alpha AJ) + \beta C$$

These routines perform the symmetric rank 2k operation $C \leftarrow (\alpha AJ)B^H + B(\alpha AJ)^H + \beta C$ or $C \leftarrow (\alpha AJ)^H B + B^H(\alpha AJ) + \beta C$ where α and β are scalars, C is a Hermitian matrix, A and B are general matrices, and J is a symmetric tridiagonal matrix. This routine returns immediately if alpha is equal to zero and beta is equal to one, or if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, or if ldb is less than one or less than k, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE he_tridiag_r2k( a, d, e, b, c [, uplo] [, trans] &
                           [, alpha] [, beta] )
  COMPLEX(<wp>), INTENT(IN) :: a(:,:), b(:,:)
  COMPLEX(<wp>), INTENT(IN) :: d(:), e(:)
  COMPLEX(<wp>), INTENT(INOUT) :: c(:,:)
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
  COMPLEX(<wp>), INTENT(IN), OPTIONAL :: alpha
  REAL(<wp>), INTENT(IN), OPTIONAL :: beta
where
 c has shape (n,n)
  if "trans" = blas_no_trans (the default)
     a and b have shape (n,k)
     d has shape (k)
     e has shape (k-1)
  if "trans" /= blas_no_trans
     a and b have shape (k,n)
     d has shape (n)
     e has shape (n-1)
```

Rank a	Rank b	trans	Operation
2	2	N	$C \leftarrow (\alpha AJ)B^H + B(\alpha AJ)^H + \beta C$
2	2	T	$C \leftarrow (\alpha AJ)^H B + B^H(\alpha AJ) + \beta C$

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xHE_TRIDIAG_R2K( UPLO, TRANS, N, K, ALPHA, A, LDA,
     $                                 D, E, B, LDB, BETA, C, LDC )
      INTEGER            K, LDA, LDB, LDC, N, TRANS, UPLO
      <ctype>            ALPHA
      <rtype>            BETA
      <ctype>            A( LDA, * ), B( LDB, * ), C( LDC, * ),
     $                   D( * ), E( * )
```

- C binding:

```
void BLAS_xhe_tridiag_r2k( enum blas_order_type order,
                           enum blas_uplo_type uplo,
                           enum blas_trans_type trans, int n, int k,
                           CSCALAR_IN alpha, const CARRAY a, int lda,
                           const CARRAY d, const CARRAY e, const CARRAY b,
                           int ldb, RSCALAR_IN beta, CARRAY c, int ldc );
```

2.8.9 Data Movement with Matrices

{GE,GB,SY,SB,SP,TR,TB,TP}_COPY (Matrix copy) $B \leftarrow A$, $B \leftarrow A^T$, $B \leftarrow A^H$

This routine copies a matrix (or its transpose or conjugate-transpose) A and stores the result in a matrix B. Matrices A and B have the same storage format. This routine returns immediately if m (for nonsymmetric matrices), n, k (for symmetric band matrices), or kl or ku (for general band matrices), is less than or equal to zero. For the routine GE_COPY, if trans equal to blas_no_trans, and if lda is less than one or less than m, or if ldb is less than one or less than m, an error flag is set and passed to the error handler. For the routine GE_COPY, if trans equal to blas_trans or blas_conj_trans, and if lda is less than one or less than m, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For the routine GB_COPY, if lda is less than kl plus ku plus one, or if ldb is less than kl plus ku plus one, an error flag is set and passed to the error handler. For the routines SY_COPY and TR_COPY, if lda is less than one or less than n, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For the routines SB_COPY and TB_COPY, if lda is less than k plus one, or if ldb is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
    SUBROUTINE ge_copy( a, b [, trans] )
General Band:
    SUBROUTINE gb_copy( a, b, m, kl [, trans] )
Symmetric:
    SUBROUTINE sy_copy( a, b [, uplo] )
Symmetric Band:
```

```
        SUBROUTINE sb_copy( a, b [, uplo] )
    Symmetric Packed:
        SUBROUTINE sp_copy( ap, bp [, uplo] )
    Triangular:
        SUBROUTINE tr_copy( a, b [, uplo] [,trans] [, diag] )
    Triangular Band:
        SUBROUTINE tb_copy( a, b [, uplo] [,trans] [, diag] )
    Triangular Packed:
        SUBROUTINE tp_copy( ap, bp [, uplo] [,trans] [, diag] )
    all:
          <type>(<wp>), INTENT(IN) :: a(:,:) or ap(:)
          <type>(<wp>), INTENT(OUT) :: b(:,:) or bp(:)
          INTEGER, INTENT(IN) :: m, kl
          TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
          TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
          TYPE (blas_diag_type), INTENT(IN), OPTIONAL :: diag
        where
           a and b have shape (n,n) for symmetric or triangular
                               (k+1,n) for symmetric banded or triangular
                                        banded (k=band width)
           ap and bp have shape (n*(n+1)/2).
        For a general or general banded matrix:
         If trans = blas_no_trans (the default)
            a, b have shape (m,n) for general matrix
                            (l,n) for general banded matrix (l > kl)
         If trans \= blas_no_trans
            a has shape (m,n) and b has shape (n,m) for general matrix
                        (l,n) and b has shape (l,m) for general banded matrix (l > kl)
```

- Fortran 77 binding:

```
    General:
          SUBROUTINE BLAS_xGE_COPY( TRANS, M, N, A, LDA, B, LDB )
    General Band:
          SUBROUTINE BLAS_xGB_COPY( TRANS, M, N, KL, KU, A, LDA, B, LDB )
    Symmetric:
          SUBROUTINE BLAS_xSY_COPY( UPLO, N, A, LDA, B, LDB )
    Symmetric Band:
          SUBROUTINE BLAS_xSB_COPY( UPLO, N, K, A, LDA, B, LDB )
    Symmetric Packed:
          SUBROUTINE BLAS_xSP_COPY( UPLO, N, AP, BP )
    Triangular:
          SUBROUTINE BLAS_xTR_COPY( UPLO, TRANS, DIAG, N, A, LDA, B, LDB )
    Triangular Band:
          SUBROUTINE BLAS_xTB_COPY( UPLO, TRANS, DIAG, N, K, A, LDA, B,
         $                          LDB )
    Triangular Packed:
          SUBROUTINE BLAS_xTP_COPY( UPLO, TRANS, DIAG, N, AP, BP )
```

```
all:
      INTEGER           DIAG, LDA, LDB, N, K, KL, KU, TRANS, UPLO
      <type>            A( LDA, * ) or AP( * ), B( LDB, * ) or BP( * )
```

- C binding:

```
General:
void BLAS_xge_copy( enum blas_order_type order, enum blas_trans_type trans,
                    int m, int n, const ARRAY a, int lda, ARRAY b, int ldb );
General Band:
void BLAS_xgb_copy( enum blas_order_type order, enum blas_trans_type trans,
                    int m, int n, int kl, int ku, const ARRAY a, int lda,
                    ARRAY b, int ldb );
Symmetric:
void BLAS_xsy_copy( enum blas_order_type order, enum blas_uplo_type uplo,
                    int n, const ARRAY a, int lda, ARRAY b, int ldb );
Symmetric Band:
void BLAS_xsb_copy( enum blas_order_type order, enum blas_uplo_type uplo,
                    int n, int k, const ARRAY a, int lda, ARRAY b, int ldb );
Symmetric Packed:
void BLAS_xsp_copy( enum blas_order_type order, enum blas_uplo_type uplo,
                    int n, const ARRAY ap, ARRAY bp );
Triangular:
void BLAS_xtr_copy( enum blas_order_type order, enum blas_uplo_type uplo,
                    enum blas_trans_type trans, enum blas_diag_type diag,
                    int n, const ARRAY a, int lda, ARRAY b, int ldb );
Triangular Band:
void BLAS_xtb_copy( enum blas_order_type order, enum blas_uplo_type uplo,
                    enum blas_trans_type trans, enum blas_diag_type diag,
                    int n, int k, const ARRAY a, int lda, ARRAY b, int ldb );
Triangular Packed:
void BLAS_xtp_copy( enum blas_order_type order, enum blas_uplo_type uplo,
                    enum blas_trans_type trans, enum blas_diag_type diag,
                    int n, const ARRAY ap, ARRAY bp );
```

{HE,HB,HP}_COPY (Matrix copy) $B \leftarrow A$

This routine copies a Hermitian matrix A and stores the result in a matrix B. This routine returns immediately if n or k is less than or equal to zero. For the routine HE_COPY, if lda is less than one or less than n, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For the routine HB_COPY, if lda is less than k plus one, or if ldb is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
Hermitian:
    SUBROUTINE he_copy( a, b [, uplo] )
```

```
Hermitian Band:
    SUBROUTINE hb_copy( a, b [, uplo] )
Hermitian Packed:
    SUBROUTINE hp_copy( ap, bp [, uplo] )
all:
      COMPLEX(<wp>), INTENT(IN) :: a(:,:) or ap(:)
      COMPLEX(<wp>), INTENT(OUT) :: b(:,:) or bp(:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
    where
        a and b have shape (n,n)
                           (k+1,n) for banded (k=band width)
        ap and bp have shape (n*(n+1)/2).
```

- Fortran 77 binding:

```
Hermitian:
      SUBROUTINE BLAS_xHE_COPY( UPLO, N, A, LDA, B, LDB )
Hermitian Band:
      SUBROUTINE BLAS_xHB_COPY( UPLO, N, K, A, LDA, B, LDB )
Hermitian Packed:
      SUBROUTINE BLAS_xHP_COPY( UPLO, N, AP, BP )
all:
      INTEGER            K, LDA, LDB, N, UPLO
      <ctype>            A( LDA, * ) or AP( * ), B( LDB, * ) or BP( * )
```

- C binding:

```
Hermitian:
void BLAS_xhe_copy( enum blas_order_type order, enum blas_uplo_type uplo,
                    int n, const CARRAY a, int lda, CARRAY b, int ldb );
Hermitian Band:
void BLAS_xhb_copy( enum blas_order_type order, enum blas_uplo_type uplo,
                    int n, int k, const CARRAY a, int lda, CARRAY b, int ldb );
Hermitian Packed:
void BLAS_xhp_copy( enum blas_order_type order, enum blas_uplo_type uplo,
                    int n, const CARRAY ap, CARRAY bp );
```

GE_TRANS (Matrix transposition) $A \leftarrow A^T, A \leftarrow A^H$

This routine performs the matrix transposition or conjugate-transposition of a square matrix A, overwriting the matrix A. This routine returns immediately if n is less than or equal to zero. If lda is less than one or less than n, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
    SUBROUTINE ge_trans( a [, conj] )
      <type>(<wp>), INTENT(INOUT) :: a(:,:)
```

```
    TYPE (blas_conj_type), INTENT(IN), OPTIONAL :: conj
  where
    a has shape (n,n)
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xGE_TRANS( CONJ, N, A, LDA )
    INTEGER            CONJ, LDA, N
    <type>             A( LDA, * )
```

- C binding:

```
void BLAS_xge_trans( enum blas_order_type order, enum blas_conj_type conj,
                     int n, ARRAY a, int lda );
```

GE_PERMUTE (Permute matrix) $A \leftarrow PA$, or $A \leftarrow AP$

This routine permutes the rows or columns of a matrix ($A \leftarrow PA$ or $A \leftarrow AP$) by the permutation matrix P. The representation of the permutation vector p is described in section 2.2.6. This routine returns immediately if m or n is less than or equal to zero. As described in section 2.5.3, the value incp less than zero is permitted. However, if incp is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or less than m, an error flag is set and passed to the error handler. For the C bindings, if `order = blas_rowmajor` and if lda is less than one or lda is less than n, an error flag is set and passed to the error handler; if `order = blas_colmajor` and if lda is less than one or lda is less than m, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
  SUBROUTINE ge_permute( p, a [, side] )
    INTEGER, INTENT(IN) :: p(:)
    <type>(<wp>), INTENT(INOUT) :: a(:,:)
    TYPE (blas_side_type), INTENT(IN), OPTIONAL :: side
  where
    a has shape (m,n)
    d has shape (p) where p = m if side = blas_left_side
                          p = n if side = blas_right_side
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xGE_PERMUTE( SIDE, M, N, P, INCP, A, LDA )
    INTEGER            INCP, LDA, M, N, SIDE
    INTEGER            P( * )
    <type>             A( LDA, * )
```

The value of `INCP` may be positive or negative. A negative value of `INCP` applies the permutation in the opposite direction.

- C binding:

```
void BLAS_xge_permute( enum blas_order_type order, enum blas_side_type side,
                       int m, int n, const int *p, int incp, ARRAY a,
                       int lda );
```

The value of `incp` may be positive or negative. A negative value of `incp` applies the permutation in the opposite direction.

2.8.10 Environmental Enquiry

FPINFO (Environmental enquiry)

This routine queries for machine-specific floating point characteristics. Refer to section 1.6 for a list of all possible return values of this routine, and sections A.4, A.5, and A.6, for their respective language dependent representations in Fortran 95, Fortran 77, and C.

- Fortran 95 binding:

```
REAL(<wp>) FUNCTION fpinfo( cmach, prec )
  TYPE (blas_cmach_type), INTENT(IN) :: cmach
  REAL (<wp>), INTENT(IN) :: prec
```

- Fortran 77 binding:

```
<rtype> FUNCTION BLAS_xFPINFO( CMACH )
INTEGER            CMACH
```

- C binding:

```
<rtype> BLAS_xfpinfo( enum blas_cmach_type cmach );
```

Chapter 3

Sparse BLAS

3.1 Overview

A matrix which contains many zero entries is often referred to as being *sparse*. Many problems arising from engineering and scientific computing give rise to large, sparse matrices, hence their importance in numerical linear algebra. Sparsity provides an opportunity to conserve storage and reduce computational requirements by storing only the significant (typically, nonzero) entries.

The Sparse BLAS interface addresses computational routines for *unstructured* sparse matrices. These are matrices that do not possess a special sparsity pattern (such as banded or triangular covered in the previous chapter on Dense/Banded specifications). Two fundamental differences between the Sparse BLAS and other chapters are

- **Functionality:** Only a small subset of the BLAS functionality is specified for sparse matrices – essentially only matrix multiply and triangular solve, along with sparse vector update, dot product and gather/scatter. These are among the basic operations used in solving large sparse linear equations using iterative techniques. Not included are general operations for direct solvers, functions for explicit matrix reordering, or operations in which both operands are sparse (e.g. the product of two sparse matrices).

- **Generic interface:** There is no single "best" method to represent a sparse matrix. The selection of the possible storage format is dependent on the algorithm being used, the original sparsity pattern of the matrix, the underlying computer architecture, together with other considerations such as in what format the data already exists, and so on. Because of this, sparse matrix arguments to the Level 2 and 3 Sparse BLAS routines are not the actual data components but rather a placeholder, or *handle*, which refers to an abstract representation of a matrix. (For portability, this handle is an integer variable.) Unlike the dense BLAS, there are many storage representations for sparse matrices, and this handle-based scheme allows one to write numerical algorithms using the Sparse BLAS independently of the matrix storage scheme.

Several routines are provided to create Sparse BLAS matrices, but the internal representation is implementation dependent. This provides BLAS library developers the best opportunity for optimizing and fine-tuning their kernels for specific situations.

Matrices in the Sparse BLAS can be constructed piece-by-piece, directly from common formats. The result is a matrix handle that can be passed as a parameter to Sparse BLAS computational kernels. Routines are also provided to extract information on a matrix identified by its handle and to release any resources related to the handle when computations with the matrix are completed. Thus, typical use of the Sparse BLAS consists of three phases:

1. create an internal sparse matrix representation and return its handle

   (Sections 3.8.6, 3.8.7, and 3.8.8).

2. use this handle as a parameter in computational Sparse BLAS routines (Sections 3.8.2, 3.8.3, and 3.8.4).

3. when the matrix is no longer needed, call a cleanup routine to free resources associated with the handle (Section 3.8.10).

Note that the releasing a matrix handle does not affect any of the user's data, but only internal BLAS resources (housekeeping data structures and internal copies of matrix data) that are not visible to the user. Thus, program resources available to the user after releasing a matrix handle should be the same as before creating that handle.

In Section 3.3 we describe the functionality of the Level 1, 2 and 3 Sparse BLAS. Section 3.4 provides an overview of the data structures used to express the sparsity of the sparse vectors and matrices, including a discussion of index bases in Section 3.4.2 and repeated indices in Section 3.4.3. Section 3.5.1 illustrates how to initialize Sparse BLAS matrices and Section 3.5.2 how to specify properties of the matrices. Sections 3.6.1– 3.6.3 discuss interface issues. Section 3.7 briefly discusses numerical accuracy and environmental enquiry. Finally, in Section 3.8, we present the interfaces for the kernels, giving details for each specific language binding for Fortran 95, Fortran 77, and C programming languages.

3.2 Naming Conventions

Because this standard addresses multiple language bindings and various precisions, the BLAS routines are typically referred to in the text by their root names. Sparse BLAS root names use the two-letter identifier `US`, for *Unstructured Sparse*, e.g. as in `USMV`, or `USDOT`. These names are a compact way to represent the various instantiations. For example, the root for matrix-vector multiplication, `USMV`, is the general form of routines such as `BLAS_dusmv` (the C version for double-precision), or `BLAS_CUSMV` (the Fortran 77 version of single-precision complex). Functions listed in the Language Bindings Section 3.8 appear under their root name, followed by their detailed language-specific bindings.

Where an `x` appears in the name of a subroutine or function binding, it should be replaced in the call by one of the letters S, D, C, Z to indicate whether the floating-point data types are real single precision, real double precision, complex single precision, or complex double precision, respectively. Notice that, for some calls, this letter and substitution does not appear since the data type is not referenced explicitly and is only accessed through the matrix handle. In the F95 language, generic calls enable the use of this letter to be avoided in all cases except the creation routines.

3.3 Functionality

This section describes the Level 1, 2, and 3 routines defined for sparse vectors and matrices. In all cases only one of the basic operands is sparse, that is there are no sparse-sparse operations. For the sake of compactness, the case involving complex operators is usually omitted, For matrices, whenever a transpose operation is described, the conjugate transpose is implied for the complex case.

3.3.1 Scalar and Vector Operations

USDOT	sparse dot product	$r \leftarrow x^T y$, $r \leftarrow x^H y$
USAXPY	sparse vector update	$y \leftarrow \alpha x + y$
USGA	sparse gather	$x \leftarrow y\|_x$
USGZ	sparse gather and zero	$x \leftarrow y\|_x;\ y\|_x \leftarrow 0$
USSC	sparse scatter	$y\|_x \leftarrow x$

Table 3.1: Sparse Vector Operations

This subsection lists the operations corresponding to the Level 1 Sparse BLAS. Table 3.1 lists the scalar and vector operations. The following notation is used: r and α are scalars, x is a compressed sparse vector, y is a dense vector, and $y|_x$ refers to the entries of y that have common indices with the sparse vector x. Details of the sparse vector storage format are given in Section 3.4.1.

3.3.2 Matrix-Vector Operations

USMV	sparse matrix/vector multiply	$y \leftarrow \alpha A x + y$ $y \leftarrow \alpha A^T x + y$ $y \leftarrow \alpha A^H x + y$
USSV	sparse triangular solve	$x \leftarrow \alpha T^{-1} x$ $x \leftarrow \alpha T^{-T} x$ $x \leftarrow \alpha T^{-H} x$

Table 3.2: Sparse Matrix-Vector Operations

Table 3.2 lists matrix/vector (Level 2) operations. The notation A represents a sparse matrix and T denotes a sparse triangular matrix. x and y are dense vectors, α is a scalar.

3.3.3 Matrix-Matrix Operations

USMM	sparse matrix/matrix multiply	$C \leftarrow \alpha AB + C$ $C \leftarrow \alpha A^T B + C$ $C \leftarrow \alpha A^H B + C$
USSM	sparse triangular solve	$B \leftarrow \alpha T^{-1} B$ $B \leftarrow \alpha T^{-T} B$ $B \leftarrow \alpha T^{-H} B$

Table 3.3: Sparse Matrix-Matrix Operations

Table 3.3 lists matrix/matrix (Level 3) operations, using the following notation: α is a scalar, A denotes a general sparse matrix, T denotes a sparse triangular matrix. B and C are dense matrices.

3.4 Describing sparsity

3.4.1 Sparse Vectors

Sparse vectors are represented by a pair of conventional vectors, one denoting the nonzero values and the other denoting their indices. That is, if x is a vector that we wish to represent in sparse format, then it is represented by a one-dimensional array, X, of the entries of x, and an integer vector of equal length to X whose values indicate the location in x of the corresponding floating-point values in X. The index values may follow the Fortran convention (where the first element has an index of 1) or the C/C++ convention (where the first element has an index of 0). These are referred to as *1-based* and *0-based* indexing, respectively, and the Sparse BLAS specification usually handles both (see Section 3.4.2). For example, using 1-based (Fortran) indexing, the vector

$$x = \begin{pmatrix} 11.0 & 0.0 & 13.0 & 14.0 & 0.0 \end{pmatrix}$$

can be represented by two vectors as

$$\begin{array}{rcl} \mathrm{X} & = & \begin{pmatrix} 11.0 & 13.0 & 14.0 \end{pmatrix} \\ \mathrm{INDX} & = & \begin{pmatrix} 1 & 3 & 4 \end{pmatrix} \end{array}$$

although the permutation

$$\begin{array}{rcl} \mathrm{X} & = & \begin{pmatrix} 14.0 & 13.0 & 11.0 \end{pmatrix} \\ \mathrm{INDX} & = & \begin{pmatrix} 4 & 3 & 1 \end{pmatrix} \end{array}$$

or any other such permutation is equally valid.

We illustrate the use of this structure, through the Fortran 77 routine for a double precision real sparse dot product :

```
W = BLAS_DUSDOT( CONJ, NZ, X, INDX, Y, INCY )
```

where NZ is the number of nonzero entries in the sparse vector x, the argument X is the double precision vector containing the entries of x, INDX is the index vector for x, Y is a dense vector with INCY defining the stride between consecutive components, and CONJ is a flag specifying if $\bar{x}$ or x is used (although this has no effect in the case of real arguments). This call computes

$$w = \sum_{I=1}^{\mathrm{NZ}} \mathrm{X(I)} * \mathrm{Y(INDX(I))}$$

3.4.2 Index bases

The Fortran and C programming languages utilize different conventions to index entries of a vector. Fortran uses a 1-based convention, (that is $x(1)$ is the first entry of vector x); C assumes 0-based index values (that is $x[0]$ is the first entry of the vector x).

For dense array operations, this difference can often be dealt with by adjustments to the array parameters in function and subroutine calls. For sparse data structures, however, the index information is part of the **semantics** of the data structure, so this must be dealt with explicitly.

The Fortran interface for the Sparse BLAS defaults to a 1-based indexing, while the C interface defaults to 0-base indexing. Both interfaces, however, can explicitly override this default with only **one exception**: the Fortran interfaces to the Level 1 sparse routines. In the following sections, we use 1-based conventions in examples and discussions, unless otherwise stated.

For Level 2 and Level 3 operations, the index base may be specified by the `blas_one_base`/`blas_zero_base` property, which can be set when constructing BLAS matrices (see Section 3.5.2).

3.4.3 Repeated Indices

In general, having the same matrix or vector entry specified multiple times in a sparse representation can lead to ambiguities. There are some cases, however, where it is useful to define the result as the sum of all entries with a common index. For example, the sparse data structure

$$\begin{array}{rcl} N & = & \quad 5 \\ \mathrm{X} & = & (\ 11.0 \quad 13.0 \quad 14.0 \quad 22.0\ ) \\ \mathrm{INDX} & = & (\quad 1 \quad\quad 3 \quad\quad 4 \quad\quad 3\ ) \end{array}$$

may be interpreted as a representation of the vector

$$x \;=\; (\ 11.0 \quad 0.0 \quad 35.0 \quad 14.0 \quad 0.0\ )$$

Analogously, a similar convention can be adopted for sparse matrices: whenever an (i, j) index is specified multiple times, the result is that its corresponding nonzero values are added together. (This is useful, for example, in the assembling of elemental matrices from finite-element formulations as in Section 3.5.6).

Because of possible ambiguities and inefficiencies, the use of repeated indices is not supported in the Level 1 BLAS operations. That is, for those routines the sparse vector parameter must have unique indices, otherwise the computational results are undefined.

3.5 Sparse BLAS Matrices

3.5.1 Creation Routines

A Sparse BLAS matrix and its associated handle are created by a sequence of calls to the routines listed in Sections 3.8.6, 3.8.7, and 3.8.8. A call must first be made to a routine to begin the matrix construction. This can be of three forms depending on whether the input matrix has entries which are scalars or are dense matrices. The calls for the scalar or single entries case have the form

```
CALL DUSCR_BEGIN( m, n, A, istat )        ( Fortran 95 )
CALL BLAS_DUSCR_BEGIN( M, N, A, ISTAT )   ( Fortran 77 )
A = BLAS_duscr_begin( m, n );             ( C )
```

where `m` and `n` are the matrix dimensions and `A` is the matrix handle.

When initializing Sparse BLAS matrices from a block-structured format, two variants of the creation routines may be used. For fixed size $k \times l$ blocks, the declaration

```
CALL DUSCR_BLOCK_BEGIN( mb, nb, k, l, A, istat )        ( Fortran 95 )
CALL BLAS_DUSCR_BLOCK_BEGIN( MB, NB, K, L, A, ISTAT )   ( Fortran 77 )
A = BLAS_duscr_block_begin( Mb, Nb, k, l );             ( C )
```

signifies that the input matrix contains `Mb` $\times$ `Nb` blocks, each of size $k \times l$, that is the total dimensions of the matrix are $(Mb * k) \ \times \ (Nb * l)$.

Likewise, for variable block matrices, the declaration

```
CALL DUSCR_VARIABLE_BLOCK_BEGIN( mb, nb, K, L, A, istat )        ( Fortran 95 )
CALL BLAS_DUSCR_VARIABLE_BLOCK_BEGIN( MB, NB, K, L, A, ISTAT )   ( Fortran 77 )
A = BLAS_duscr_variable_block_begin(Mb, Nb, K, L );              ( C )
```

denotes that the input matrix has a variable block structure denoted by the integer vectors `K` and `L`.

`blas_non_unit_diag`	nonzero diagonal entries are stored (Default)
`blas_unit_diag`	diagonal entries are not stored and assumed to be 1.0
`blas_no_repeated_indices`	indices are unique (Default)
`blas_repeated_indices`	nonzero values of repeated indices are summed
`blas_lower_symmetric`	only lower half of symmetric matrix is specified by user.
`blas_upper_symmetric`	only upper half of symmetric matrix is specified by user.
`blas_lower_hermitian`	only lower half of Hermitian matrix is specified by user.
`blas_upper_hermitian`	only upper half of Hermitian matrix is specified by user.
`blas_lower_triangular`	sparse matrix is lower triangular
`blas_upper_triangular`	sparse matrix is upper triangular
`blas_zero_base`	indices of inserted items are 0-based (Default for C)
`blas_one_base`	indices of inserted items are 1-based (Default for Fortran)
	Applicable for block entries only
`blas_rowmajor`	dense block stored row major order (Default for C)
`blas_colmajor`	dense block stored col major order (Default for Fortran)
`blas_irregular`	general unstructured matrix
`blas_regular`	structured matrix
`blas_block_irregular`	unstructured matrix best represented by blocks
`blas_block_regular`	structured matrix best represented by blocks
`blas_unassembled`	matrix is best represented by cliques

Table 3.4: Matrix properties (can be set by USSP).

3.5.2 Specifying matrix properties

The creation routines allow one to specify various properties about the matrix and optionally provide hints to the underlying BLAS implementation about how the matrix will be used in subsequent BLAS calls, so that possible optimization may take place.

When creating a handle to a BLAS sparse matrix, one or more of the properties in Table 3.4 may be specified with the use of the USSP (set property) routine (See Section 3.8.9). For example,

```
USSP( A, blas_lower_triangular );
USSP( A, blas_unit_diag );
```

denotes a lower triangular matrix, with an implicit unit diagonal.

The input properties (Table 3.4), are mutually exclusive for each category and may be specified only once. The result is undefined if incompatible properties are requested.

An optional description of the sparsity pattern of the matrix may be specified at construction time. These properties are listed as the last group in Table 3.4 and their use may assist the underlying implementation in choosing the most efficient internal data structure for subsequent computation. Note that each description is mutually exclusive. The specification of these properties is optional and does not effect the correctness of the program.

3.5.3 Sparse Matrices: Inserting a Single Entry

The basic insertion routine `USCR_INSERT_ENTRY` allows one to build a sparse matrix, one scalar entry at a time, by specifying its row and column index together with its numeric value. Although there are other insertion routines for special structures (see below) this version is the simplest and most universal, as it allows one to build a BLAS Sparse Matrix from any given format.

`blas_num_rows`	returns the number of rows of matrix
`blas_num_cols`	returns the number of columns of matrix
`blas_num_nonzeros`	returns the number of stored entries
`blas_complex`	matrix values are complex
`blas_real`	matrix values are real
`blas_integer`	matrix values are integer
`blas_double_precision`	matrix values are single precision
`blas_single_precision`	matrix values are double precision
`blas_general`	neither symmetric nor Hermitian (Default)
`blas_symmetric`	sparse matrix is symmetric
`blas_hermitian`	(complex) sparse matrix is Hermitian
`blas_lower_triangular`	sparse matrix is lower triangular
`blas_upper_triangular`	sparse matrix is upper triangular
`blas_zero_base`	indices of inserted items are 0-based (Default for C)
`blas_one_base`	indices of inserted items are 1-based (Default for Fortran)
	Applicable for block entries only
`blas_rowmajor`	dense block stored row major order (Default for C)
`blas_colmajor`	dense block stored col major order (Default for Fortran)
`blas_void_handle`	handle not currently in use
`blas_new_handle`	handle created but no entries inserted so far
`blas_open_handle`	an entry has been inserted but creation not yet finished
`blas_valid_handle`	creation completed (`USCR_END` has been called)

Table 3.5: Matrix properties (can be read by USGP).

3.5.4 Sparse Matrices: Inserting List of Entries

The insertion routine `USCR_INSERT_ENTRIES` allows us to pass a list of entries with arbitrary row and column indices. We describe this list with a similar set of data structures as used for sparse vectors, but now need two integer vectors, one containing the row indices (called INDX) and another containing the column indices (called JNDX).

To illustrate this, consider the following matrix:

$$A = \begin{pmatrix} 1.1 & 0 & 0 & 0 \\ 0 & 2.2 & 0 & 2.4 \\ 0 & 0 & 3.3 & 0 \\ 4.1 & 0 & 0 & 4.4 \end{pmatrix}. \tag{3.1}$$

We can pass in all entries (following a call to one of the BEGIN routines) by defining NZ = 6 and setting

$$\begin{array}{rcl} \text{VAL} & = & (\ 1.1\ \ 2.2\ \ 2.4\ \ 3.3\ \ 4.1\ \ 4.4\ ) \\ \text{INDX} & = & (\ 1\ \ 2\ \ 2\ \ 3\ \ 4\ \ 4\ ) \\ \text{JNDX} & = & (\ 1\ \ 2\ \ 4\ \ 3\ \ 1\ \ 4\ ). \end{array}$$

Note that calls to the C interface would default to using 0-based indices (see Section 3.4.2). The ordering of the entries is arbitrary.

3.5.5 Sparse Matrices: Inserting Row and Column Vectors

The insertion routines `USCR_INSERT_COL` and `USCR_INSERT_ROW` allow us to pass a list of entries that all belong to the same column or row of a matrix. The data structures used to pass the information are identical to those used to describe a sparse vector in Section 3.4.1.

3.5.6 Sparse Matrices: Inserting Cliques

A clique is a two-dimensional array of values with integer row and column vectors that describe how the values will be scattered into the sparse matrix. Such data structures are common in finite element computations. Consider the matrix A in Section 3.5.4. We can pass in the (2,2), (2,4), (4,2) and (4,4) entries as a clique by defining a two-dimensional array

$$\mathrm{VAL} = \begin{pmatrix} 2.2 & 2.4 \\ 0.0 & 4.4 \end{pmatrix} \tag{3.2}$$

and its associated row and column scattering vectors as

$$\begin{aligned} \mathrm{INDX} &= (\;\; 2 \;\; 4 \;\;) \\ \mathrm{JNDX} &= (\;\; 2 \;\; 4 \;\;). \end{aligned}$$

Note that the structure allows cliques to be other than principal submatrices (in which case arrays INDX and JNDX could differ) and indeed allows the clique matrices to be rectangular.

3.6 Interface Issues

3.6.1 Interface Issues for Fortran 95

- Predefined constants for the Sparse BLAS are included in the module "`blas_sparse_namedconstants`". These include the sparse matrix properties constants defined in Tables 3.4 and 3.5. A module "`blas_sparse_proto`" of explicit interfaces to all routines is also provided.
- Sparse matrix/vector indices are assumed to begin at 1 (that is they are 1-based), but can be overridden by specifying `blas_zero_base` at the time of creation.
- The values of the named constants are as specified in Section A.4.
- Error handling is as defined in Section 2.4.6.

The interface example below illustrates multiplying a sparse 4×4 matrix

$$A = \begin{pmatrix} 1.1 & 0 & 0 & 0 \\ 0 & 2.2 & 0 & 2.4 \\ 0 & 0 & 3.3 & 0 \\ 4.1 & 0 & 0 & 4.4 \end{pmatrix} \tag{3.3}$$

with the vector $x = \{1.0, 1.0, 1.0, 1.0\}$ performing the operation $y \leftarrow Ax$. In this example, the sparse matrix is input by point (rather than block) entries.

```fortran
! Fortran 95 example: sparse matrix-vector multiplication

PROGRAM F95_EX
USE blas_sparse

IMPLICIT NONE
INTEGER NMAX, NNZ
PARAMETER (NMAX = 4, NNZ = 6)
INTEGER i, n, a, istat
INTEGER, DIMENSION(:), ALLOCATABLE::indx,jndx
DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE:: val, x, y

ALLOCATE(val(NNZ),x(NMAX),y(NMAX),indx(NNZ),jndx(NNZ))

indx=(/1,2,2,3,4,4/)
jndx=(/1,2,4,3,1,4/)
val=(/1.1,2.2,2.4,3.3,4.1,4.4/)

      N = NMAX
!
!     ----------------------------------
!     Step 1:  Create Sparse BLAS Handle
!     ----------------------------------
!
      CALL  duscr_begin(n,n,a,istat)
!
!     ----------------------------------
!     Step 2:  Insert entries one-by-one
!     ----------------------------------
!
      DO i=1, nnz
        CALL uscr_insert_entry(A, val(i), indx(i), jndx(i), istat)
      END DO
!
!     -----------------------------------------------
!     Step 3:  Complete construction of sparse matrix
!     -----------------------------------------------
!
      CALL uscr_end(a,istat)
!
!     ---------------------------------------------
!     Step 4: Compute Matrix vector product y = A*x
!     ---------------------------------------------
!
      CALL usmv(a,x,y,istat)
!
!     ------------------------------
!     Step 5:  Release Matrix Handle
```

```
!       ------------------------------
!
        CALL usds(a,istat)
        END
```

3.6.2 Interface Issues for Fortran 77

Although Fortran 77 is no longer a standard, Fortran 77 compilers are still heavily used and there are many Fortran applications that, even if compiled with a Fortran 95 compiler, use a subset of the language that is very close to Fortran 77. In addition, we have seen in the C interface to the legacy BLAS (see Chapter B) that a Fortran 77 library can provide the vast majority of functionality required by a higher level interface and greatly reduce the overall amount of work required to develop and support multiple language bindings. For these reasons we provide a Fortran 77 interface to the sparse BLAS.

- Predefined constants for the Sparse BLAS are included in the header file "`blas_namedconstants.h`". These include the sparse matrix properties constants defined in Tables 3.4 and 3.5.
- Sparse matrix/vector indices are assumed to begin at 1 (that is they are 1-based), but can be overridden by specifying `blas_zero_base` at the time of creation.
- The values of the named constants are as specified in Section A.5.
- Error handling is as defined in Section 2.5.6.

The following program illustrates the use of Fortran 77 codes on the matrix 3.3.

```
C     Fortran 77 example: sparse matrix-vector multiplication

      PROGRAM F77_EX
      IMPLICIT NONE
      INCLUDE "blas_namedconstants.h"
      INTEGER NMAX, NNZ
      PARAMETER (NMAX = 4, NNZ = 6)
      INTEGER I, N, ISTAT, A
      INTEGER INDX(NNZ), JNDX(NNZ)
      DOUBLE PRECISION VAL(NNZ), X(NMAX), Y(NMAX)
C
C     ----------------------------------------------
C     Define Matrix, LHS and RHS in Coordinate format
C     ----------------------------------------------
C
      DATA VAL   / 1.1, 2.2, 2.4, 3.3, 4.1, 4.4/
      DATA INDX  /   1,   2,   2,   3,   4,   4/
      DATA JNDX  /   1,   2,   4,   3,   1,   4/
C
      DATA X     / 1., 1., 1., 1./
      DATA Y     / 0., 0., 0., 0./
C
```

```
      N = NMAX
C
C     ----------------------------------
C     Step 1:  Create Sparse BLAS Handle
C     ----------------------------------
C
      CALL BLAS_DUSCR_BEGIN( N, N, A, ISTAT)
C
C     ----------------------------------
C     Step 2:  Insert entries one-by-one
C     ----------------------------------
C
      DO 10 I=1, NNZ
        CALL BLAS_DUSCR_INSERT_ENTRY(A, VAL(I), INDX(I), JNDX(I), ISTAT)
 10   CONTINUE
C
C     -----------------------------------------------
C     Step 3:  Complete construction of sparse matrix
C     -----------------------------------------------
C
      CALL BLAS_USCR_END(A, ISTAT)
C
C     ---------------------------------------------
C     Step 4: Compute Matrix vector product y = A*x
C     ---------------------------------------------
C
      CALL BLAS_DUSMV( BLAS_NO_TRANS, 1.0, A, X, 1, Y, 1, ISTAT )
C
C     ------------------------------
C     Step 5:  Release Matrix Handle
C     ------------------------------
C
      CALL BLAS_USDS(A,ISTAT)

      END
```

3.6.3 Interface Issues for C

- Predefined constants for the Sparse BLAS are included in the header file "`blas_enum.h`". These include the sparse matrix properties constants defined in Tables 3.4 and 3.5.
- Sparse matrix/vector indices are assumed to begin at 0 (that is they are 0-based), but can be overridden by specifying `blas_one_base` at the time of creation.
- Sparse matrix handles are integers, but are `typedef` to `blas_sparse_matrix` for clarity.
- The values of the enumerated types are as specified in Section A.6.
- Error handling is as defined in Section 2.6.9.

The following program illustrates the use of C codes on the matrix 3.3.

```
    /* C example:  sparse matrix/vector multiplication */

    #include "blas_sparse.h"

int main()
{
    const int N = 4;
    const int nz = 6;
    double val[] = { 1.1, 2.2, 2.4, 3.3, 4.1, 4.4 };
    int   indx[] = {   0,   1,   1,   2,   3,   3};
    int   jndx[] = {   0,   1,   3,   2,   0,   3};
    double   x[] = { 1.0, 1.0, 1.0, 1.0 };
    double   y[] = { 0.0, 0.0, 0.0, 0.0 };

    blas_sparse_matrix A;
    int i;
    double alpha = 1.0;

    /*------------------------------------*/
    /* Step 1: Create Sparse BLAS Handle  */
    /*------------------------------------*/

    A = BLAS_duscr_begin( N, N );

    /*-------------------------*/
    /*  Step 2: insert entries */
    /*-------------------------*/

    for (i=0; i<nz; i++)
        BLAS_duscr_insert_entry(A, val[i], indx[i], jndx[i]);

    /*-------------------------------------------------*/
    /* Step 3:  Complete construction of sparse matrix */
    /*-------------------------------------------------*/

    BLAS_uscr_end(A);

    /*------------------------------------------------*/
    /* Step 4:  Compute Matrix vector product y = A*x */
    /*------------------------------------------------*/

    BLAS_dusmv( blas_no_trans, alpha, A, x, 1, y, 1 );

    /*---------------------------------*/
    /* Step 5:  Release Matrix Handle  */
    /*---------------------------------*/
```

```
    BLAS_usds(A);

    return 0;
}
```

3.7 Numerical Accuracy and Environmental Enquiry

All the comments on the accuracy of numerical methods made in Sections 1.6 and 2.7 apply here. In particular, subroutine FPINFO described in Section 2.7 should be used to get floating-point parameters needed for error bounds.

3.8 Language Bindings

3.8.1 Overview

This sections lists BLAS routines by their root name (see Section 3.2) together with their specific bindings for Fortran 95, Fortran 77, and C.

- Level 1 computational routines (Section 3.8.2)
  - `USDOT` sparse dot product
  - `USAXPY` sparse vector update
  - `USGA` sparse gather
  - `USGZ` sparse gather and zero
  - `USSC` sparse scatter
- Level 2 computational routines (Section 3.8.3)
  - `USMV` matrix/vector multiply
  - `USSV` matrix/vector triangular solve
- Level 3 computational routines (Section 3.8.4)
  - `USMM` matrix/matrix multiply
  - `USSM` matrix/matrix triangular solve
- Handle Management routines (Level 2/3) (Section 3.8.5)
  - Creation routine (Section 3.8.6)
    - `USCR_BEGIN` begin construction
    - `USCR_BLOCK_BEGIN` begin block-entry construction
    - `USCR_VARIABLE_BLOCK_BEGIN` begin variable block-entry construction
  - Insertion routines (Section 3.8.7)
    - `USCR_INSERT_ENTRY` add point-entry to construction
    - `USCR_INSERT_ENTRIES` add list of point-entries to construction
    - `USCR_INSERT_COL` add a compressed column to construction

    - * USCR_INSERT_ROW add a compressed row to construction
    - * USCR_INSERT_CLIQUE add a dense matrix clique to construction
    - * USCR_INSERT_BLOCK add a block entry at block coordinate (bi, bj)
  - Completion of construction routine (Section 3.8.8)
    - * USCR_END entries completed; build internal representation
  - Matrix property routines (Section 3.8.9)
    - * USGP get/test for matrix property
    - * USSP set matrix property
  - Destruction routine (Section 3.8.10)
    - * USDS release matrix handle

3.8.2 Level 1 Computational Routines

General conventions: in all Level 1 routines, the following common arguments are used:

- x : a sparse vector x, with nz nonzeros
- indx : an (integer) index vector corresponding to x,
- y : a dense vector
- index_base: (C bindings only.) By convention, the Fortran 77 and Fortran 95 bindings assume that all offsets begin at 1 (that is $x(1)$ is the first entry). For the C language bindings, offsets can start at 0 (the default for C arrays) or 1 (for Fortran compatibility).

Note that, as stated in Section 3.4.3, the result of a Level 1 BLAS operation called with repeated indices in array indx will be undefined. The actual return will be dependent on the implementation.

USDOT (Sparse dot product) $r \leftarrow x^T y$

The function USDOT computes the dot product of sparse vector x with dense vector y. The routine returns a real zero if the length of arrays x and indx are less than or equal to zero. When x and y are complex vectors, the vector components x_i are used unconjugated or conjugated as specified by the operator argument conj. If x and y are real vectors, the operator argument conj has no effect. For the C binding, the lack of a complex data type forces us to return the result in the parameter r.

- Fortran 95 binding:

```
<type>(<wp> FUNCTION usdot( x, indx, y [, conj] )
  INTEGER, INTENT(IN) :: indx(:)
  <type>(<wp>), INTENT(IN) :: x(:), y(:)
  TYPE(blas_conj_type), INTENT(IN), OPTIONAL :: conj
```

- Fortran 77 binding:

```
<type> FUNCTION BLAS_xUSDOT( CONJ, NZ, X, INDX, Y, INCY )
<type>           X( * ), Y( * )
INTEGER          NZ, INDX( * ), INCY
INTEGER          CONJ
```

- C binding:

```
void BLAS_xusdot( enum blas_conj_type conj, int nz, const ARRAY x,
                  const int *indx, const ARRAY y, int incy,
                  SCALAR_INOUT r, enum blas_base_type index_base );
```

USAXPY (Sparse vector update) $y \leftarrow \alpha x + y$

The routine USAXPY scales the sparse vector x by α and adds the result to the dense vector y. If the length of arrays x and indx are less than or equal to zero or if α is equal to zero, this routine returns without modifying y. Note that we do not allow a scaling on the vector y (that is, we do not implement a USAXPBY) as this would change the complexity of our routine because scaling a dense vector requires n operations while the sparse operations are only $O(nz)$. If the dense vector y is to be scaled, the appropriate Level 1 dense BLAS kernel should be used.

- Fortran 95 binding:

```
SUBROUTINE usaxpy( x, indx, y [, alpha] )
  <type>(<wp>), INTENT(IN) :: x(:)
  <type>(<wp>), INTENT(INOUT) :: y(:)
  INTEGER, INTENT(IN) :: indx(:)
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
```

The default value for α is `1.0`.

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSAXPY( NZ, ALPHA, X, INDX, Y, INCY )
<type>           ALPHA
<type>           X( * ), Y( * )
INTEGER          NZ, INDX( * ), INCY
```

- C binding:

```
void BLAS_xusaxpy( int nz, SCALAR_IN alpha, const ARRAY x, const int *indx,
                   ARRAY y, int incy, enum blas_base_type index_base );
```

USGA (Sparse gather into compressed form) $x \leftarrow y|_x$

Using indx to denote the list of indices of the sparse vector x, for each component i in this list, the routine USGA assigns x(i) = y(indx(i)). For example, if x is a sparse vector with nonzeros $\{3.1, 4.9\}$ and indices $\{1, 4\}$ (using 1-based offsets), and y is the dense vector $\{12.7, 68.1, 38.1, 54.0\}$, then the USGA routine changes x to $\{12.7, 54.0\}$. If the length of x and indx is non-positive, this routines returns without any modification to its parameters.

- Fortran 95 binding:

```
SUBROUTINE usga( y, x, indx )
  <type>(<wp>), INTENT(IN) :: y(:)
  <type>(<wp>), INTENT(OUT) :: x(:)
  INTEGER, INTENT(IN) :: indx(:)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSGA( NZ, Y, INCY, X, INDX )
INTEGER           NZ, INDX( * ), INCY
<type>            Y( * ), X( * )
```

- C binding:

```
void BLAS_xusga( int nz, const ARRAY y, int incy, ARRAY x, const int *indx,
                 enum blas_base_type index_base );
```

USGZ (Sparse gather and zero) $x \leftarrow y|_x, \;\; y|_x \leftarrow 0$

This routine combines two operations: (1) a sparse gather of y into x. (see USGA above), followed by (2) setting the corresponding values of y (y(indx(i)) to zero. For example, if x is a sparse vector with nonzeros $\{3.1, 4.9\}$ and indices $\{1, 4\}$ (using 1-based offsets), and y is the dense vector $\{12.7, 68.1, 38.1, 54.0\}$, then the USGA routine changes the nonzero values of x to $\{12.7, 54.0\}$ and changes y to $\{0.0, 68.1, 38.1, 0.0\}$.

- Fortran 95 binding:

```
SUBROUTINE usgz( y, x, indx )
  <type>(<wp>), INTENT(INOUT) :: y(:)
  <type>(<wp>), INTENT(OUT) :: x(:)
  INTEGER, INTENT(IN) :: indx(:)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSGZ( NZ, Y, INCY, X, INDX )
INTEGER           NZ, INDX( * ), INCY
<type>            Y( * ), X( * )
```

- C binding:

```
void BLAS_xusgz( int nz, ARRAY y, int incy, ARRAY x, const int *indx,
                 enum blas_base_type index_base );
```

USSC (Sparse scatter) $y|_x \leftarrow x$

This routine copies the nonzero values of x into the corresponding locations in the dense vector y. For example, if x is a sparse vector with nonzeros $\{3.1, 4.9\}$ and indices $\{1, 4\}$ (using 1-based offsets), and y is the dense vector $\{12.7, 68.1, 38.1, 54.0\}$, then the USSC routine changes y to $\{3.1, 68.1, 38.1, 4.9\}$. If the length of arrays x and indx are less than or equal to zero, this routine returns without any modification to its parameters.

- Fortran 95 binding:

```
SUBROUTINE ussc( x, y, indx )
  <type>(<wp>), INTENT(IN) :: x(:)
  <type>(<wp>), INTENT(INOUT) :: y(:)
  INTEGER, INTENT(IN) :: indx(:)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSSC( NZ, X, Y, INCY, INDX )
INTEGER           NZ, INDX( * ), INCY
<type>            X( * ), Y( * )
```

- C binding:

```
void BLAS_xussc( int nz, const ARRAY x, ARRAY y, int incy, const int *indx,
                 enum blas_base_type index_base );
```

3.8.3 Level 2 Computational Routines

USMV (Sparse Matrix/Vector Multiply)

$$y \leftarrow \alpha\, Ax + y$$
$$y \leftarrow \alpha\, A^T x + y$$

This routine multiplies a dense vector x by a sparse matrix A (or its transpose), and adds it to the vector operand y. The matrix handle A must be valid, i.e. `USGP(A, blas_valid_handle)` must be true, and the precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE usmv( a, x, y, istat [, transa] [, alpha] )
  INTEGER, INTENT(IN) :: a
  <type>(<wp>), INTENT(IN) :: x(:)
  <type>(<wp>), INTENT(INOUT) :: y(:)
  INTEGER, INTENT(OUT) :: istat
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transa
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
```

Default values for `transa` and α are `blas_no_trans` and `1.0`, respectively.

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSMV( TRANSA, ALPHA, A, X, INCX, Y, INCY, ISTAT )
INTEGER          INCX, INCY, A, TRANSA, ISTAT
<type>           ALPHA
<type>           X( * ), Y( * )
```

- C binding:

```
int BLAS_xusmv( enum blas_trans_type transa, SCALAR_IN alpha,
        blas_sparse_matrix A, const ARRAY x, int incx, ARRAY y, int incy );
```

USSV (Sparse Triangular Solve) $x \leftarrow \alpha\, T^{-1} x$
$x \leftarrow \alpha\, T^{-T} x$

This routine solves one of the systems of equations $x \leftarrow \alpha T^{-1} x$ or $x \leftarrow \alpha T^{-T} x$, where x is a dense vector and the matrix T is a triangular sparse matrix. The matrix handle `T` must be valid, i.e. `USGP(T, blas_valid_handle)` is true, must represent a valid triangular matrix, i.e. either `USGP(T, blas_lower_triangular` or `USGP(T, blas_upper_triangular)` must be true, and the precision type of the sparse matrix represented by the handle T must match the remaining floating-point arguments. `istat` is used as an error flag and will be zero if the routine executes successfully. The C binding returns `istat` as the function return value.

- Fortran 95 binding:

```
SUBROUTINE ussv( t, x, istat, [, transt] [, alpha] )
  INTEGER, INTENT(IN) :: t
  <type>(<wp>), INTENT(INOUT) :: x(:)
  INTEGER, INTENT(OUT) :: istat
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transt
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
```

Default values for `transt` and α are `.TRUE.` and `1.0` respectively.

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSSV( TRANST, ALPHA, T, X, INCX, ISTAT )
INTEGER          T, INCX, TRANST, ISTAT
<type>           ALPHA
<type>           X( * )
```

- C binding:

```
int BLAS_xussv( enum blas_trans_type transt, SCALAR_IN alpha,
        blas_sparse_matrix T, ARRAY x, int incx );
```

3.8.4 Level 3 Computational Routines

USMM (Sparse Matrix Multiply)

$$C \leftarrow \alpha\, AB + C$$
$$C \leftarrow \alpha\, A^T B + C$$

This routine multiplies a dense matrix B by a sparse matrix A (or its transpose), and adds it to a dense matrix operand C. A is of size m by n, B is of size of n by nrhs, and C is of size m by nrhs. The input argument nrhs must be greater than zero, and the matrix handle A must be valid, i.e. `USGP(A, blas_valid_handle)` must be true, and the precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE usmm( a, b, c, istat, [, transa] [, alpha] )
  INTEGER, INTENT(IN) :: a
  <type>(<wp>), INTENT(IN) :: b(:,:)
  <type>(<wp>), INTENT(INOUT) :: c(:,:)
  INTEGER, INTENT(OUT) :: istat
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transa
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
```

  Default values for `transa` and α are `.TRUE.` and `1.0`, respectively.

- Fortran 77 binding:

```
 SUBROUTINE BLAS_xUSMM( TRANSA, NRHS, ALPHA, A, B, LDB, C, LDC,
$                       ISTAT )
 INTEGER          NRHS, A, LDB, LDC, TRANSA, ISTAT
 <type>           ALPHA
 <type>           B( LDB, * ), C( LDC, * )
```

- C binding:

```
int BLAS_xusmm( enum blas_order_type order, enum blas_trans_type transa,
                int nrhs, SCALAR_IN alpha, blas_sparse_matrix A,
                const ARRAY B, int ldb, ARRAY C, int ldc );
```

USSM (Sparse Triangular Solve)

$$B \leftarrow \alpha\, T^{-1} B$$
$$B \leftarrow \alpha\, T^{-T} B$$

This routine solves one of the systems of equations $B \leftarrow \alpha T^{-1} B$ or $B \leftarrow \alpha T^{-T} B$, where B is a dense matrix and T is a triangular sparse matrix. T is of size n by n, B is of size of n by nrhs, and C is of size n by nrhs. The input argument nrhs must be greater than zero, and the matrix handle T must be valid, i.e. `USGP(T, blas_valid_handle)` must be true, and represent a valid triangular matrix, i.e. either `USGP(T, blas_lower_triangular` or `USGP(T, blas_upper_triangular)` must be true. The precision type of the sparse matrix represented by the handle T must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE ussm( t, b, istat [, transt] [, alpha] )
  INTEGER, INTENT(IN) :: t
  <type>(<wp>), INTENT(INOUT) :: b(:,:)
  INTEGER, INTENT(OUT) :: istat
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transt
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
```

  Default values for `transt` and α are `.TRUE.` and `1.0` respectively.

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSSM( TRANST, NRHS, ALPHA, T, B, LDB, ISTAT )
INTEGER          NRHS, T, LDB, TRANST, ISTAT
<type>           ALPHA
<type>           B( LDB, * )
```

- C binding:

```
int BLAS_xussm( enum blas_order_type order, enum blas_trans_type transt,
         int nrhs, SCALAR_IN alpha, blas_sparse_matrix T, ARRAY B, int ldb
);
```

3.8.5 Handle Management

The Handle Management routines can be divided into five sets; the creation routines (Section 3.8.6), the insertion routines (Section 3.8.7), the completion routine (Section 3.8.8), matrix property routines (Section 3.8.9), and the destruction routine (Section 3.8.10). A brief discussion of these routines was given in Section 3.5.1.

3.8.6 Creation Routines

USCR_BEGIN (begin point-entry construction) $A \leftarrow (...)$

USCR_BEGIN is used to create a sparse matrix handle where the matrix is held in normal point-wise form (by single scalar entries). m and n must be greater than zero. The `x` prefix in the binding names specifies the scalar type and precision of the matrix, as described in 3.2. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns a new handle as its function return value; this handle is void, i.e. `USGP(return_value, blas_void_handle)` is true, if the routine did not execute successfully.

- Fortran 95 binding:

```
SUBROUTINE xuscr_begin( m, n, a, istat )
  INTEGER, INTENT(IN) :: m, n
  INTEGER, INTENT(OUT) :: a, istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSCR_BEGIN( M, N, A, ISTAT )
INTEGER           M, N, A, ISTAT
```

- C binding:

```
blas_sparse_matrix BLAS_xuscr_begin( int m, int n );
```

USCR_BLOCK_BEGIN (begin constant block-entry construction) $A \leftarrow (...)$

USCR_BLOCK_BEGIN is used to create a sparse matrix handle referring to a block-entry matrix where the blocksize of all entries is constant, that is block entries are $k \times l$. Mb, Nb, k and l must all be greater than zero. The x prefix in the binding names specifies the scalar type and precision of the matrix, as described in 3.2. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns a new handle as its function return value; this handle is void, i.e. `USGP(return_value, blas_void_handle)` is true, if the routine did not execute successfully.

- Fortran 95 binding:

```
SUBROUTINE xuscr_block_begin( Mb, Nb, k, l, a, istat )
  INTEGER, INTENT(IN) :: Mb, Nb, k, l
  INTEGER, INTENT(OUT) :: a, istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSCR_BLOCK_BEGIN( MB, NB, K, L, A, ISTAT )
INTEGER           MB, NB, K, L, A, ISTAT
```

- C binding:

```
blas_sparse_matrix BLAS_xuscr_block_begin( int Mb, int Nb, int k, int l );
```

USCR_VARIABLE_BLOCK_BEGIN (begin variable block-entry construction) $A \leftarrow (...)$

USCR_VARIABLE_BLOCK_BEGIN is used to create a sparse matrix handle referring to a block-entry matrix whose entries may have variable block sizes. The blocksizes are given by the integer arrays K and L such that the dimension of the (i, j) block entry is $K(i) \times L(j)$. Mb, Nb, and all elements of K and L must be greater than zero. The x prefix in the binding names specifies the scalar type and precision of the matrix, as described in 3.2. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns a new handle as its function return value; this handle is void, i.e. `USGP(return_value, blas_void_handle)` is true, if the routine did not execute successfully.

- Fortran 95 binding:

```
SUBROUTINE xuscr_variable_block_begin( Mb, Nb, k, l, a, istat )
  INTEGER, INTENT(IN) :: Mb, Nb, k(:), l(:)
  INTEGER, INTENT(OUT) :: a, istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSCR_VARIABLE_BLOCK_BEGIN( MB, NB, K, L, A, ISTAT )
INTEGER          MB, NB, A, ISTAT
INTEGER          K( * ), L( * )
```

- C binding:

```
blas_sparse_matrix BLAS_xuscr_variable_block_begin( int Mb, int Nb,
                                                     const int *k,
                                                     const int *l );
```

3.8.7 Insertion routines

USCR_INSERT_ENTRY (insert single value at coordinate (i, j)) $A \leftarrow (val, i, j)$

USCR_INSERT_ENTRY is used to build a sparse matrix, passing in one scalar entry at a time. This routine may only be called on a matrix handle that was opened via the USCR_BEGIN routine and has not yet been closed via the USCR_END routine. Furthermore, matrix properties cannot be modified after any insertions, so this call must follow all settings made to the matrix via the `USSP` routine. The matrix handle must be in a new state (i.e `USPG(A, blas_new_handle`) is true) upon the first call to this routine. Upon successful completion, the matrix handle is an open state (i.e. `USGP(A, blas_open_handle)` is true) and subsequent calls to this routine will keep the matrix in this state, until a call to `USCR_END` is issued. The precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE uscr_insert_entry( a, val, i, j, istat )
  INTEGER, INTENT(IN) :: a, i, j
  <type>(<wp>), INTENT(IN) :: val
  INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSCR_INSERT_ENTRY ( A, VAL, I, J, ISTAT )
INTEGER          A, I, J, ISTAT
<type>           VAL
```

- C binding:

```
int BLAS_xuscr_insert( blas_sparse_matrix A, SCALAR val, int i, int j );
```

USCR_INSERT_ENTRIES (insert a list of values in coordinate form (val, i, j)) $A \leftarrow (val, i, j)$

USCR_INSERT_ENTRIES is used to build a sparse matrix, passing in a list of point entries. This routine may only be called on a matrix handle that was opened via the USCR_BEGIN routine and has not yet been closed via the USCR_END routine. Furthermore, matrix properties cannot be modified after any insertions, so this call must follow all settings made to the matrix via the USSP routine. The matrix handle must be in a new state (i.e USPG(A, blas_new_handle) is true) upon the first call to this routine. Upon successful completion, the matrix handle is an open state (i.e. USGP(A, blas_open_handle) is true) and subsequent calls to this routine will keep the matrix in this state, until a call to USCR_END is issued. The precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE uscr_insert_entries( a, val, indx, jndx, istat )
  INTEGER, INTENT(IN) :: a, indx( : ), jndx( : )
  <type>(<wp>), INTENT(IN) :: val ( : )
  INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
 SUBROUTINE BLAS_xUSCR_INSERT_ENTRIES( A, NZ, VAL, INDX, JNDX,
$                                      ISTAT )
 INTEGER          A, NZ, INDX( * ), JNDX( * ), ISTAT
 <type>           VAL( * )
```

- C binding:

```
int BLAS_xuscr_insert_entries( blas_sparse_matrix A, int nz,
                               const ARRAY val,
                               const int *indx, const int *jndx );
```

USCR_INSERT_COL (insert a compressed column) $A \leftarrow (...)$

USCR_INSERT_COL is used to build a sparse matrix, passing in one column at a time. This routine may only be called on a matrix handle that was opened via the USCR_BEGIN routine and has not yet been closed via the USCR_END routine. Furthermore, matrix properties cannot be modified after any insertions, so this call must follow all settings made to the matrix via the USSP routine. The matrix handle must be in a new state (i.e USPG(A, blas_new_handle) is true) upon the first call to this routine. Upon successful completion, the matrix handle is an open state (i.e. USGP(A, blas_open_handle) is true) and subsequent calls to this routine will keep the matrix in this state, until a call to USCR_END is issued. The precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE uscr_insert_col( a, j, val, indx, istat )
  INTEGER, INTENT(IN) ::  a, j, indx(:)
  <type>(<wp>), INTENT(IN) ::  val(:)
  INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSCR_INSERT_COL( A, J, NZ, VAL, INDX, ISTAT )
INTEGER           A, J, NZ, INDX( * ), ISTAT
<type>            VAL( * )
```

- C binding:

```
int BLAS_xuscr_insert_col( blas_sparse_matrix A, int j, int nz,
                           const ARRAY val, const int *indx );
```

USCR_INSERT_ROW (insert a compressed row) $A \leftarrow (...)$

USCR_INSERT_ROW is used to build a sparse matrix, passing in one row at a time. This routine may only be called on a matrix handle that was opened via the USCR_BEGIN routine and has not yet been closed via the USCR_END routine. Furthermore, matrix properties cannot be modified after any insertions, so this call must follow all settings made to the matrix via the USSP routine. The matrix handle must be in a new state (i.e USPG(A, blas_new_handle) is true) upon the first call to this routine. Upon successful completion, the matrix handle is an open state (i.e. USGP(A, blas_open_handle) is true) and subsequent calls to this routine will keep the matrix in this state, until a call to USCR_END is issued. The precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE uscr_insert_row( a, i, val, indx, istat )
  INTEGER, INTENT(IN) ::  a, i, indx(:)
  <type>(<wp>), INTENT(IN) ::  val(:)
  INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSCR_INSERT_ROW( A, I, NZ, VAL, INDX, ISTAT )
INTEGER           A, I, NZ, INDX( * ), ISTAT
<type>            VAL( * )
```

- C binding:

```
int BLAS_xuscr_insert_row( blas_sparse_matrix A, int i, int nz,
                           const ARRAY val, const int *indx );
```

USCR_INSERT_CLIQUE (insert a dense matrix clique) $A \leftarrow (val, i, j)$

USCR_INSERT_CLIQUE is used to build a sparse matrix, passing in a dense matrix val of dimension $\mathsf{k} \times \mathsf{l}$ and corresponding integer arrays containing the list of (i, j) indices describing the clique. This routine may only be called on a matrix handle that was opened via the USCR_BEGIN routine and has not yet been closed via the USCR_END routine. Furthermore, matrix properties cannot be modified after any insertions, so this call must follow all settings made to the matrix via the `USSP` routine. The matrix handle must be in a new state (i.e `USPG(A, blas_new_handle`) is true) upon the first call to this routine. Upon successful completion, the matrix handle is an open state (i.e. `USGP(A, blas_open_handle)` is true) and subsequent calls to this routine will keep the matrix in this state, until a call to `USCR_END` is issued. The precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE uscr_insert_clique( a, val, indx, jndx, istat )
  INTEGER, INTENT(IN) :: a, indx(:), jndx(:)
  <type>(<wp>), INTENT(IN) :: val(:,:)
  INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xUSCR_INSERT_CLIQUE( A, K, L, VAL, LDV, INDX,
     $                                    JNDX, ISTAT )
      INTEGER          A, K, L, LDV, INDX( * ), JNDX( * ), ISTAT
      <type>           VAL( LDV, * )
```

- C binding:

```
int BLAS_xuscr_insert_clique( blas_sparse_matrix A, const int k,
                              const int l, const ARRAY val,
                              const int row_stride, const int col_stride,
                              const int *indx,
                              const int *jndx );
```

USCR_INSERT_BLOCK (insert a block entry at block coordinate (bi, bj)) $A \leftarrow (val, bi, bj)$

USCR_INSERT_BLOCK is used to insert a block entry into a block-entry matrix. This routine may only be called on a matrix handle that was opened with one of the block creation routines (USCR_BLOCK_BEGIN or USCR_VARIABLE_BLOCK_BEGIN) and has not yet been closed via the USCR_END routine. Furthermore, matrix properties cannot be modified after any insertions, so this call must follow all settings made to the matrix via the USSP routine. The matrix handle must be in a new state (i.e USPG(A, blas_new_handle) is true) upon the first call to this routine. Upon successful completion, the matrix handle is an open state (i.e. USGP(A, blas_open_handle) is true) and subsequent calls to this routine will keep the matrix in this state, until a call to USCR_END is issued. The dimensions of the block entry are determined from the blocksize information passed to USCR_BLOCK_BEGIN or USCR_VARIABLE_BLOCK_BEGIN. In the Fortran 77 binding, LDV denotes the leading dimension of the dense array VAL. The precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE uscr_insert_block( a, val, bi, bj, istat )
  INTEGER, INTENT(IN) :: a, bi, bj
  INTEGER, INTENT(OUT) :: istat
  <type>(<wp>), INTENT(IN) :: val(:,:)
```

- Fortran 77 binding:

```
SUBROUTINE F_xUSCR_INSERT_BLOCK( A, VAL, LDV, BI, BJ, ISTAT )
INTEGER          A, LDV, BI, BJ, ISTAT
<type>           VAL( LDV, * )
```

- C binding:

```
int BLAS_xuscr_insert_block( int a, const ARRAY val, int row_stride,
                             int col_stride, int bi, int bj );
```

3.8.8 Completion of construction routine

USCR_END (entries completed; build valid matrix handle) $A \leftarrow (...)$

USCR_END is used to complete the construction phase and build a valid sparse matrix handle. This routine may be called only with a sparse matrix handle that was previously created via the routines USCR_BEGIN, USCR_BLOCK_BEGIN or USCR_VARIABLE_BLOCK_BEGIN. The matrix handle must be in an open or new state, i.e. either USGP(A, blas_open_handle) or USGP(A, blas_new_handle) is true. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE uscr_end( a, istat )
  INTEGER, INTENT(IN) :: a
  INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_USCR_END( A, ISTAT )
      INTEGER          A, ISTAT
```

- C binding:

```
int BLAS_uscr_end( blas_sparse_matrix A );
```

3.8.9 Matrix property routines

USGP (get/test matrix property) *property-value*$\leftarrow A$

For a given sparse matrix A, the routine USGP returns the value of the given property name. The first argument is the matrix handle and the second argument is one of the properties listed in in Table 3.5. Each grouping denotes a subset of mutually exclusive properties. The properties `blas_num_rows`, `blas_num_cols`, and `blas_num_nonzeros` return integer values, all other properties return 1 if true, and 0 otherwise. If the matrix handle is corrupt, i.e. `USGP(A, blas_void_handle)` is true, all other Boolean properties are false, and integer valued properties (`blas_num_rows`, `blas_num_cols`, and `blas_num_nonzeros`) return 0.

- Fortran 95 binding:

```
      SUBROUTINE usgp( a, pname, m )
        INTEGER, INTENT(IN) :: a
        INTEGER, INTENT(IN) :: pname
        INTEGER, INTENT(OUT) :: m
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_USGP( A, PNAME, M )
      INTEGER          A, PNAME, M
```

- C binding:

```
int BLAS_usgp( blas_sparse_matrix A, int pname );
```

USSP (set matrix property) A $\leftarrow$ *property-value*

For a given valid sparse matrix handle A, the routine USSP sets the value of the given matrix property. This routine must be called after the handle has been created, and before any of the INSERT routines have been called. That is, the matrix handle must be in a new state, i.e. `USGP(A, blas_new_handle)` is true. istat is used as an error flag and will be zero if the routine executes successfully and is set to -1 if the handle is corrupt, i.e. if `USGP(A, blas_void_handle)` is true. The C binding returns istat as the function return value.

The first argument is the matrix handle; the second argument is one of the properties listed in in Table 3.4. Each grouping denotes a subset of mutually exclusive properties.

If two incompatible properties from the same group are set, the results are undefined. For example, the sequence

```
BLAS_ussp(A, blas_zero_base);
BLAS_ussp(A, blas_one_base);
```

leads to an ambiguity and the resulting handle is void (i.e. `USGP(A, blas_void_handle)` is true). It is possible to guard against this by testing the properties first.

- Fortran 95 binding:

```
SUBROUTINE ussp( a, pname, istat )
INTEGER, INTENT(INOUT) :: a
INTEGER, INTENT(IN) :: pname
INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_USSP( A, PNAME, ISTAT )
INTEGER           A, PNAME, ISTAT
```

- C binding:

```
int BLAS_ussp( blas_sparse_matrix A, int pname );
```

3.8.10 Destruction routine

USDS (release matrix handle) $(...) \leftarrow A$

The routine USDS releases any memory internally used by the sparse matrix handle A. The handle must have been previously closed by the `USCR_END` routine, i.e. `USGP(A, blas_valid_handle)` must be true. It turns this into a handle that is no longer in use, i.e. `USGP(A, blas_void_handle)` is true. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE usds( a, istat )
INTEGER, INTENT(IN) :: a
INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_USDS( A, ISTAT )
INTEGER           A, ISTAT
```

- C binding:

```
int BLAS_usds( blas_sparse_matrix A );
```

Chapter 4

Extended and Mixed Precision BLAS

4.1 Overview

This Chapter describes *extended* and *mixed* precision implementations of the BLAS described in other chapters. Extended precision is used only internally to the BLAS; the input and output arguments remain as before. Extended precision permits us to implement some algorithms that may be simpler, more accurate, and sometimes even faster than without it. Mixed precision refers to having some input/output parameters that are both single precision and double precision, or both real and complex. Mixed precision similarly permits us to write simpler or faster algorithms. But given the complexity that could result by allowing too many combinations of types and precisions, we must choose a parsimonious subset that is both useful and reasonable to implement.

The rest of this chapter is organized as follows. Section 4.2 summarizes the designs goals and decisions that guide our design, with details left to [42]. Section 4.3 summarizes the functions supported in extended and mixed precision. This includes a discussion of the error bounds that routines must satisfy. Section 4.4 summarizes the issues in our design of language bindings for Fortran 95, Fortran 77 and C. Section 4.5 contains the detailed calling sequences for the subroutines in the three languages. A complete justification of our design appears in [42].

4.2 Design Goals and Summary

Our proposal to have extended and mixed precision in the BLAS is motivated by the following facts:

- A number of important linear algebra algorithms can become simpler, more accurate and sometimes faster if internal computations carry more precision (and sometimes more range) than is used for the input and output arguments. These include linear system solving, least squares problems, and eigenvalue problems. Often the benefits of wider arithmetic cost only a small fractional addition to the total work.

- For single precision input, the computer's native double precision is a way to achieve these benefits easily on all commercially significant computers, at least when only a few extra-precision operations are needed. (Crays and their emulators implement 64-bit single in hardware and much slower 128-bit double in software, so if a great many double precision operations are needed, these machines will slow down significantly.)

- Intel and similar processors are designed to run fastest performing arithmetic to the full 80-bit width, wider than double precision, of their internal registers. These computers confer some benefits of wider arithmetic at little or no performance penalty. Some BLAS on these computers already perform wider arithmetic internally but, without knowing this for sure, programmers cannot exploit it.

- All computers can simulate quadruple precision or something like it in software at the cost of arithmetic slower than double precision by at worst an order of magnitude. Less slowdown is incurred for a rough double-double precision on machines (IBM RS/6000, PowerPC/Mac, SGI/MIPS R8000,HP PA RISC 2.0) with special fused multiply-accumulate instructions. Since some algorithms require very little extra precise arithmetic to get a large benefit, the slowdown is practically negligible.

Given the variety of implementation techniques hinted at above, we need to carefully examine the costs and benefits of exploiting various arithmetic features beyond the most basic ones, and choose a parsimonious subset that

Goal 1: is reasonable to implement,

Goal 2: supports some if not all important application examples,

Goal 3: is easy to use,

Goal 4: encourages the writing of portable code, and

Goal 5: accommodates growth as we learn about new algorithms exploiting our arithmetic features.

Here is an outline of our design decisions. These are discussed and justified in detail in [42].

1. We will not require that the user explicitly declare or use any new extended precision data types, i.e. beyond the standard single and double precisions, since these are not supported in a standard way by every language and compiler. Thus the only extended precision that we mandate will be hidden inside the BLAS, and so can be implemented in any convenient machine dependent way. This supports Goals 1, 3 and 4 above.

2. This internal extended precision will support most of the application examples listed in [42], supporting Goal 2.

3. Since we cannot predict all the future applications of extended or mixed precision, we will accommodate growth by making our proposal as orthogonal as possible to the rest of this proposal, showing how to take any BLAS routine, determine whether extra precision is worth using (since sometimes it is not), and define the extended precision version if it is. This supports Goal 5.

4. Since the number of possible routines with mixed precision inputs is very large, we will specify a small subset of mixed precision routines which seems to cover most foreseeable needs. This supports Goals 1 and 2.

5. In order to easily estimate error bounds in code by running with different internal precisions and then comparing the answers, (see Example 8 in [42]), we need to be able to specify the extended precision at runtime; we will do this with a variable we will call PREC. This supports Goal 2.

6. Since different machines may best support extended precision in different ways, PREC could potentially take on many machine-dependent values. Instead we have chosen a parsimonious subset that will be available on all machines, permitting the implementor to support others if desired. This supports all the Goals 1 and 4 above.
7. Since the precision specified by one value of PREC can still have different implementations and so different error bounds on different machines, we have specified environmental enquiries for the user to be able to discover the actual machine precision (or over/underflow thresholds) used at runtime. This lets the user pick appropriate stopping criteria for iterations, etc. This supports Goals 3 and 4.

4.3 Functionality

This section describes the functionality of extended and mixed precision BLAS in a language independent way. Section 4.3.1 describes how extra precision is specified via the PREC argument. Section 4.3.2 describes in general what kind of mixed precision operations will be supported. Section 4.3.3 describes the error bounds that BLAS operations must satisfy; this is where the semantics of "extra precision" are precisely specified. Finally, section 4.3.4 lists the functions that will be supported in extra and/or mixed precision.

4.3.1 Specifying Extra Precision

The internal precision to be used by an extended precision routine will be specified by an argument called PREC. It is not entirely straightforward to describe PREC because even on a single machine there may be multiple ways of implementing wider-than-double-precision arithmetic (see [42]).

To encourage portability, we specify names for precisions that may map to different formats and techniques on different machines. As discussed in section 1.6, historically the words "single" and "double" have referred to very different formats on different architectures. Nonetheless, we all agree on single precision as a word with a certain meaning, and double precision too, meaning twice or more precision than single. The definitions below add two more precisions, whose implementation details are discussed in [42].

PREC = **Single** . This means single precision, whatever single means on the particular machine, language and compiler.

PREC = **Double** . This means double precision, again whatever that means on a particular machine, language and compiler.

PREC = **Indigenous** . This means the widest hardware-supported format available. Its intention is to let the machine run close to top speed, while being as accurate as possible. On some machines this would be a 64-bit format (whether it is called single or double), but on Intel machines and ones like them it means the 80-bit IEEE format of the floating point registers.

PREC = **Extra** . This means anything at least 1.5 times as accurate than double, and in particular wider than 80-bits (see section 4.3.3 for details). An existing quadruple precision format could be used to implement this, but it can probably be implemented implemented more efficiently using native double (or indigenous) operations in a technique called "double-double", described in [42, 46, 47]. It is possible to write a portable and reasonably efficient reference implementation of all proposed routines using these techniques [42].

The actual names for PREC values are specified in section A.3. Here are the rules for using PREC:

1. The internal precision used must always be at least as high as the most precise input or output. So if the user requests less internal precision than in the most precise input or output, then the implementor must use more than requested.

2. The implementor may always use a higher precision than the one requested in the subroutine call, if this is convenient or faster.

3. The precision actually used is available to the user via the environmental enquiry in section 4.3.3.

4. PREC may take on other machine dependent values provided by the implementor, provided these are documented via the environmental enquiry routine.

Advice to implementors: While it appears that as many as seven new implementations of each routine are needed (four when the arguments are single, and three when the arguments are double), in fact fewer are needed: Two exist already as the standard BLAS (single input/output with PREC = Single, and double input/output when PREC = Double), Indigenous = Double or Indigenous = Single on many machines, and wider precision than requested may be used. Thus the only really new implementations may be single input/output with Double or Extra internal precision, and double input/output with Extra internal precision. Of these, only Extra internal precision may need arithmetic not already native to the machine. A reference implementation is described in [42].

4.3.2 Mixed Precision

Suppose a subroutine has several floating point arguments, some scalars and some arrays. Mixed precision refers to permitting these arguments to have different *mathematical types*, meaning real and complex, or different *precisions*, meaning single and double. Some BLAS in Chapter 2 are naturally defined with arguments of mixed mathematical type (e.g. HERK), but most have a single mathematical type; all are defined with the same precision for all arguments.

The permitted combinations of mathematical types and precisions are defined as follows. There are two cases:

1. The mathematical types of the input/output floating point arguments are identical to the BLAS as defined in Chapter 1. All scalar arguments and the output argument (scalar or array) are double precision. At least one array argument must be single precision.

   For example, suppose the function being implemented is matrix-matrix multiplication $C = \alpha \cdot A \cdot B + \beta \cdot C$, where α and β are scalars and A, B and C are arrays. Then the allowed types are as follows (S = Single real, D = Double real, C = Single complex, Z = Double complex).

α	A	B	β	C
D	S	S	D	D
D	S	D	D	D
D	D	S	D	D
Z	C	C	Z	Z
Z	C	Z	Z	Z
Z	Z	C	Z	Z

2. The precision of all floating point arguments must be single, or all must be double. All scalar arguments and the output argument (scalar or array) are complex (unless a scalar argument must be real for mathematical reasons, like α and β in HERK). At least one input array argument must be real.

   For example, suppose the function being implemented is matrix-matrix multiplication as before. Then the allowed types are as follows:

α	A	B	β	C
C	S	S	C	C
C	S	C	C	C
C	C	S	C	C
Z	D	D	Z	Z
Z	D	Z	Z	Z
Z	Z	D	Z	Z

Note that we specify only 16 versions of matrix-matrix multiplication (the 12 mixed ones above, and 4 unmixed), in contrast to the maximum possible $4^5 = 1024$.

4.3.3 Numerical Accuracy and Environmental Enquiries

The machine dependent interpretations of PREC require us to have a more complicated environmental enquiry routine to describe the numerical behavior of the routine in this chapter than the simpler FPINFO routine described in sections 1.6 and 2.7. While FPINFO should still be available for the user to call to get basic properties of the single and double precision floating point types, here we will specify an additional routine FPINFO_X that depends on PREC.

The calling sequence of this function is

```
result = FPINFO_X (CMACH, PREC)
```

Both arguments are input arguments, with the requested information returned as the integer value of FPINFO_X. The exact input values depend on the language, and are described in section 4.4. PREC has the same meaning as before. Input argument CMACH may take on the named constant values below, which are a subset of those permitted by function FPINFO as described in section 2.7. Only the first six values of CMACH from section 2.7 are permitted, because 1) they are sufficient to define the remaining parameters by using the formulas in section 1.6, and 2) the values returned by FPINFO_X are representable integer values, whereas the other possible return values, like the overflow and underflow thresholds, may not be representable in any user-declarable format.

Floating Point parameter	Description
BASE	base of the machine
T	number of (BASE) digits in the mantissa
RND	1 when "proper rounding" occurs in addition, 0 otherwise
IEEE	1 when rounding in addition occurs in "IEEE style", 0 otherwise
EMIN	minimum exponent before (gradual) underflow
EMAX	largest exponent before overflow

We will use the following notation to describe machine parameters derivable from the values returned by FPINFO_X using the formulas in section 1.6:

- ϵ_{PREC} is *relative machine precision* or *machine epsilon* of the internal precision specified by PREC,
- ϵ_o is the machine epsilon for the output precision,
- $\mathrm{OV}_{\mathrm{PREC}}$ and $\mathrm{UN}_{\mathrm{PREC}}$ are the overflow and underflow thresholds for internal precision PREC, and
- OV_o and UN_o are the overflow and underflow thresholds for for the output precision.

Here are the error bounds satisfied by the extra precision routines, and how they depend on ϵ. There are two cases of interest.

1. Suppose each component of the computed result is of the form

$$r_{true} = \alpha \cdot (\sum_{i=1}^{n} a_i \cdot b_i) + \beta \cdot c \ ,$$

where all quantities are scalars. This covers the dot product, scaled vector addition and scaled vector accumulation, all variants of matrix-vector and matrix-matrix products, and low-rank updates (sometimes with α and β taking on special values like zero and one). In this case the error bound, in the absence of over/underflow of any intermediate or output quantities, should satisfy

$$|r_{computed} - r_{true}| \le \gamma(n+2) \cdot \epsilon_{\mathrm{PREC}}(|\alpha| \cdot \sum_{i=1}^{n} |a_i \cdot b_i| + |\beta \cdot c|) + \epsilon_o \cdot |r_{true}| \ .$$

where $\gamma = 1$ if all data is real and $\gamma = 2\sqrt{2}$ if any data is complex.

Rationale: This accommodates all reasonable, non-Strassen based implementations, with real or complex scalars (and conventional multiplication of complex scalars), that perform all intermediate floating point operations with machine epsilon ϵ_{PREC}, with or without a guard digit, before rounding the final result to precision ϵ_o. Underflow is guaranteed to be absent if no intermediate quantity stored in precision PREC is less than $\mathrm{UN}_{\mathrm{PREC}}$ in magnitude (unless its exact value is zero) and $|r_{computed}|$ is not less than UN_o (unless its exact value is zero). Similarly, overflow is guaranteed to be absent if no intermediate quantity in precision PREC= exceeds $\mathrm{OV}_{\mathrm{PREC}}$ in magnitude, and $|r_{computed}|$ does not exceed OV_o. We avoid specifying what happens with underflow, because the implementor may reasonably choose to compute r using $\alpha \cdot (\sum a_i \cdot b_i)$, $\sum(\alpha \cdot a_i) \cdot b_i$ or $\sum a_i \cdot (\alpha \cdot b_i)$ depending on dimensions, and the error bounds in the presence of underflow can differ significantly in these three cases. See [42] for implementation recommendations and detailed error bounds in the presence of underflow.

2. Suppose the computed solution consists of one or more vectors x satisfying an n-by-n triangular system of equations

$$Tx = \alpha b$$

where α is a scalar, b is a vector (or vectors), and T is a triangular matrix. In this case the computed solution, in the absence of over/underflow in intermediate or output quantities, satisfies

$$(T+E)(x_{computed} + e) = \alpha(b+f)$$

where $|E_{ij}| \le \rho n \epsilon_{\mathrm{PREC}} |T_{ij}|$, $|e_i| \le \epsilon_o |x_{computed,i}|$, $|f_i| \le \rho n \epsilon_{\mathrm{PREC}} |b_i|$, $\rho = 1$ if all data is real, and $\rho = 6 + 4\sqrt{2}$ if any data is complex.

Rationale: This accommodates all reasonable, substitution-based methods of solution, with summations evaluated in any order, with all intermediate floating point operations done with machine epsilon ϵ_{PREC}, and with all intermediate quantities stored to the same precision. In particular, this means that the entries of $x_{computed}$ must be temporarily stored with precision ϵ_{PREC} before being rounded to the output precision at the end. Overflow and underflow are defined and treated as before. See [42] for implementation recommendations and detailed error bounds in the presence of underflow.

The values of ϵ_{PREC} must satisfy the following inequalities:

$$\begin{aligned} \epsilon_{DOUBLE} &\le \epsilon_{SINGLE}^2 \\ \epsilon_{INDIGENOUS} &\le \epsilon_{SINGLE} \\ \epsilon_{EXTRA} &\le \epsilon_{DOUBLE}^{1.5} \end{aligned}$$

The first inequality says that double precision is at least twice as accurate (has twice as many significant digits) as single precision. The second inequality says that indigenous is at least as accurate as single precision. The third inequality says that extra precision is at least 1.5 times as accurate (has 1.5 times as many significant digits) as double precision.

Advice to implementors: This is only a lower bound on the number of significant digits in extra precision; most reasonable implementations can get close to twice as many digits as double precision [42]. The lower bound is intended to exclude the use of the 80-bit IEEE format as Extra precision when Double is the 64-bit IEEE format. It is important that $BASE$, T, and RND are chosen so that EPS defined by $EPS = BASE^{1-T}$ if $RND = 0$ and $EPS = .5 * BASE^{1-T}$ if $RND = 1$ can be used for error analysis. For example in the reference implementation of EXTRA precision in [42], $T = 105$ even though 106 bits are stored. Though we do not require this, the simplest way to achieve the error bounds described above is for floating operations $\odot \in \{+, -, *, /\}$ to satisfy the following bounds in the absense of over/underflow: $fl(a \odot b) = (a \odot b)(1 + \delta)$ for some $|\delta| \le EPS$ when a and b are real, $fl(a \pm b) = (a \pm b)(1 + \delta)$ for some $|\delta| \le \sqrt{2} \cdot EPS$ when a and b are complex, $fl(a * b) = (a * b)(1 + \delta)$ for some $|\delta| \le 2\sqrt{2} \cdot EPS$ when a and b are complex, and $fl(a/b) = (a/b)(1 + \delta)$ for some $|\delta| \le (6 + 4\sqrt{2}) \cdot EPS$ when a and b are complex.

The semantics of overflow and underflow are discussed more carefully in [42]; they become more complicated concepts when using implementation techniques like double-double for extra precision. The important properties they should satisfy are

1. In any precision, a quantity greater than OV generates an exception, a $\pm\infty$ symbol, or otherwise somehow indicates its complete loss of precision.

2. In any precision, the error in a floating point operation that might underflow (during some part of the calculation, if for example it is double-double) is described by $fl(a \odot b) = (a \odot b)(1 + \delta) + \eta$, for some $|\delta| \le EPS$ and $|\eta| \le UN$ if a and b are real, and for slighlty larger $|\delta|$ and $|\eta|$ if a and b are complex.

We choose not to specify the overflow and underflow thresholds in more detail, in order not to eliminate innovative ways of implementing extra precision.

4.3.4 Function Tables

As discussed in [42], not all BLAS routines from Chapter 2 are worth converting to extra or mixed precision, so we only include the subset that is worth converting.

Table 4.1 is a subset of Table 2.1 in Chapter 2, Reduction Operations.

Table 4.2 is a subset of Table 2.3 in Chapter 2, Vector Operations.

Table 4.3 is a subset of Table 2.5 in Chapter 2, Matrix-Vector Operations.

Table 4.4 is a subset of Table 2.7 in Chapter 2, Matrix Matrix Operations.

Dot product	$r \leftarrow \beta r + \alpha x^T y$	DOT
Sum	$r \leftarrow \sum_i x_i$	SUM

Table 4.1: Extra and Mixed Precision Reduction Operations

Scaled vector accumulation	$y \leftarrow \alpha x + \beta y,$	AXPBY
Scaled vector addition	$w \leftarrow \alpha x + \beta y$	WAXPBY

Table 4.2: Extra and Mixed Precision Vector Operations

Matrix vector product	$y \leftarrow \alpha A x + \beta y$	GE, GB, SY, SP, SB, HE, HP, HB	MV
	$y \leftarrow \alpha A^T x + \beta y$	GE, GB	MV
	$x \leftarrow \alpha T x,\ x \leftarrow \alpha T^T x$	TR, TB, TP	MV
Summed matrix vector multiplies	$y \leftarrow \alpha A x + \beta B x$	GE	SUM_MV
Triangular solve	$x \leftarrow \alpha T^{-1} x,\ x \leftarrow \alpha T^{-T} x$	TR, TB, TP	SV

Table 4.3: Extra and Mixed Precision Matrix Vector Operations

Matrix matrix product	$C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha A^T B + \beta C$	GE	MM
	$C \leftarrow \alpha AB^T + \beta C,\ C \leftarrow \alpha A^T B^T + \beta C$		
	$C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha BA + \beta C$	SY, HE	MM
Triangular multiply	$B \leftarrow \alpha TB,\ B \leftarrow \alpha BT$	TR	MM
	$B \leftarrow \alpha T^T B,\ B \leftarrow \alpha BT^T$		
Triangular solve	$B \leftarrow \alpha T^{-1} B,\ B \leftarrow \alpha BT^{-1}$	TR	SM
	$B \leftarrow \alpha T^{-T} B,\ B \leftarrow \alpha BT^{-T}$		
Symmetric rank k & $2k$	$C \leftarrow \alpha AA^T + \beta C,\ C \leftarrow \alpha A^T A + \beta C$	SY, HE	RK
updates ($C = C^T$)	$C \leftarrow (\alpha A)B^T + B(\alpha A)^T + \beta C$	SY, HE	R2K

Table 4.4: Extra and Mixed Precision Matrix Matrix Operations

4.4 Interface Issues

This section describes the common issues for our three language bindings: Fortran 95, Fortran 77 and C. Here is a summary of the systematic way we take a subroutine name and its argument list,

Environmental Enquiry	machine epsilon, over/underflow thresholds

Table 4.5: Environmental Enquiries for Extra and Mixed Precision Operations

and modify them to allow for extra or mixed precision:

1. **Subroutine names and mixed precision inputs.** If the language permits a subroutine argument to have more than one type, because it can dispatch the right routine based on the actual type at compile time (Fortran 95, but not Fortran 77 or C), then the subroutine name does not have to change to accommodate mixed precision. Otherwise, a new subroutine name is required, and will be created from the old one by appending characters indicating the types of the arguments.

2. **Subroutine names and extended precision.** If the language permits PREC to be an optional argument (Fortran 95, but not Fortran 77 or C), then the same subroutine name as for the non-extended precision version can be used without change. If a new name is required, it will be formed by appending _X (or _x) to the existing name. If the name has already been modified to accommodate mixed precision, _X (or _x) should be added to the end of the new name.

3. **Location of** PREC **in the calling sequence.** The new calling sequence will consist of the original calling sequence (for the BLAS routine without extra or mixed precision) with PREC appended at the end.

4. **Type of PREC.** It will be a derived type in Fortran 95, an integer in Fortran 77, and an enumerated type in C. Standard names are listed below.

5. **Environmental enquiry function.** Its output type is an integer. The input PREC is specified as above.

4.4.1 Interface Issues for Fortran 95

1. **Subroutine names and mixed precision inputs.** No new subroutine names are needed because we can exploit the optional argument interface of Fortran 95.

2. **Subroutine names and extended precision.** No new subroutine names are needed by letting PREC be an optional argument. The default in the case of no mixed precision is the standard BLAS implementation. The default in the case of mixed precision is at the discretion of the implementor, subject to the constraints of section 4.3.1.

3. **Type of** PREC. PREC is a derived type, as defined in the module `blas_operator_arguments` (see section A.4).

4. **Environmental enquiry function.** fpinfo_x(CMACH,PREC) returns an integer. PREC is as specified above. CMACH is as defined in sections 1.6, 2.7, 4.3.3, and A.4.

5. **Error Handling.** Error handling is as defined in section 2.4.6.

4.4.2 Interface Issues for Fortran 77

As described in Chapter 2, this proposal violates the letter of the ANSI Fortran 77 standard by having subroutine and variable names longer than 6 characters and with embedded underscores.

1. **Subroutine names and mixed precision inputs.** The unmodified subroutine name has a character (S, D, C or Z) that specifies the floating point argument types. This will be the type of the output argument. By applying the rules in Section 4.3.2, this also determines the types of the scalar arguments. The possible types of the remaining array arguments are listed in Section 4.3.2. The types of these arguments (written _S, _D, _C or _Z) are appended to the unmodified subroutine name, in the order in which they appear in the argument list.

   For example, consider BLAS_ZGEMM(α,A,B,β,C) (only the floating point arguments are shown). The Z in BLAS_ZGEMM means that C, α and β are all double-complex. The possible types of A and B, and the corresponding subroutine names, are:

Type of A	Type of B	Modified subroutine name
C	C	BLAS_ZGEMM_C_C
C	Z	BLAS_ZGEMM_C_Z
Z	C	BLAS_ZGEMM_Z_C
D	D	BLAS_ZGEMM_D_D
D	Z	BLAS_ZGEMM_D_Z
Z	D	BLAS_ZGEMM_Z_D

2. **Subroutine names and extended precision.** To accommodate extended precision, PREC is added as the last argument, and _X is appended to the end of subroutine name (which may already have been modified to accommodate mixed precision).

   For example, double-complex matrix-matrix multiplication implemented with extended precision is named BLAS_ZGEMM_X. Double-complex matrix-matrix multiplication where the A and B arguments are single-complex is named BLAS_ZGEMM_C_C_X.

3. **Type of** PREC. PREC is an integer (named constant), as defined in the include file `blas_namedconstants.h` (see section A.5).

4. **Environmental enquiry function.** BLAS_FPINFO_X(CMACH,PREC) returns an integer. PREC is as specified above. CMACH is as defined in sections 1.6, 2.7, 4.3.3, and A.5.

5. **Error Handling.** Error handling is as defined in section 2.5.6.

To shorten the subroutine specifications in section 4.5, we will abbreviate the list of possible subroutine names for GEMM to a single one: BLAS_xGEMM{_a_b}{_X} The prefix x may be S (single), D (double), C (complex) or Z (double complex). Also, the subroutine name may optionally be appended with _a_b, where a and b are the types of A and B respectively, and then optionally be appended with _X. At least one of _a_b or _X must appear.

4.4.3 Interface Issues for C

1. **Subroutine names and mixed precision inputs.** The same scheme is used as in Fortran 77, as described above, except that all characters in subroutine names are lower case.

2. **Subroutine names and extended precision.** The same scheme is used as in Fortran 77, as described above, except that all characters in subroutine names are lower case.

3. **Type of PREC.** PREC is an enumerated type, as defined in the include file `blas_enum.h` (see section A.6).

4. **Environmental enquiry function.** BLAS_fpinfo_x(CMACH,PREC) returns an integer. PREC is as specified above. CMACH is as defined in sections 1.6, 2.7, 4.3.3, and A.6.

5. **Error Handling.** Error handling is as defined in section 2.6.9.

4.5 Language Bindings

4.5.1 Overview

As in Chapter 2, each specification of a routine will correspond to an operation outlined in the functionality tables. Operations are organized analogous to the order in which they are presented in the functionality tables. The specification will have the form:

NAME (*multi-word description of operation*) $< mathematical\ representation >$

- Fortran 95 binding
- Fortran 77 binding
- C binding

Section 4.4 describes abbreviations we use below. For example,

```
SUBROUTINE BLAS_xDOT{_a_b}{_X}( N, ALPHA, X, INCX, BETA,
             Y, INCY, R [, PREC])
```

means that the subroutine name may optionally be appended with _a_b, where a and b are the types of X and Y, respectively, and also optionally appended with _X, in which case the parameter PREC must also appear.

The routines specified here are

- Reduction Operations (section 4.5.2)
  - DOT (Dot product)
  - SUM (Sum)
- Vector Operations (section 4.5.3)
  - AXPBY (Scaled vector accumulation)
  - WAXPBY (Scaled vector addition)
- Matrix-Vector Operations (section 4.5.4)
  - {GE,GB}MV (Matrix vector product)
  - {SY,SB,SP}MV (Symmetric matrix vector product)

  - {HE,HB,HP}MV (Hermitian matrix vector product)
  - {TR,TB,TP}MV (Triangular matrix vector product)
  - GE_SUM_MV (Summed matrix vector multiplies)
  - {TR,TB,TP}SV (Triangular solve)

- Matrix-Matrix Operations (section 4.5.5)
  - GEMM (General Matrix Matrix product)
  - SYMM (Symmetric matrix matrix product)
  - HEMM (Hermitian matrix matrix product)
  - TRMM (Triangular matrix matrix multiply)
  - TRSM (Triangular solve)
  - SYRK (Symmetric rank-k update)
  - HERK (Hermitian rank-k update)
  - SYR2K (Symmetric rank-2k update)
  - HER2K (Hermitian rank-2k update)

4.5.2 Mixed and Extended Precision Reduction Operations

DOT (Dot Product) $x, y \in \mathbb{R}^n, r \leftarrow \beta r + \alpha x^T y = \beta r + \alpha \sum_{i=0}^{n-1} x_i y_i$

$$x, y \in \mathbb{C}^n, r \leftarrow \beta r + \alpha x^T y = \beta r + \alpha \sum_{i=0}^{n-1} x_i y_i \text{ or } r \leftarrow \beta r + \alpha x^H y = \beta r + \alpha \sum_{i=0}^{n-1} \bar{x}_i y_i$$

The routine DOT adds the scaled dot product of two vectors x and y into a scaled scalar r. The routine returns immediately if n is less than zero, or, if beta is equal to one and either alpha or n is equal to zero. If alpha is equal to zero then x and y are not read. Similarly, if beta is equal to zero, r is not read. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

When x and y are complex vectors, the vector components x_i are used unconjugated or conjugated as specified by the operator argument conj. If x and y are real vectors, the operator argument conj has no effect.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE dot( x, y, r [, conj] [, alpha] [, beta] [, prec] )
  <type>(<wp>), INTENT (IN) :: x(:)
  <type>(<wp>), INTENT (IN) :: y(:)
  <type>(<wp>), INTENT (INOUT) :: r
  TYPE (blas_conj_type), INTENT(IN), OPTIONAL :: conj
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  x and y have shape (n)
```

The types of alpha, x, y, beta and r are governed according to the rules of mixed precision arguments set down in section 4.3: the types of x and y can optionally differ from that of r, alpha and beta.

- Fortran 77 binding:

```
 SUBROUTINE BLAS_xDOT{_a_b}{_X}( CONJ, N, ALPHA, X, INCX, BETA, Y, INCY,
$                                R, [, PREC] )
 INTEGER           CONJ, INCX, INCY, N [, PREC]
 <type>            ALPHA, BETA, R
 <type>            X( * )
 <type>            Y( * )
```

The types of ALPHA, X, Y, BETA and R are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of X and _b is the type of Y. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a_b and _X must be present.

- C binding:

```
void BLAS_xdot{_a_b}{_x}( enum blas_conj_type conj, int n, SCALAR_IN alpha,
                          const ARRAY x, int incx, SCALAR_IN beta,
                          const ARRAY y, int incy, SCALAR_INOUT r,
                          [, enum blas_prec_type prec] );
```

The types of alpha, x, y, beta and r are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of argument x and _b is the type of argument y. The suffix _x is present if and only if prec is present. One or both of the suffixes _a_b and _x must be present.

SUM (Sum) $r \leftarrow \sum_{i=0}^{n-1} x_i$

The routine SUM computes the sum of the entries of a vector x. If n is less than or equal to zero, this routine returns immediately with the output scalar r set to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

Extended precision is permitted, but not mixed precision.

This routine has the same specification as in Chapter 2, except that extended precision is permitted. Mixed precision is not permitted.

- Fortran 95 binding:

```
<type>(<wp>) FUNCTION sum( x, prec )
  <type>(<wp>), INTENT (IN) :: x(:)
  TYPE (blas_prec_type), INTENT (IN) :: prec
where
  x has shape (n)
```

The types of `sum` and `x` are identical.

- Fortran 77 binding:

```
<type> FUNCTION BLAS_xSUM_X( N, X, INCX, PREC )
INTEGER            INCX, N, PREC
<type>             X( * )
```

The types of `BLAS_xSUM_X` and argument `X` are both specified by the prefix `x`.

- C binding:

```
void BLAS_xsum_x( int n, const ARRAY x, int incx, SCALAR_INOUT sum,
                  enum blas_prec_type prec );
```

The types of arguments `sum` and `x` are both specified by the prefix `x`.

4.5.3 Mixed and Extended Precision Vector Operations

AXPBY (Scaled vector accumulation) $y \leftarrow \alpha x + \beta y$

The routine AXPBY scales the vector x by α and the vector y by β, adds these two vectors to one another and stores the result in the vector y. If n is less than or equal to zero, or if α is equal to zero and β is equal to one, this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler.

Extended and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE axpby( x, y [, alpha] [, beta] [, prec] )
  <type>(<wp>), INTENT (IN) :: x(:)
  <type>(<wp>), INTENT (INOUT) :: y(:)
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  x and y have shape (n)
```

The default value for β is 1.0 and (1.0,0.0).

The types of `x`, `y`, `alpha`, and `beta` are governed according to the rules of mixed precision arguments set down in section 4.3: the type of `x` can optionally differ from that of `alpha`, `beta` and `y`.

- Fortran 77 binding:

```
SUBROUTINE BLAS_xAXPBY{_a}{_X}( N, ALPHA, X, INCX, BETA, Y, INCY
                                [, PREC] )
INTEGER             INCX, INCY, N [, PREC]
<type>              ALPHA, BETA
<type>              X( * )
<type>              Y( * )
```

The types of `ALPHA`, `X`, `Y`, and `BETA` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a` is present then `_a` is the type of `X`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a` and `_X` must be present.

- C binding:

```
void BLAS_xaxpby{_a}{_x}( int n, SCALAR_IN alpha, const ARRAY x, int incx,
                          SCALAR_IN beta, ARRAY y, int incy,
                          [, enum blas_prec_type prec] );
```

The types of `alpha`, `x`, `y`, and `beta` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a` is present then `_a` is the type of argument `x`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a` and `_x` must be present.

WAXPBY (Scaled vector addition) $w \leftarrow \alpha x + \beta y$

The routine WAXPBY scales the vector x by α and the vector y by β, adds these two vectors to one another and stores the result in the vector w. If n is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy or incw less than zero is permitted. However, if either incx or incy or incw is equal to zero, an error flag is set and passed to the error handler.

Extended and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE waxpby( x, y, w [, alpha] [, beta] [, prec] )
  <type>(<wp>), INTENT (IN) :: x(:)
  <type>(<wp>), INTENT (IN) :: y(:)
  <type>(<wp>), INTENT (OUT) :: w(:)
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  x, y and w have shape (n)
```

The default value for β is 1.0 and (1.0,0.0).

The types of `x`, `y`, `w`, `alpha` and `beta` are governed according to the rules of mixed precision arguments set down in section 4.3: the types of `x` and `y` can optionally differ from that of `w`, `alpha` and `beta`.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xWAXPBY{_a_b}{_X}( N, ALPHA, X, INCX, BETA, Y, INCY,
     $                                  W, INCW [, PREC] )
      INTEGER            INCW, INCX, INCY, N [, PREC]
      <type>             ALPHA, BETA
      <type>             W( * )
      <type>             X( * )
      <type>             Y( * )
```

The types of `X`, `Y`, `W`, `ALPHA` and `BETA` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of `X` and `_b` is the type of `Y`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a_b` and `_X` must be present.

- C binding:

```
void BLAS_xwaxpby{_a_b}{_x}( int n, SCALAR_IN alpha, const ARRAY x, int incx,
                             SCALAR_IN beta, const ARRAY y, int incy, ARRAY w,
                             int incw [, enum blas_prec_type prec] );
```

The types of `x`, `y`, `w`, `alpha` and `beta` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of argument `x` and `_b` is the type of argument `y`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a_b` and `_x` must be present.

4.5.4 Mixed and Extended Precision Matrix-Vector Operations

{GE,GB}MV (Matrix vector product) $y \leftarrow \alpha Ax + \beta y$, $y \leftarrow \alpha A^T x + \beta y$ or $y \leftarrow \alpha A^H x + \beta y$

The routines multiply a vector x by a general (or general band) matrix A or its transpose, or its conjugate transpose, scales the resulting vector and adds it to the scaled vector operand y. If m or n is less than or equal to zero or if beta is equal to one and alpha is equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. For the routine GEMV, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine GBMV, if kl or ku is less than zero, or if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

Extended and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE gbmv( a, m, kl, x, y [, trans] [, alpha] [, beta] [, prec] )
  <type>(<wp>), INTENT(IN) :: a(:,:), x(:)
```

```
      INTEGER, INTENT(IN) :: m, kl
      <type>(<wp>), INTENT(INOUT) :: y(:)
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      TYPE (blas_prec_type), INTENT(IN), OPTIONAL :: prec
    where
      if trans = blas_no_trans then
         x has shape (n)
         y has shape (m)
      else if trans =/ blas_no_trans then
         x has shape (m)
         y has shape (n)
      end if
```

The functionality of gemv is covered by gemm.

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGEMV{_a_b}{_X}( TRANS, M, N, ALPHA, A, LDA,
     $                                 X, INCX, BETA, Y, INCY [, PREC] )
General Band:
      SUBROUTINE BLAS_xGBMV{_a_b}{_X}( TRANS, M, N, KL, KU, ALPHA, A,
     $                                 LDA, X, INCX, BETA, Y, INCY [, PREC] )
all:
      INTEGER            INCX, INCY, KL, KU, LDA, M, N, [PREC,] TRANS
      <type>             ALPHA, BETA
      <type>             A( LDA, * )
      <type>             X( * )
      <type>             Y( * )
```

The types of ALPHA, A, X, Y, and BETA are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of A and _b is the type of X. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a_b and _X must be present.

- C binding:

```
General:
void BLAS_xgemv{_a_b}{_x}( enum blas_order_type order,
                           enum blas_trans_type trans, int m, int n,
                           SCALAR_IN alpha, const ARRAY a, int lda,
                           const ARRAY x, int incx, SCALAR_IN beta, ARRAY y,
                           int incy [, enum blas_prec_type prec] );
General Band:
void BLAS_xgbmv{_a_b}{_x}( enum blas_order_type order,
                           enum blas_trans_type trans, int m, int n,
                           int kl, int ku, SCALAR_IN alpha,
                           const ARRAY a, int lda, const ARRAY x, int incx,
```

```
                              SCALAR_IN beta, ARRAY y, int incy
                              [, enum blas_prec_type prec] );
```

The types of alpha, a, x, y and beta are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of argument a and _b is the type of argument x. The suffix _x is present if and only if prec is present. One or both of the suffixes _a_b and _x must be present.

{SY,SB,SP}MV (Symmetric matrix vector multiply) $y \leftarrow \alpha Ax + \beta y$ with $A = A^T$

The routines multiply a vector x by a real or complex symmetric matrix A, scales the resulting vector and adds it to the scaled vector operand y. If n is less than or equal to zero or if beta is equal to one and alpha is equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. For the routine SYMV, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine SBMV, if lda is less than k plus one, an error flag is set and passed to the error handler.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
Symmetric Band:
    SUBROUTINE sbmv( a, x, y [, uplo] [, alpha] [, beta] [, prec] )
Symmetric Packed:
    SUBROUTINE spmv( ap, x, y [, uplo] [, alpha] [, beta] [, prec] )
      <type>(<wp>), INTENT(IN) :: <aa>
      <type>(<wp>), INTENT(IN) :: x(:)
      <type>(<wp>), INTENT(INOUT) :: y(:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      TYPE (blas_prec_type), INTENT(IN), OPTIONAL :: prec
    where
      <aa> ::= a(:,:) or ap(:)
    and
      SB  a has shape (k+1,n)
      SP  ap has shape (n*(n+1)/2)
      x and y have shape (n)
```

The types of alpha, a or ap, x, beta, and y are governed by the rules of mixed precision arguments set down in section 4.3: the types of a or ap and x can optionally differ from that of y, alpha and beta.

The functionality of symv is covered by symm.

- Fortran 77 binding:

```
Symmetric:
      SUBROUTINE BLAS_xSYMV{_a_b}{_X}( UPLO, N, ALPHA, A, LDA, X, INCX,
     $                                 BETA, Y, INCY [, PREC] )
Symmetric Band:
      SUBROUTINE BLAS_xSBMV{_a_b}{_X}( UPLO, N, K, ALPHA, A, LDA, X, INCX,
     $                                 BETA, Y, INCY [, PREC] )
Symmetric Packed:
      SUBROUTINE BLAS_xSPMV{_a_b}{_X}( UPLO, N, ALPHA, AP, X, INCX, BETA,
     $                                 Y, INCY [, PREC] )
all:
      INTEGER            INCX, INCY, K, LDA, N, UPLO [, PREC]
      <type>             ALPHA, BETA
      <type>             A( LDA, * ) or AP( * )
      <type>             X( * )
      <type>             Y( * )
```

The types of ALPHA, A or AP, X, Y and BETA are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of A or AP, and _b is the type of X. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a_b and _X must be present.

- C binding:

```
Symmetric:
void BLAS_xsymv{_a_b}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                           int n, SCALAR_IN alpha, const ARRAY a, int lda,
                           const ARRAY x, int incx, SCALAR_IN beta, ARRAY y,
                           int incy [, enum blas_prec_type prec] );
Symmetric Band:
void BLAS_xsbmv{_a_b}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                           int n, int k, SCALAR_IN alpha, const ARRAY a,
                           int lda, const ARRAY x, int incx, SCALAR_IN beta,
                           ARRAY y, int incy [, enum blas_prec_type prec] );
Symmetric Packed:
void BLAS_xspmv{_a_b}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                           int n, SCALAR_IN alpha, const ARRAY ap,
                           const ARRAY x, int incx, SCALAR_IN beta, ARRAY y,
                           int incy [, enum blas_prec_type prec] );
```

The types of alpha, a or ap, x, y, and beta are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of argument a or ap and _b is the type of argument x. The suffix _x is present if and only if prec is present. One or both of the suffixes _a_b and _x must be present.

{HE,HB,HP}MV (Hermitian matrix vector product) $y \leftarrow \alpha A x + \beta y$ with $A = A^H$

The routines multiply a vector x by a Hermitian matrix A, scales the resulting vector and adds it to the scaled vector operand y. If n is less than or equal to zero or if beta is equal to one and alpha is equal to zero, this routine returns immediately. The imaginary part of the diagonal entries of the matrix operand are supposed to be zero and should not be referenced. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. For the routine HEMV, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine HBMV, if lda is less than k plus one, an error flag is set and passed to the error handler.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
Hermitian Band:
    SUBROUTINE hbmv{_a}{_x}( a, x, y  [, uplo] [, alpha] [, beta] [, prec] )
Hermitian Packed:
    SUBROUTINE hpmv{_a}{_x}( ap, x, y [, uplo] [, alpha] [, beta] [, prec] )
      COMPLEX(<wp>), INTENT(IN) :: <aa>
      COMPLEX(<wp>), INTENT(IN) :: x(:)
      COMPLEX(<wp>), INTENT(INOUT) :: y(:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      COMPLEX(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      TYPE (blas_prec_type), INTENT(IN), OPTIONAL :: prec
    where
      <aa>  ::= a(:,:) or ap(:)
    and
      HB  a has shape (k+1,n)
      HP  ap has shape (n*(n+1)/2)
      x and y have shape (n)
```

The types of alpha, a or ap, x, beta, and y are governed by the rules of mixed precision arguments set down in section 4.3: the types of a or ap and x can optionally differ from that of y, alpha and beta.

The functionality of hemv is covered by hemm.

- Fortran 77 binding:

```
Hermitian:
      SUBROUTINE BLAS_xHEMV{_a_b}{_X}( UPLO, N, ALPHA, A, LDA, X, INCX,
     $                                 BETA, Y, INCY [, PREC] )
Hermitian Band:
      SUBROUTINE BLAS_xHBMV{_a_b}{_X}( UPLO, N, K, ALPHA, A, LDA, X, INCX,
     $                                 BETA, Y, INCY [, PREC] )
Hermitian Packed:
      SUBROUTINE BLAS_xHPMV{_a_b}{_X}( UPLO, N, ALPHA, AP, X, INCX,
     $                                 BETA, Y, INCY [, PREC] )
all:
```

```
INTEGER          INCX, INCY, K, LDA, N, UPLO [, PREC]
<ctype>          ALPHA, BETA
<ctype>          A( LDA, * ) or AP( * )
<ctype>          X( * )
<ctype>          Y( * )
```

The types of `ALPHA`, `A` or `AP`, `X`, `Y`, and `BETA` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of `A` or `AP` and `_b` is the type of `X`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a_b` and `_X` must be present.

- C binding:

```
Hermitian:
void BLAS_xhemv{_a_b}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                           int n, CSCALAR_IN alpha, const CARRAY a, int lda,
                           const CARRAY x, int incx, CSCALAR_IN beta, CARRAY y,
                           int incy [, enum blas_prec_type prec] );
Hermitian Band:
void BLAS_xhbmv{_a_b}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                           int n, int k, CSCALAR_IN alpha, const CARRAY a,
                           int lda, const CARRAY x, int incx, CSCALAR_IN beta,
                           CARRAY y, int incy [, enum blas_prec_type prec] );
Hermitian Packed:
void BLAS_xhpmv{_a_b}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                           int n, CSCALAR_IN alpha, const CARRAY ap,
                           const CARRAY x, int incx, CSCALAR_IN beta, CARRAY y,
                           int incy [, enum blas_prec_type prec] );
```

The types of `alpha`, `a` or `ap`, `x`, `y`, and `beta` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of argument `a` or `ap` and `_b` is the type of argument `x`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a_b` and `_x` must be present.

{TR,TB,TP}MV (Triangular matrix vector product) $x \leftarrow \alpha T x$, $x \leftarrow \alpha T^T x$ or $x \leftarrow \alpha T^H x$

The routines multiply a vector x by a general triangular matrix T or its transpose, or its conjugate transpose, and copies the resulting vector in the vector operand x. If n is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler. For the routine TRMV, if ldt is less than one or ldt is less than n, an error flag is set and passed to the error handler. For the routine TBMV, if ldt is less than k plus one, an error flag is set and passed to the error handler.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
Triangular Band:
    SUBROUTINE tbmv( t, x  [, uplo] [, transt] [, diag] [, alpha] [, prec] )
Triangular Packed:
    SUBROUTINE tpmv( tp, x [, uplo] [, transt] [, diag] [, alpha] [, prec] )
      <type>(<wp>), INTENT(IN) :: <tt>
      <type>(<wp>), INTENT(INOUT) :: x(:)
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
      TYPE (blas_diag_type), INTENT(IN), OPTIONAL :: diag
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transt
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE (blas_prec_type), INTENT(IN), OPTIONAL :: prec
    where
      <tt>  ::= t(:,:) or tp(:)
    and
      x has shape (n)
      TB  t has shape (k+1,n)
      TP  tp has shape (n*(n+1)/2)
    (k=band width)
```

The types of alpha, t or tp, and x are governed by the rules of mixed precision arguments set down in section 4.3: the type of t or tp can optionally differ from that of x and alpha.

The functionality of trmv is covered by trmm.

- Fortran 77 binding:

```
Triangular:
      SUBROUTINE BLAS_xTRMV{_a}{_X}( UPLO, TRANS, DIAG, N, ALPHA, T, LDT, X,
     $                              INCX [, PREC] )
Triangular Band:
      SUBROUTINE BLAS_xTBMV{_a}{_X}( UPLO, TRANS, DIAG, N, K, ALPHA, T, LDT,
     $                              X, INCX [, PREC] )
Triangular Packed:
      SUBROUTINE BLAS_xTPMV{_a}{_X}( UPLO, TRANS, DIAG, N, ALPHA, TP, X, INCX
     $                              [, PREC] )
all:
      INTEGER            DIAG, INCX, K, LDT, N, TRANS, UPLO [, PREC]
      <type>             ALPHA
      <type>             T( LDT, * ) or TP( * )
      <type>             X( * )
```

The types of ALPHA, T or TP, and X are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of T or TP. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a and _X must be present.

- C binding:

```
Triangular:
void BLAS_xtrmv{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, enum blas_diag_type diag,
                         int n, SCALAR_IN alpha, const ARRAY t, int ldt,
                         ARRAY x, int incx [, enum blas_prec_type prec] );
Triangular Band:
void BLAS_xtbmv{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, enum blas_diag_type diag,
                         int n, int k, SCALAR_IN alpha, const ARRAY t, int ldt,
                         ARRAY x, int incx [, enum blas_prec_type prec] );
Triangular Packed:
void BLAS_xtpmv{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, enum blas_diag_type diag,
                         int n, SCALAR_IN alpha, const ARRAY tp,
                         ARRAY x, int incx [, enum blas_prec_type prec] );
```

The types of `alpha`, `t` or `tp`, and `x` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a` is present then `_a` is the type of argument `t` or `tp`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a` and `_x` must be present.

GE_SUM_MV (Summed matrix vector multiplies) $y \leftarrow \alpha Ax + \beta Bx$

This routine adds the product of two scaled matrix vector products. It can be used to compute the residual of an approximate eigenvector and eigenvalue of the generalized eigenvalue problem $A * x = \lambda * B * x$. If m or n is less than or equal to zero or if beta is equal to one and alpha is equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or lda is less than m, or ldb is less than one or ldb is less than m, an error flag is set and passed to the error handler.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE ge_sum_mv( a, x, b, y [, alpha] [, beta] [, prec])
  <type>(<wp>), INTENT (IN) :: a(:,:), b(:,:)
  <type>(<wp>), INTENT (IN) :: x(:)
  <type>(<wp>), INTENT (OUT) :: y(:)
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
  <type>(blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  x has shape (n);
  y has shape (m);
  a and b have shape (m,n) for general matrices
```

The types of `alpha`, `a`, `x`, `beta`, `b`, and `y` are governed according to the rules of mixed precision arguments set down in section 4.3: the types of `a` and `b` can optionally differ from that of `x`, `y`, `alpha` and `beta`. Arguments `a` and `b` must have the same type.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xGE_SUM_MV{_a_b}{_X}( M, N, ALPHA, A, LDA, X, INCX,
     $                                     BETA, B, LDB, Y, INCY
     $                                     [, PREC] )
      INTEGER            INCX, INCY, LDA, LDB, M, N [, PREC]
      <type>             ALPHA, BETA
      <type>             A( LDA, * ), B( LDB, * )
      <type>             X( * )
      <type>             Y( * )
```

The types of `ALPHA`, `A`, `X`, `BETA`, `B`, and `Y` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a` is present then `_a` is the type of `A` and `B`, and `_b` is the type of `x`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a_b` and `_X` must be present.

- C binding:

```
void BLAS_xge_sum_mv{_a_b}{_x}( enum blas_order_type order, int m, int n,
                                SCALAR_IN alpha, const ARRAY a, int lda,
                                const ARRAY x, int incx, SCALAR_IN beta,
                                const ARRAY B, int ldb, ARRAY y, int incy
                                [, enum blas_prec_type prec] );
```

The types of `alpha`, `a`, `x`, `beta`, `b`, and `y` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of `a` and `b`, and `_b` is the type of `x`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a_b` and `_x` must be present.

{TR,TB,TP}SV (Triangular solve) $x \leftarrow \alpha T^{-1} x$, $x \leftarrow \alpha T^{-T} x$

These functions solve one of the systems of equations $x \leftarrow \alpha T^{-1} x$ or $y \leftarrow \alpha T^{-1} x$, where x and y are vectors and the matrix T is a unit, non-unit, upper or lower triangular (or triangular banded or triangular packed) matrix. If n is less than or equal to zero, this function returns immediately. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler. If ldt is less than one or ldt is less than n, an error flag is set and passed to the error handler.

Extended precision and mixed precision are permitted.

Advice to implementors. Note that no check for singularity, or near singularity is specified for these triangular equation-solving functions. The requirements for such a test depend on the application, and so we felt that this should not be included, but should instead be performed before calling the triangular solver.

To implement this function when the internal precision requested is higher than the precision of x, temporary workspace is needed to compute and store x internally to higher precision. (*End of advice to implementors.*)

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
Triangular Band:
    SUBROUTINE tbsv( t, x [, uplo] [, transt] [, diag] [, alpha] [, prec] )
Triangular Packed:
    SUBROUTINE tpsv( tp, x [, uplo] [, trans] [, diag] [, alpha] [, prec] )
      <type>(<wp>), INTENT(IN) :: <tt>
      <type>(<wp>), INTENT(INOUT) :: x(:)
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
      TYPE (blas_diag_type), INTENT(IN), OPTIONAL :: diag
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
      TYPE (blas_prec_type), INTENT(IN), OPTIONAL :: prec
    where
      <tt>  ::= t(:,:) or tp(:)
    and
      x has shape (n)
      TB  t has shape (k+1,n)
      TP  tp has shape (n*(n+1)/2)
    (k=band width)
```

The types of `alpha`, `t` or `tp`, and `x` are governed by the rules of mixed precision arguments set down in section 4.3: the type of `t` or `tp` can optionally differ from that of `x` and `alpha`.

The functionality of trsv is covered by trsm.

- Fortran 77 binding:

```
Triangular:
       SUBROUTINE BLAS_xTRSV{_a}{_X}( UPLO, TRANS, DIAG, N, ALPHA, T, LDT,
      $                               X, INCX [, PREC] )
Triangular Band:
       SUBROUTINE BLAS_xTBSV{_a}{_X}( UPLO, TRANS, DIAG, N, K, ALPHA, T,
      $                               LDT, X, INCX [, PREC] )
Triangular Packed:
       SUBROUTINE BLAS_xTPSV{_a}{_X}( UPLO, TRANS, DIAG, N, ALPHA, TP, X,
      $                               INCX [, PREC] )
all:
      INTEGER            DIAG, INCX, K, LDT, N, TRANS, UPLO [, PREC]
      <type>             ALPHA
      <type>             T( LDT, * ) or TP( * )
      <type>             X( * )
```

The types of `ALPHA`, `T` or `TP`, and `X` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a` is present then `_a` is the type of `T` or `TP`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a` and `_X` must be present.

- C binding:

```
Triangular:
void BLAS_xtrsv{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, enum blas_diag_type diag,
                         int n, SCALAR_IN alpha, const ARRAY t, int ldt,
                         ARRAY x, int incx [, enum blas_prec_type prec] );
Triangular Band:
void BLAS_xtbsv{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, enum blas_diag_type diag,
                         int n, int k, SCALAR_IN alpha, const ARRAY t, int ldt,
                         ARRAY x, int incx [, enum blas_prec_type prec] );
Triangular Packed:
void BLAS_xtpsv{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, enum blas_diag_type diag,
                         int n, SCALAR_IN alpha, const ARRAY tp, ARRAY x,
                         int incx [, enum blas_prec_type prec] );
```

The types of `alpha`, `t` or `tp`, and `x` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a` is present then `_a` is the type of argument `t` or `tp`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a` and `_x` must be present.

4.5.5 Mixed and Extended Precision Matrix-Matrix Operations

In the following section, $op(X)$ denotes X, or X^T or X^H where X is a matrix.

GEMM (General Matrix Matrix Product) $C \leftarrow \alpha op(A)op(B) + \beta C$

The routine performs a general matrix matrix multiply $C \leftarrow \alpha op(A)op(B) + \beta C$ where α and β are scalars, and A, B, and C are general matrices. This routine returns immediately if m or n or k is less than or equal to zero. If lda is less than one or less than m, or if ldb is less than one or less than k, or if ldc is less than one or less than m, an error flag is set and passed to the error handler.

This interface encompasses the Legacy BLAS routine xGEMM.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE gemm( a, b, c [, transa] [, transb] [, alpha] [, beta] &
                 [, prec] )
  <type>(<wp>), INTENT(IN) :: <aa>
  <type>(<wp>), INTENT(IN) :: <bb>
  <type>(<wp>), INTENT(INOUT) :: <cc>
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transa, transb
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
```

```
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  <aa>  ::= a(:,:) or a(:)
  <bb>  ::= b(:,:) or b(:)
  <cc>  ::= c(:,:) or c(:)
and
  c, rank 2, has shape (m,n)
       a has shape (m,k) if transa = blas_no_trans (the default)
                   (k,m) if transa /= blas_no_trans
                   (m) if rank 1
       b has shape (k,n) if transb = blas_no_trans (the default)
                   (n,k) if transb /= blas_no_trans
                   (n) if rank 1
  c, rank 1, has shape (m)
       a has shape (m,n) if transa = blas_no_trans (the default)
                   (n,m) if transa /= blas_no_trans
       b has shape (n)
```

Rank `a`	Rank `b`	Rank `c`	`transa`	`transb`	Operation	Arguments
2	2	2	N	N	$C \leftarrow \alpha AB + \beta C$	`real or complex`
2	2	2	N	T	$C \leftarrow \alpha AB^T + \beta C$	`real or complex`
2	2	2	N	H	$C \leftarrow \alpha AB^H + \beta C$	`complex`
2	2	2	T	N	$C \leftarrow \alpha A^T B + \beta C$	`real or complex`
2	2	2	T	T	$C \leftarrow \alpha A^T B + \beta C$	`real or complex`
2	2	2	H	N	$C \leftarrow \alpha A^H B + \beta C$	`complex`
2	2	2	H	H	$C \leftarrow \alpha A^H B^H + \beta C$	`complex`
2	1	1	N	-	$c \leftarrow \alpha Ab + \beta c$	`real or complex`
2	1	1	T	-	$c \leftarrow \alpha A^T b + \beta c$	`real or complex`
2	1	1	H	-	$c \leftarrow \alpha A^H b + \beta c$	`complex`
1	1	2	-	-	$C \leftarrow \alpha ab^T + \beta C$	`real or complex`
1	1	2	-	H	$C \leftarrow \alpha ab^H + \beta C$	`complex`

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The functionality of xGEMV is also covered by this generic procedure.

The types of `a`, `b`, `c`, `alpha` and `beta` are governed according to the rules of mixed precision arguments set down in section 4.3: the types of `a` and `b` can optionally differ from that of `c`, `alpha` and `beta`.

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGEMM{_a_b}{_X}( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA,
     $                                 B, LDB, BETA, C, LDC [, PREC] )
      INTEGER            K, LDA, LDB, LDC, M, N, TRANSA, TRANSB [, PREC]
      <type>             ALPHA, BETA
      <type>             A( LDA, * )
```

```
    <type>             B( LDB, * )
    <type>             C( LDC, * )
```

The types of `ALPHA`, `A`, `B`, `BETA` and `C` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of `A` and `_b` is the type of `B`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a_b` and `_X` must be present.

- C binding:

```
void BLAS_xgemm{_a_b}{_x}( enum blas_order_type order,
                           enum blas_trans_type transa,
                           enum blas_trans_type transb, int m, int n, int k,
                           SCALAR_IN alpha, const ARRAY a, int lda,
                           const ARRAY b, int ldb,
                           SCALAR_IN beta, ARRAY c, int ldc
                           [, enum blas_prec_type prec] );
```

The types of `alpha`, `a`, `b`, `beta` and `c` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of argument `a` and `_b` is the type of argument `b`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a_b` and `_x` must be present.

SYMM (Symmetric Matrix Matrix Product) $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$

This routine performs one of the symmetric matrix matrix operations $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$ where α and β are scalars, A is a symmetric matrix, and B and C are general matrices. This routine returns immediately if m or n is less than or equal to zero. For side equal to blas_left_side, and if lda is less than one or less than m, or if ldb is less than one or less than m, or if ldc is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if lda is less than one or less than n, or if ldb is less than one or less than n, or if ldc is less than one or less than n, an error flag is set and passed to the error handler.

The interfaces encompass the Legacy BLAS routine xSYMM with added functionality for complex symmetric matrices.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE symm( a, b, c [, side] [, uplo] [, alpha] [, beta] [, prec] )
  <type>(<wp>), INTENT(IN) :: a(:,:)
  <type>(<wp>), INTENT(IN) :: <bb>
  <type>(<wp>), INTENT(INOUT) :: <cc>
  TYPE (blas_side_type), INTENT(IN), OPTIONAL :: side
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
```

```
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  <bb>  ::= b(:,:) or b(:)
  <cc>  ::= c(:,:) or c(:)
and
 c, rank 2, has shape (m,n), b same shape as c
   SY  a has shape (m,m) if side = blas_left_side (the default)
       a has shape (n,n) if side /= blas_left_side
 c, rank 1, has shape (m), b same shape as c
   SY  a has shape (m,m)
```

Rank b	Rank c	side	Operation
2	2	L	$C \leftarrow \alpha AB + \beta C$
2	2	R	$C \leftarrow \alpha BA + \beta C$
1	1	-	$c \leftarrow \alpha Ab + \beta c$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The functionality of xSYMV is covered by `symm`.

The types of `a`, `b`, `c`, `alpha` and `beta` are governed according to the rules of mixed precision arguments set down in section 4.3: the types of `a` and `b` can optionally differ from that of `c`, `alpha` and `beta`.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xSYMM{_a_b}{_X}( SIDE, UPLO, M, N, ALPHA, A, LDA,
     $                                 B, LDB, BETA, C, LDC [, PREC] )
      INTEGER            LDA, LDB, LDC, M, N, SIDE, UPLO [, PREC]
      <type>             ALPHA, BETA
      <type>             A( LDA, * )
      <type>             B( LDB, * )
      <type>             C( LDC, * )
```

The types of `ALPHA`, `A`, `B`, `BETA` and `C` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of `A` and `_b` is the type of `B`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a_b` and `_X` must be present.

- C binding:

```
void BLAS_xsymm{_a_b}{_x}( enum blas_order_type order,
                           enum blas_side_type side,
                           enum blas_uplo_type uplo, int m, int n,
                           SCALAR_IN alpha, const ARRAY a, int lda,
                           const ARRAY b, int ldb, SCALAR_IN beta, ARRAY c,
                           int ldc [, enum blas_prec_type prec] );
```

The types of `alpha`, `a`, `b`, `beta` and `c` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of argument `a`, and `_b` is the type of argument `b`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a_b` and `_x` must be present.

HEMM (Hermitian Matrix Matrix Product) $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$

This routine performs one of the Hermitian matrix matrix operations $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$ where α and β are scalars, A is a Hermitian matrix, and B and C are general matrices. This routine returns immediately if m or n is less than or equal to zero. For side equal to blas_left_side, and if lda is less than one or less than m, or if ldb is less than one or less than m, or if ldc is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if lda is less than one or less than n, or if ldb is less than one or less than n, or if ldc is less than one or less than n, an error flag is set and passed to the error handler.

The interfaces encompass the Legacy BLAS routine xHEMM.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
Hermitian:
    SUBROUTINE hemm( a, b, c [, side] [, uplo] [, alpha] [, beta] [, prec] )
      COMPLEX(<wp>), INTENT(IN) :: a(:,:)
      COMPLEX(<wp>), INTENT(IN) :: <bb>
      COMPLEX(<wp>), INTENT(INOUT) :: <cc>
      TYPE (blas_side_type), INTENT(IN), OPTIONAL :: side
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      COMPLEX(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
    where
      <bb>  ::= b(:,:) or b(:)
      <cc>  ::= c(:,:) or c(:)
    and
     c, rank 2, has shape (m,n), b same shape as c
       HE  a has shape (m,m) if "side" = blas_left_side (the default)
           a has shape (n,n) if "side" /= blas_left_side
     c, rank 1, has shape (m), b same shape as c
       HE  a has shape (m,m)
```

Rank b	Rank c	side	Operation
2	2	L	$C \leftarrow \alpha AB + \beta C$
2	2	R	$C \leftarrow \alpha BA + \beta C$
1	1	-	$c \leftarrow \alpha Ab + \beta c$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The functionality of xHEMV is covered by hemm.

The types of a, b, c, alpha and beta are governed according to the rules of mixed precision arguments set down in section 4.3: the types of a and b can optionally differ from that of c, alpha and beta.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xHEMM{_a_b}{_X}( SIDE, UPLO, M, N, ALPHA, A, LDA,
     $                                 B, LDB, BETA, C, LDC [, PREC] )
      INTEGER            LDA, LDB, LDC, M, N, SIDE, UPLO [, PREC]
      <ctype>            ALPHA, BETA
      <ctype>            A( LDA, * )
      <ctype>            B( LDB, * )
      <ctype>            C( LDC, * )
```

The types of ALPHA, A, B, BETA and C are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of A and _b is the type of B. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a_b and _X must be present.

- C binding:

```
void BLAS_xhemm{_a_b}{_x}( enum blas_order_type order,
                           enum blas_side_type side,
                           enum blas_uplo_type uplo, int m, int n,
                           CSCALAR_IN alpha, const CARRAY a, int lda,
                           const CARRAY b, int ldb, CSCALAR_IN beta, CARRAY c,
                           int ldc [, enum blas_prec_type prec] );
```

The types of alpha, a, b, beta and c are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of argument a, and _b is the type of argument b. The suffix _x is present if and only if prec is present. One or both of the suffixes _a_b and _x must be present.

TRMM (Triangular Matrix Matrix Multiply) $B \leftarrow \alpha op(T)B$ or $B \leftarrow \alpha Bop(T)$

These routines perform one of the matrix-matrix operations $B \leftarrow \alpha op(T)B$ or $B \leftarrow \alpha Bop(T)$ where α is a scalar, B is a general matrix, and T is a unit, or non-unit, upper or lower triangular matrix. This routine returns immediately if m or n is less than or equal to zero. For side equal to blas_left_side, and if ldt is less than one or less than m, or if ldb is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if ldt is less than one or less than n, or if ldb is less than one or less than m, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xTRMM.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE trmm( t, b [, side] [, uplo] [, transt] [, diag] &
                  [, alpha] [, prec] )
  <type>(<wp>), INTENT(IN) :: t(:,:)
  <type>(<wp>), INTENT(INOUT) :: <bb>
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
  TYPE (blas_diag_type), INTENT(IN), OPTIONAL :: diag
  TYPE (blas_side_type), INTENT(IN), OPTIONAL :: side
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transt
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  <bb>  ::= b(:,:) or b(:)
and
 b, rank 2, has shape (m,n)
   TR  t has shape (m,m) if side = blas_left_side (the default)
       t has shape (n,n) if side /= blas_left_side
 b, rank 1, has shape (m)
   TR  t has shape (m,m)
```

Rank `b`	`transa`	`side`	Operation
2	N	L	$B \leftarrow \alpha TB$
2	T	L	$B \leftarrow \alpha T^T B$
2	H	L	$B \leftarrow \alpha T^H B$
2	N	R	$B \leftarrow \alpha BT$
2	T	R	$B \leftarrow \alpha BT^T$
2	H	R	$B \leftarrow \alpha BT^H$
1	N	-	$b \leftarrow \alpha Tb$
1	T	-	$b \leftarrow \alpha T^T b$
1	H	-	$b \leftarrow \alpha T^H b$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The functionality of xTRMV is covered by `trmm`.

The types of `alpha`, `t`, and `b` are governed according to the rules of mixed precision arguments set down in section 4.3: the type of `t` can optionally differ from that of `b` and `alpha`.

- Fortran 77 binding:

```
  SUBROUTINE BLAS_xTRMM{_a}{_X}( SIDE, UPLO, TRANST, DIAG, M, N,
 $                               ALPHA, T, LDT, B, LDB [, PREC] )
  INTEGER            DIAG, LDT, LDB, M, N, SIDE, TRANST, UPLO
 $                   [, PREC]
  <type>             ALPHA
  <type>             T( LDT, * )
  <type>             B( LDB, * )
```

The types of ALPHA, T, and B are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of T. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a and _X must be present.

- C binding:

```
void BLAS_xtrmm{_a}{_x}(enum blas_order_type order, enum blas_side_type side,
                        enum blas_uplo_type uplo, enum blas_trans_type transa,
                        enum blas_diag_type diag, int m, int n,
                        SCALAR_IN alpha, const ARRAY t, int ldt, ARRAY b,
                        int ldb [, enum blas_prec_type prec] );
```

The types of alpha, t, and b are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of argument t. The suffix _x is present if and only if prec is present. One or both of the suffixes _a and _x must be present.

TRSM (Triangular Solve) $B \leftarrow \alpha op(T^{-1})B$ or $B \leftarrow \alpha B op(T^{-1})$

This routine solves one of the matrix equations $B \leftarrow \alpha op(T^{-1})B$ or $B \leftarrow \alpha B op(T^{-1})$ where α is a scalar, B is a general matrix, and T is a unit, or non-unit, upper or lower triangular matrix. This routine returns immediately if m or n is less than or equal to zero. For side equal to blas_left_side, and if ldt is less than one or less than m, or if ldb is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if ldt is less than one or less than n, or if ldb is less than one or less than m, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xTRSM.

Extended precision and mixed precision are permitted.

Advice to implementors. Note that no check for singularity, or near singularity is specified for these triangular equation-solving functions. The requirements for such a test depend on the application, and so we felt that this should not be included, but should instead be performed before calling the triangular solver.

To implement this function when the internal precision requested is higher than the precision of B, temporary workspace is needed to compute and store B internally to higher precision. (*End of advice to implementors.*)

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE trsm( t, b [, side] [, uplo] [, transt] [, diag] &
                [, alpha] [, prec] )
  <type>(<wp>), INTENT(IN) :: t(:,:)
  <type>(<wp>), INTENT(INOUT) :: <bb>
  TYPE (blas_side_type), INTENT(IN), OPTIONAL :: side
```

```
      TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transt
      TYPE (blas_diag_type), INTENT(IN), OPTIONAL :: diag
      <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
      TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
    where
      <bb>  ::= b(:,:) or b(:)
    and
     b, rank 2, has shape (m,n)
       TR  t has shape (m,m) if side = blas_left_side (the default)
           t has shape (n,n) if side /= blas_left_side
     b, rank 1, has shape (m)
       TR  t has shape (m,m)
```

Rank b	transa	side	Operation
2	N	L	$B \leftarrow \alpha T^{-1} B$
2	T	L	$B \leftarrow \alpha T^{-T} B$
2	H	L	$B \leftarrow \alpha T^{-H} B$
2	N	R	$B \leftarrow \alpha B T^{-1}$
2	T	R	$B \leftarrow \alpha B T^{-T}$
2	H	R	$B \leftarrow \alpha B T^{-H}$
1	N	-	$b \leftarrow \alpha T^{-1} b$
1	T	-	$b \leftarrow \alpha T^{-T} b$
1	H	-	$b \leftarrow \alpha T^{-H} b$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The functionality of xTRSV is covered by `trsm`.

The types of `t`, `x` and `alpha` are governed according to the rules of mixed precision arguments set down in section 4.3: the type of `t` can optionally differ from that of `x` and `alpha`.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xTRSM{_a}{_X}( SIDE, UPLO, TRANST, DIAG, M, N,
     $                               ALPHA, T, LDT, B, LDB [, PREC] )
      INTEGER            DIAG, LDT, LDB, M, N, SIDE, TRANST, UPLO
     $                   [, PREC]
      <type>             ALPHA
      <type>             T( LDT, * )
      <type>             B( LDB, * )
```

The types of `ALPHA`, `T`, and `B` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a` is present then `_a` is the type of `T`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a` and `_X` must be present.

- C binding:

```
void BLAS_xtrsm{_a}{_x}( enum blas_order_type order, enum blas_side_type side,
                         enum blas_uplo_type uplo, enum blas_trans_type transt,
                         enum blas_diag_type diag, int m, int n,
                         SCALAR_IN alpha, const ARRAY t, int ldt, ARRAY b,
                         int ldb [, enum blas_prec_type prec] );
```

The types of `alpha`, `t`, and `b` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if _a is present then _a is the type of argument `t`. The suffix _x is present if and only if `prec` is present. One or both of the suffixes _a and _x must be present.

SYRK (Symmetric Rank K update) $C \leftarrow \alpha AA^T + \beta C$, $C \leftarrow \alpha A^T A + \beta C$

This routine performs one of the symmetric rank k operations $C \leftarrow \alpha AA^T + \beta C$ or $C \leftarrow \alpha A^T A + \beta C$ where α and β are scalars, C is a symmetric matrix, and A is a general matrix. This routine returns immediately if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xSYRK with added functionality for complex symmetric matrices.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE syrk( a, c [, uplo] [, trans] [, alpha] [, beta] &
                 [, prec] )
  <type>(<wp>), INTENT(IN) :: <aa>
  <type>(<wp>), INTENT(INOUT) :: c(:,:)
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  <aa>  ::= a(:,:) or a(:)
and
 c has shape (n,n)
 a has shape (n,k) if trans = blas_no_trans (the default)
             (k,n) if trans /= blas_no_trans
             (n) if rank 1
```

Rank a	trans	Operation
2	N	$C \leftarrow \alpha AA^T + \beta C$
2	T	$C \leftarrow \alpha A^T A + \beta C$
1	-	$C \leftarrow \alpha aa^T + \beta C$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The types of alpha, a, beta and c are governed according to the rules of mixed precision arguments set down in section 4.3: the type of a can optionally differ from those of c, alpha and beta.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xSYRK{_a}{_X}( UPLO, TRANS, N, K, ALPHA, A, LDA, BETA,
     $                               C, LDC [, PREC] )
      INTEGER            K, LDA, LDC, N, TRANS, UPLO [, PREC]
      <type>             ALPHA, BETA
      <type>             A( LDA, * )
      <type>             C( LDC, * )
```

The types of ALPHA, A, BETA and C are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of A. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a and _X must be present.

- C binding:

```
void BLAS_xsyrk{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, int n, int k,
                         SCALAR_IN alpha, const ARRAY a, int lda,
                         SCALAR_IN beta, ARRAY c, int ldc
                         [, enum blas_prec_type prec] );
```

The types of alpha, a, beta and c are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of argument a. The suffix _x is present if and only if prec is present. One or both of the suffixes _a and _x must be present.

HERK (Hermitian Rank K update) $C \leftarrow \alpha AA^H + \beta C$, $C \leftarrow \alpha A^H A + \beta C$

This routine performs one of the Hermitian rank k operations $C \leftarrow \alpha AA^H + \beta C$ or $C \leftarrow \alpha A^H A + \beta C$ where α and β are scalars, C is a Hermitian matrix, and A is a general matrix. This routine returns immediately if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xHERK.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE herk( a, c [, uplo] [, trans] [, alpha] [, beta] &
                [, prec] )
  COMPLEX(<wp>), INTENT(IN) :: <aa>
  COMPLEX(<wp>), INTENT(INOUT) :: c(:,:)
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
  REAL(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  <aa>  ::= a(:,:) or a(:)
and
 c has shape (n,n)
 a has shape (n,k) if trans = blas_no_trans (the default)
             (k,n) if trans /= blas_no_trans
             (n) if rank 1
```

Rank a	trans	Operation
2	N	$C \leftarrow \alpha AA^H + \beta C$
2	T	$C \leftarrow \alpha A^H A + \beta C$
1	-	$C \leftarrow \alpha aa^H + \beta C$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The types of alpha, a, beta and c are governed according to the rules of mixed precision arguments set down in section 4.3: the type of a can optionally differ from those of c, alpha and beta.

- Fortran 77 binding:

```
 SUBROUTINE BLAS_xHERK{_a}{_X}( UPLO, TRANS, N, K, ALPHA, A, LDA, BETA,
$                               C, LDC [, PREC] )
 INTEGER            K, LDA, LDC, N, TRANS, UPLO [, PREC]
 <rtype>            ALPHA, BETA
 <ctype>            A( LDA, * )
 <ctype>            C( LDC, * )
```

The types of ALPHA, A, BETA and C are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of A. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a and _X must be present.

- C binding:

```
void BLAS_xherk{_a}{_x}( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_trans_type trans, int n, int k,
                         RSCALAR_IN alpha, const CARRAY a, int lda,
                         RSCALAR_IN beta, CARRAY c, int ldc
                         [, enum blas_prec_type prec] );
```

The types of alpha, a, beta and c are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a is present then _a is the type of argument a. The suffix _x is present if and only if prec is present. One or both of the suffixes _a and _x must be present.

SYR2K (Symmetric rank 2k update) $C \leftarrow (\alpha A)B^T + B(\alpha A)^T + \beta C$

$C \leftarrow (\alpha A)^T B + B^T(\alpha A) + \beta C$

These routines perform the symmetric rank 2k operation $C \leftarrow (\alpha A)B^T + B(\alpha A)^T + \beta C$ or $C \leftarrow (\alpha A)^T B + B^T(\alpha A) + \beta C$ where α and β are scalars, C is a symmetric matrix, and A and B are general matrices. This routine returns immediately if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, or if ldb is less than one or less than k, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xSYR2K with added functionality for complex symmetric matrices.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE syr2k( a, b, c [, uplo] [, trans] [, alpha] [, beta]
                  [, prec] )
  <type>(<wp>), INTENT(IN) :: <aa>
  <type>(<wp>), INTENT(IN) :: <bb>
  <type>(<wp>), INTENT(INOUT) :: c(:,:)
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha, beta
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  <aa>  ::= a(:,:) or a(:)
  <bb>  ::= b(:,:) or b(:)
and
 c has shape (n,n)
 if trans = blas_no_trans (the default)
    a has shape (n,k)
    b has shape (n,k)
 if trans /= blas_no_trans
    a has shape (k,n)
    b has shape (k,n)
```

Rank a	Rank b	trans	Operation
2	2	N	$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$
2	2	T	$C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$
1	1	-	$C \leftarrow \alpha ab^T + \alpha ba^T + \beta C$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The types of `alpha`, `a`, `b`, `beta` and `c` are governed according to the rules of mixed precision arguments set down in section 4.3: the types of `a` and `b` can optionally differ from those of `c`, `alpha` and `beta`.

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xSYR2K{_a_b}{_X}( UPLO, TRANS, N, K, ALPHA, A, LDA,
     $                                  B, LDB, BETA, C, LDC [, PREC] )
      INTEGER            K, LDA, LDB, LDC, N, TRANS, UPLO [, PREC]
      <type>             ALPHA, BETA
      <type>             A( LDA, * )
      <type>             B( LDB, * )
      <type>             C( LDC, * )
```

The types of `ALPHA`, `A`, `B`, `BETA` and `C` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of `A` and `_b` is the type of `B`. The suffix `_X` is present if and only if `PREC` is present. One or both of the suffixes `_a_b` and `_X` must be present.

- C binding:

```
void BLAS_xsyr2k{_a_b}{_x}( enum blas_order_type order,
                            enum blas_uplo_type uplo,
                            enum blas_trans_type trans, int n, int k,
                            SCALAR_IN alpha, const ARRAY a, int lda,
                            const ARRAY b, int ldb,
                            SCALAR_IN beta, ARRAY c, int ldc
                            [, enum blas_prec_type prec] );
```

The types of `alpha`, `a`, `b`, `beta` and `c` are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix `x` is the floating point type of the arguments, but if `_a_b` is present then `_a` is the type of argument `a` and `_b` is the type of argument `b`. The suffix `_x` is present if and only if `prec` is present. One or both of the suffixes `_a_b` and `_x` must be present.

HER2K (Hermitian rank 2k update)

$$C \leftarrow (\alpha A)B^H + B(\alpha A)^H + \beta C$$
$$C \leftarrow (\alpha A)^H B + B^H(\alpha A) + \beta C$$

These routines perform the Hermitian rank 2k operation $C \leftarrow (\alpha A)B^H + B(\alpha A)^H + \beta C$ or $C \leftarrow (\alpha A)^H B + B^H(\alpha A) + \beta C$ where α and β are scalars, C is a Hermitian matrix, and A and B are general matrices. This routine returns immediately if n or k is less than or equal to zero. If ldc is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_no_trans, and if lda is less than one or less than n, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For trans equal to blas_trans, and if lda is less than one or less than k, or if ldb is less than one or less than k, an error flag is set and passed to the error handler.

These interfaces encompass the Legacy BLAS routine xHER2K.

Extended precision and mixed precision are permitted.

This routine has the same specification as in Chapter 2, except that extended precision and mixed precision are permitted.

- Fortran 95 binding:

```
SUBROUTINE her2k( a, b, c [, uplo] [, trans] [, alpha] [, beta]
                  [, prec] )
  COMPLEX(<wp>), INTENT(IN) :: <aa>
  COMPLEX(<wp>), INTENT(IN) :: <bb>
  COMPLEX(<wp>), INTENT(INOUT) :: c(:,:)
  TYPE (blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: trans
  COMPLEX(<wp>), INTENT(IN), OPTIONAL :: alpha
  REAL(<wp>), INTENT(IN), OPTIONAL :: beta
  TYPE (blas_prec_type), INTENT (IN), OPTIONAL :: prec
where
  <aa>  ::= a(:,:) or a(:)
  <bb>  ::= b(:,:) or b(:)
and
 c has shape (n,n)
 a and b have shape (n,k) if trans = blas_no_trans (the default)
                    (k,n) if trans /= blas_no_trans
                    (n) if rank 1
```

Rank a	Rank b	trans	Operation
2	2	N	$C \leftarrow \alpha AB^H + \bar{\alpha}BA^H + \beta C$
2	2	T	$C \leftarrow \alpha A^H B + \bar{\alpha}B^H A + \beta C$
1	1	-	$C \leftarrow \alpha ab^H + \bar{\alpha}ba^H + \beta C$

The table defining the operation as a function of the operator arguments is identical to Chapter 2.

The types of `alpha`, `a`, `b`, `beta` and `c` are governed according to the rules of mixed precision arguments set down in section 4.3: the types of `a` and `b` can optionally differ from those of `c`, `alpha` and `beta`.

- Fortran 77 binding:

```
  SUBROUTINE BLAS_xHER2K{_a_b}{_X}( UPLO, TRANS, N, K, ALPHA, A, LDA,
 $                                  B, LDB, BETA, C, LDC [, PREC] )
  INTEGER            K, LDA, LDB, LDC, N, TRANS, UPLO [, PREC]
  <ctype>            ALPHA
  <rtype>            BETA
  <ctype>            A( LDA, * )
  <ctype>            B( LDB, * )
  <ctype>            C( LDC, * )
```

The types of ALPHA, A, B, BETA and C are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of A and _b is the type of B. The suffix _X is present if and only if PREC is present. One or both of the suffixes _a_b and _X must be present.

- C binding:

```
void BLAS_xher2k{_a_b}{_x}( enum blas_order_type order,
                            enum blas_uplo_type uplo,
                            enum blas_trans_type trans, int n, int k,
                            CSCALAR_IN alpha, const CARRAY A, int lda,
                            const CARRAY b, int ldb,
                            RSCALAR_IN beta, CARRAY c, int ldc
                            [, enum blas_prec_type prec] );
```

The types of alpha, a, b, beta and c are governed according to the rules of mixed precision arguments set down in section 4.3. The prefix x is the floating point type of the arguments, but if _a_b is present then _a is the type of argument a and _b is the type of argument b. The suffix _x is present if and only if prec is present. One or both of the suffixes _a_b and _x must be present.

4.5.6 Environmental Enquiry

FPINFO_X (Environmental enquiry)
This routine queries for machine-specific floating point characteristics.

- Fortran 95 binding:

```
INTEGER FUNCTION fpinfo_x( cmach, prec )
  TYPE (blas_cmach_type), INTENT (IN) :: cmach
  TYPE (blas_prec_type), INTENT (IN) :: prec
```

- Fortran 77 binding:

```
INTEGER FUNCTION BLAS_FPINFO_X( cmach, prec )
INTEGER          cmach, prec
```

- C binding:

```
int BLAS_fpinfo_x( enum blas_cmach_type cmach,
                   enum blas_prec_type prec );
```

Annex A

Appendix

This appendix contains overall notation, definitions, and implementation details for the chapters of the BLAS Technical Forum Standard.

A.1 Vector Norms

There are a variety of ways to define the norm of a vector, in particular for vectors of complex numbers, several of which have been used in the existing Level 1 BLAS and in various LAPACK auxiliary routines. Our definitions include all of these in a systematic way.

Data Type	Name	Notation	Definition
Real	one-norm	$\|x\|_1$	$\sum_i \|x_i\|$
	two-norm	$\|x\|_2$	$\sqrt{\sum_i x_i^2}$
	infinity-norm	$\|x\|_\infty$	$\max_i \|x_i\|$
Complex	one-norm	$\|x\|_1$	$\sum_i \|x_i\|$
			$= \sum_i (Re(x_i)^2 + Im(x_i)^2)^{1/2}$
	real one-norm	$\|x\|_{1R}$	$\sum_i (\|Re(x_i)\| + \|Im(x_i)\|)$
	two-norm	$\|x\|_2$	$\sqrt{\sum_i \|x_i\|^2}$
			$= (\sum_i (Re(x_i)^2 + Im(x_i)^2))^{1/2}$
	infinity-norm	$\|x\|_\infty$	$\max_i \|x_i\|$
			$= \max_i (Re(x_i)^2 + Im(x_i)^2)^{1/2}$
	real infinity-norm	$\|x\|_{\infty R}$	$\max_i (\|Re(x_i)\| + \|Im(x_i)\|)$

Table A.1: Vector Norms

Rationale. The reason for the two extra norms of complex vectors, the real one-norm and real infinity-norm, is to avoid the expense of up to n square roots, where n is the length of the vector x. The two-norm only requires one square root, so a real version is not needed. The infinity norm only requires one square root in principle, but this would require tests and branches, making it more complicated and slower than the real infinity-norm. When x is real, the one-norm and real one-norm are identical, as are the infinity-norm and real infinity-norm. We note that the Level 1 BLAS routine ICAMAX, which finds the largest entry of a complex vector, finds the largest value of $|Re(x_i)| + |Im(x_i)|$. (*End of rationale.*)

Computing the two-norm or Frobenius-norm of a vector is equivalent. However, this is not the case for computing matrix norms. For consistency of notation between vector and matrix norms, both norms are available.

A.2 Matrix Norms

Analogously to vector norms as discussed in Section A.1, there are a variety of ways to define the norm of a matrix, in particular for matrices of complex numbers. Our definitions include all of these in a systematic way.

Data Type	Name	Notation	Definition
Real	one-norm	$\|A\|_1$	$\max_j \sum_i \|a_{ij}\|$
	Frobenius-norm	$\|A\|_F$	$\sqrt{\sum_i \sum_j a_{ij}^2}$
	infinity-norm	$\|A\|_\infty$	$\max_i \sum_j \|a_{ij}\|$
	max-norm	$\|A\|_{\max}$	$\max_i \max_j \|a_{ij}\|$
Complex	one-norm	$\|A\|_1$	$\max_j \sum_i \|a_{ij}\|$
			$= \max_j \sum_i (Re(a_{ij})^2 + Im(a_{ij})^2)^{1/2}$
	real one-norm	$\|A\|_{1R}$	$\max_j \sum_i (\|Re(a_{ij})\| + \|Im(a_{ij})\|)$
	Frobenius-norm	$\|A\|_F$	$\sqrt{\sum_i \sum_j \|a_{ij}\|^2}$
			$= (\sum_i \sum_j (Re(a_{ij})^2 + Im(a_{ij})^2))^{1/2}$
	infinity-norm	$\|A\|_\infty$	$\max_i \sum_j \|a_{ij}\|$
			$= \max_i \sum_j (Re(a_{ij})^2 + Im(a_{ij})^2)^{1/2}$
	real infinity-norm	$\|A\|_{\infty R}$	$\max_i \sum_j (\|Re(a_{ij})\| + \|Im(a_{ij})\|)$
	max-norm	$\|A\|_{\max}$	$\max_i \max_j \|a_{ij}\|$
			$= \max_i \max_j (Re(a_{ij})^2 + Im(a_{ij})^2)^{1/2}$
	real max-norm	$\|A\|_{\max R}$	$= \max_i \max_j (\|Re(a_{ij})\| + \|Im(a_{ij})\|)$

Table A.2: Matrix Norms

In contrast to computing vector norms, computing the two-norm and Frobenius-norm of a matrix are not equivalent. If the user asks for the two-norm of a matrix, where the matrix is 2-by-2 or larger, an error flag is raised. The one exception occurs when the matrix is a single column or a single row. In this case, the two-norm is requested and the Frobenius-norm is returned.

A.3 Operator Arguments

The following table lists the operator arguments and their associated named constants. For complete details of the meanings of the operator prec, refer to section 4.3.1.
Example: Consider the matrix-vector products $x = Ax$, $x = A^T x$ and $x = A^H x$. It is convenient to use the `trans` operator and define $op(A)$ as being A, A^T or A^H depending on the value of the `trans` operator argument. Again, the specification of the type and the valid values such an operator should have will be defined in the language-dependent section and may vary from one language binding to another.

It is worthwhile noticing that in some rare cases, the meaning of the `trans` operator argument is extended to a function of the matrix to which it applies. Consider for example the symmetric

operator argument	named constant	meaning
norm	blas_one_norm	1-norm
	blas_real_one_norm	real 1-norm
	blas_two_norm	2-norm
	blas_frobenius_norm	Frobenius-norm
	blas_inf_norm	infinity-norm
	blas_real_inf_norm	real infinity-norm
	blas_max_norm	max-norm
	blas_real_max_norm	real max-norm
sort	blas_increasing_order	sort in increasing order
	blas_decreasing_order	sort in decreasing order
side	blas_left_side	operate on the left-hand side
	blas_right_side	operate on the right-hand side
uplo	blas_upper	reference upper triangle only
	blas_lower	reference lower triangle only
trans$x$	blas_no_trans	operate with x
	blas_trans	operate with x^T
	blas_conj_trans	operate with x^H
conj	blas_conj	operate with $\bar{x}$
	blas_no_conj	operate with x
diag	blas_non_unit_diag	non-unit triangular
	blas_unit_diag	unit triangular
jrot	blas_jrot_inner	inner rotation $c \geq \frac{1}{\sqrt{2}}$
	blas_jrot_outer	outer rotation $0 \leq c \leq \frac{1}{\sqrt{2}}$
	blas_jrot_sorted	sorted rotation $abs(a) \geq abs(b)$
order	blas_colmajor	assume column-major ordering
	blas_rowmajor	assume row-major ordering
index_base	blas_zero_base	assumes zero-based indexing
	blas_one_base	assumes one-based indexing
prec	blas_prec_single	internal computation performed in single precision
	blas_prec_double	internal computation performed in double precision
	blas_prec_indigenous	internal computation performed in the widest hardware-supported format available
	blas_prec_extra	internal computation performed in format wider than 80-bits

Table A.3: Operator Arguments

rank-k update operations, $C \leftarrow C + AA^T$ and $C \leftarrow C + A^TA$ where C is a symmetric matrix. The value of the trans operator refers to the product AA^T. It follows that these operations can be specified by $C \leftarrow C + op(AA^T)$ where $op(AA^T)$ is AA^T or A^TA depending on the input value of the trans argument.

All possible values of the operator argument trans are not always meaningful. For example, in the symmetric rank-k update operations defined above, when the matrix C is complex symmetric, the only valid values of $op(AA^T)$ are AA^T or A^TA. Similarly, when the matrix C is complex Hermitian, the only valid values of $op(AA^H)$ are AA^H or A^HA. Such restrictions are detailed for each dense and banded BLAS function to which they apply.

Some BLAS routines have more than one trans operator argument because such an argument is needed for each matrix to which it applies. For example, a general matrix-multiply operation can be specified as $C \leftarrow op(A)op(B)$ where A, B and C are general matrices. A trans argument is needed for each of the input matrices A and B; by convention we denote those formal arguments transA and transB.

> *Rationale.* As mentioned above, section (1.4) does not specify how the objects manipulated by the BLAS routines are stored. This important aspect of the interface specification is deferred to the language-dependent specification sections. In particular, the operator arguments **do not** indicate whether only half or all entries of triangular, symmetric and Hermitian matrices are stored, or even how these entries are stored. The intent is to provide each language binding with the opportunity to choose the appropriate data structures for each object. Note that a given language binding specification may provide multiple functions performing the same operation on operands stored differently. For example, triangular matrices may be stored within conventional two-dimensional arrays or in packed storage, where the triangle may be packed by rows or columns. Consequently, a BLAS routine specified in the functionality tables may induce multiple functions in a particular language binding, say for instance, to provide the user with the same operation on objects that are stored differently. (*End of rationale.*)

It follows that, in general, a mathematical operation involving a matrix A, where A could be general or banded, triangular, symmetric or Hermitian, induces the language-independent specification of multiple routines. However, this language-independent section ignores the fact that a given language binding may choose to provide multiple storage schemes for some specific classes of matrices, such as triangular matrices.

A.4 Fortran 95 Modules

Several Fortran 95 modules are provided, allowing for the flexible inclusion of only select portions of the document. The modules `blas_dense`, `blas_sparse`, and `blas_extended`, are provided for Chapters 2, 3, and 4, respectively.

```
http://www.netlib.org/blas/blast-forum/blas_dense.f90
http://www.netlib.org/blas/blast-forum/blas_sparse.f90
http://www.netlib.org/blas/blast-forum/blas_extended.f90
```

Each of these modules in turn contains a `USE` statement to include the module of operator arguments (`blas_operator_arguments` for Chapters 2 and 4, and `blas_sparse_namedconstants` for Chapter 3), and the respective module(s) of explicit interfaces for that chapter.

For Chapters 2 and 4, one derived type is specified for each category of operator arguments (such as trans) and some parameters are defined of this type (for the different settings). For consistency, the suffix `_type` is used to name all of the derived types. This suffix is needed in some cases to differentiate between the type and one of the parameters (for example, `blas_trans_type` is a type and `blas_trans` is a parameter of this type). The Sparse BLAS chapter represents its operator arguments and a list of matrix properties (see section 3.5.1) as named constants.

Advice to implementors. For Chapter 2, all the entities (derived types, named constants and BLAS procedures) must be accessible to the user via the module `blas_dense`.

There are many ways to create this module. However the following three conditions **MUST** be adhered to:

- all entities can be accessed by the module
- the generic names must be the same as in the Fortran 95 bindings
- the specific name must be standard. The standard that we recommend is "suffix _d, _z, _s and _c" for double precision, double complex, real and complex.

For example the Fortran 95 bindings gives the generic name `gemm`. This is a generic procedure for the following 12 specific procedures:

gemm_d	corresponds to BLAS_DGEMM (legacy DGEMM)
gemm_z	corresponds to BLAS_ZGEMM (legacy ZGEMM)
gemm_s	corresponds to BLAS_SGEMM (legacy SGEMM)
gemm_c	corresponds to BLAS_CGEMM (legacy CGEMM)
gemv_d	corresponds to BLAS_DGEMV (legacy DGEMV)
gemv_z	corresponds to BLAS_ZGEMV (legacy ZGEMV)
gemv_s	corresponds to BLAS_SGEMV (legacy SGEMV)
gemv_c	corresponds to BLAS_CGEMV (legacy CGEMV)
ger_d	corresponds to BLAS_DGER (legacy DGER)
ger_z	corresponds to BLAS_ZGER (legacy SGER)
ger_s	corresponds to BLAS_SGER (legacy ZGERU, ZGERC)
ger_c	corresponds to BLAS_CGER (legacy CGERU, CGERC)

A specific procedure could be an external procedure or a module procedure.

One approach for creating the module `blas_dense` is to:

- create one file for each procedure
- create the interface blocks for the generic names using one or more modules
- create the module `blas_dense` from the modules in the last step and other modules such as `blas_operator_arguments`

Assuming we are using external procedures, the following files could be used as templates to create the module `blas_dense`. The interface blocks are grouped according to the grouping in section 2.8.1. The files are:

- `http://www.netlib.org/blas/blast-forum/blas_operator_arguments.f90`
  file containing the module blas_operator_arguments
- `http://www.netlib.org/blas/blast-forum/blas_precision.f90`
  file containing the module used to specify the precision (not visible to the user)
- `http://www.netlib.org/blas/blast-forum/blas_dense_red_op.f90`
  file containing the interface blocks for the reduction operations (section 2.8.2)
- `http://www.netlib.org/blas/blast-forum/blas_dense_gen_trans.f90`
  file containing the interface blocks for the generate transformations procedures (section 2.8.3)

- `http://www.netlib.org/blas/blast-forum/blas_dense_vec_op.f90`
  file containing the interface blocks for the vector operations (section 2.8.4)
- `http://www.netlib.org/blas/blast-forum/blas_dense_vec_mov.f90`
  file containing the interface blocks for the data movement with vectors (section 2.8.5)
- `http://www.netlib.org/blas/blast-forum/blas_dense_mat_vec_op.f90`
  file containing the interface blocks for the matrix_vector operations (section 2.8.6)
- `http://www.netlib.org/blas/blast-forum/blas_dense_mat_op.f90`
  file containing the interface blocks for the matrix operations (section 2.8.7)
- `http://www.netlib.org/blas/blast-forum/blas_dense_mat_mat_op.f90`
  file containing the interface blocks for the matrix_matrix operations (section 2.8.8)
- `http://www.netlib.org/blas/blast-forum/blas_dense_mat_mov.f90`
  file containing the interface blocks for the data movement with matrices (section 2.8.9)
- `http://www.netlib.org/blas/blast-forum/blas_dense_fpinfo.f90`
  file containing the interface blocks for the environmental enquiry (section 2.8.10)
- `http://www.netlib.org/blas/blast-forum/blas_dense.f90`
  file containing the module `blas_dense` that imports the information from all other modules and makes them available.

The specifications for all specific procedures MUST be as they appear in the above files. The only change is the way that the precision is specified. (*End of advice to implementors.*)

A.5 Fortran 77 Include File

One Fortran 77 include file is provided, `blas_namedconstants.h`. This include file contains the values of all named constants, and applies to Chapters 2, 3, and 4.

`http://www.netlib.org/blas/blast-forum/blas_namedconstants.h`

Operator arguments norm, sort, side, uplo, trans, conj, diag, jrot, index_base, and prec are represented in the Fortran 77 interface as INTEGERs. These operator arguments assume the named constant values as defined in section A.3. The Sparse BLAS chapter defines a list of matrix properties (see section 3.5.1) that must also be defined.

Advice to implementors. This specification is a deviation from the Legacy BLAS, where these operator arguments were defined as `CHARACTER*1`. (*End of advice to implementors.*)

A.6 C Include Files

Several C include files are provided, allowing for the flexible inclusion of only select portions of the document. The file `blas.h` contains the enumerated types and all prototypes for Chapters 2, 3, and 4. The files `blas_dense.h`, `blas_sparse.h`, and `blas_extended.h`, include the values of the operator arguments (enumerated types) and the function prototypes for the respective chapter.

```
http://www.netlib.org/blas/blast-forum/blas.h
http://www.netlib.org/blas/blast-forum/blas_dense.h
http://www.netlib.org/blas/blast-forum/blas_sparse.h
http://www.netlib.org/blas/blast-forum/blas_extended.h
```

The file `blas_enum.h` contains the values of all enumerated types, applying to all chapters. The files `blas_dense_proto.h`, `blas_sparse_proto.h`, and `blas_extended_proto.h`, contain the respective function prototypes for Chapters 2, 3, and 4.

```
http://www.netlib.org/blas/blast-forum/blas_enum.h
http://www.netlib.org/blas/blast-forum/blas_dense_proto.h
http://www.netlib.org/blas/blast-forum/blas_sparse_proto.h
http://www.netlib.org/blas/blast-forum/blas_extended_proto.h
```

All operator arguments are handled by enumerated types in the C interface. This allows for tighter error checking, and provides less opportunity for user error. In addition to the operator arguments of norm, sort, side, uplo, trans, conj, diag, jrot, index_base, and prec, this interface adds another such argument to all routines involving two dimensional arrays, order. order designates if the array elements are stored in row-major or column-major ordering. Refer to section 2.6.6 for further details. The Sparse BLAS chapter defines a list of matrix properties (see section 3.5.1) that must also be defined.

Annex B

Legacy BLAS

B.1 Introduction

This chapter addresses additional language bindings for the original Level 1, 2, and 3 BLAS. The Level 1, 2, and 3 BLAS will hereafter be referred to as the Legacy BLAS.

B.2 C interface to the Legacy BLAS

This section gives a detailed discussion of the proposed C interface to the legacy BLAS. Every mention of "BLAS" in this chapter should be taken to mean the legacy BLAS. Each interface decision is discussed in its own section. Each section also contains a *Considered methods* subsection, where other solutions to that particular problem are discussed, along with the reasons why those options were not chosen. These *Considered methods* subsections are indented and *italicized* in order to distinguish them from the rest of the text.

It is largely agreed among the group (and unanimous among the vendors) that user demand for a C interface to the BLAS is insufficient to motivate vendors to support a completely separate standard. This proposal therefore confines itself to an interface which can be readily supported on top of the already existing Fortran 77 callable BLAS (i.e., the legacy BLAS).

The interface is expressed in terms of ANSI/ISO C. Very few platforms fail to provide ANSI/ISO C compilers at this time, and for those platforms, free ANSI/ISO C compilers are almost always available (eg., `gcc`).

B.2.1 Naming scheme

The naming scheme consists of taking the Fortran 77 routine name, making it lower case, and adding the prefix `cblas_`. Therefore, the routine `DGEMM` becomes `cblas_dgemm`.

Considered methods

> *Various other naming schemes have been proposed, such as adding* `C_` *or* `c_` *to the name. Most of these schemes accomplish the requirement of separating the Fortran 77 and C name spaces. It was argued, however, that the addition of the* `blas` *prefix unifies the naming scheme in a logical and useful way (making it easy to search for BLAS use in a code, for instance), while not placing too great a burden on the typist. The letter* `c` *is used to distinguish this language interface from possible future interfaces.*

B.2.2 Indices and I_AMAX

The Fortran 77 BLAS return indices in the range $1 \leq I \leq N$ (where N is the number of entries in the dimension in question, and I is the index), in accordance with Fortran 77 array indexing conventions. This allows functions returning indices to be directly used to index standard arrays. The C interface therefore returns indices in the range $0 \leq I < N$ for the same reason.

The only BLAS routine which returns an index is the function `I_AMAX`. This function is declared to be of type `CBLAS_INDEX`, which is guaranteed to be an integer type (i.e., no cast is required when assigning to any integer type). `CBLAS_INDEX` will usually correspond to `size_t` to ensure any array can be indexed, but implementors might choose the integer type which matches their Fortran 77 `INTEGER`, for instance. It is defined that zero is returned as the index for a zero length vector (eg., For $N = 0$, `I_AMAX` will always return zero).

B.2.3 Character arguments

All arguments which were characters in the Fortran 77 interface are handled by enumerated types in the C interface. This allows for tighter error checking, and provides less opportunity for user error. The character arguments present in the Fortran 77 interface are: `SIDE`, `UPLO`, `TRANSPOSE`, and `DIAG`. This interface adds another such argument to all routines involving two dimensional arrays, `ORDER`. The standard dictates the following enumerated types:

```
enum CBLAS_ORDER     {CblasRowMajor=101, CblasColMajor=102};
enum CBLAS_TRANSPOSE {CblasNoTrans=111, CblasTrans=112, CblasConjTrans=113};
enum CBLAS_UPLO      {CblasUpper=121, CblasLower=122};
enum CBLAS_DIAG      {CblasNonUnit=131, CblasUnit=132};
enum CBLAS_SIDE      {CblasLeft=141, CblasRight=142};
```

Considered methods

> *The other two most commonly suggested methods were accepting these arguments as either* `char *` *or* `char`*. It was noted that both of these options require twice as many comparisons as normally required to branch (so that the character may be either upper or lower case). Both methods also suffered from ambiguity (what does it mean to have* `DIAG='H'`*, for instance). If* `char` *was chosen, the words could not be written out as they can for the Fortran 77 interface (you couldn't write "NoTranspose"). If* `char *` *were used, some compilers might fail to optimize string constant use, causing unnecessary memory usage.*
>
> *The main advantage of enumerated data types, however, is that much of the error checking can be done at compile time, rather than at runtime (i.e., if the user fails to pass one of the valid options, the compiler can issue the error).*
>
> *There was much discussion as to whether the integer values should be specified, or whether only the enumerated names should be so specified. The group could find no substansive way in which specifying the integer values would restrict an implementor, and specifying the integer values was seen as an aid to inter-language calls.*

B.2.4 Handling of complex data types

All complex arguments are accepted as `void *`. A complex element consists of two consecutive memory locations of the underlying data type (i.e., `float` or `double`), where the first location contains the real component, and the second contains the imaginary part of the number.

In practice, programmers' methods of handling complex types in C vary. Some use various data structures (some examples are discussed below). Others accept complex numbers as arrays of the underlying type.

Complex numbers are accepted as void pointers so that widespread type casting will not be required to avoid warning or errors during compilation of complex code.

An ANSI/ISO committee is presently working on an extension to ANSI/ISO C which defines complex data types. The definition of a complex element is the same as given above, and so the handling of complex types by this interface will not need to be changed when ANSI/ISO C standard is extended.

Considered methods

Probably the most strongly advocated alternative was defining complex numbers via a structure such as
`struct NON_PORTABLE_COMPLEX {float r; float i;};` *The main problem with this solution is the lack of portability. By the ANSI/ISO C standard, elements in a structure are not guaranteed to be contiguous. With the above structure, padding between elements has been experimentally observed (on the CRAY T3D), so this problem is not purely theoretical.*

To get around padding problems within the structure, a structure such as
`struct NON_PORTABLE_COMPLEX {float v[2];};` *has been suggested. With this structure there will obviously be no padding between the real and imaginary parts. However, there still exists the possibility of padding between elements within an array. More importantly, this structure does not lend itself nearly as well as the first to code clarity.*

A final proposal is to define a structure which may be addressed the same as the one above (i.e., `ptr->r`, `ptr->i`*), but whose actual definition is platform dependent. Then, hopefully, various vendors will either use the above structure and ensure via their compilers its contiguousness, or they will create a different structure which can be accessed in the same way.*

This requires vendors to support something which is not in the ANSI C standard, and so there is no way to ensure this would take place. More to the point, use of such a structure turns out to not offer much in the way of real advantage, as discussed in the following section.

All of these approaches require the programmer to either use the specified data type throughout the code which will call the BLAS, or to perform type casting on each BLAS call. When complex numbers are accepted as void pointers, no type casting or data type is dictated, with the only restriction being that a complex number have the definition given above.

B.2.5 Return values of complex functions

BLAS routines which return complex values in Fortran 77 are instead recast as subroutines in the C interface, with the return value being an output parameter added to the end of the argument list. This allows the output parameter to be accepted as void pointers, as discussed above.

Further, the name is suffixed by `_sub`. There are two main reasons for this name change. First, the change from a function to a subroutine is a significant change, and thus the name should reflect this. More importantly, the "traditional" name space is specifically reserved for use when the forthcoming ANSI/ISO C extension is finalized. When this is done, this C interface will be extended to include functions using the "traditional" names which utilize the new ANSI/ISO complex type to return the values.

Considered methods

> *This is the area where use of a structure is most desired. Again, the most common suggestion is a structure such as* `struct NON_PORTABLE_COMPLEX {float r; float i;};`.
>
> *If one is willing to use this structure throughout one's code, then this provides a natural and convenient mechanism. If, however, the programmer has utilized a different structure for complex, this ease of use breaks down. Then, something like the following code fragment is required:*

```
NON_PORTABLE_COMPLEX ctmp;
float cdot[2];

ctmp = cblas_cdotc(n, x, 1, y, 1);
cdot[0] = ctmp.r;
cdot[1] = ctmp.i;
```

> *which is certainly much less convenient than:* `cblas_cdotc_sub(n, x, 1, y, 1, cdot)`.
>
> *It should also be noted that the primary reason for having a function instead of a subroutine is already invalidated by C's lack of a standard complex type. Functions are most useful when the result may be used directly as part of an in-line computation. However, since ANSI/ISO C lacks support for complex arithmetic primitives or operator overloading, complex functions cannot be standardly used in this way. Since the function cannot be used as a part of a larger expression, nothing is lost by recasting it as a subroutine; indeed a slight performance win may be obtained.*

B.2.6 Array arguments

Arrays are constrained to being contiguous in memory. They are accepted as pointers, not as arrays of pointers.

All BLAS routines which take one or more two dimensional arrays as arguments receive one extra parameter as their first argument. This argument is of the enumerated type
`enum CBLAS_ORDER {CblasRowMajor=101, CblasColMajor=102};`.
If this parameter is set to `CblasRowMajor`, it is assumed that elements within a row of the array(s) are contiguous in memory, while elements within array columns are separated by a constant stride given in the `stride` parameter (this parameter corresponds to the leading dimension [e.g. `LDA`] in the Fortran 77 interface).

If the order is given as `CblasColMajor`, elements within array columns are assumed to be contiguous, with elements within array rows separated by `stride` memory elements.

Note that there is only one `CBLAS_ORDER` parameter to a given routine: all array operands are required to use the same ordering.

Considered methods

> *This solution comes after much discussion. C users appear to split roughly into two camps. Those people who have a history of mixing C and Fortran 77 (in particular making use of the Fortran 77 BLAS from C), tend to use column-major arrays in order to allow ease of inter-language operations. Because of the flexibility of pointers, this is*

not appreciably harder than using row-major arrays, even though C "natively" possesses row-major arrays.

The second camp of C users are not interested in overt C/Fortran 77 interoperability, and wish to have arrays which are row-major, in accordance with standard C conventions. The idea that they must recast their row-oriented algorithms to column-major algorithms is unacceptable; many in this camp would probably not utilize any BLAS which enforced a column-major constraint.

Because both camps are fairly widely represented within the target audience, it is impossible to choose one solution to the exclusion of the other.

Column-major array storage can obviously be supported directly on top of the legacy Fortran 77 BLAS. Recent work, particularly code provided by D.P. Manley of DEC, has shown that row-major array storage may also be supported in this way with little cost. Appendix B.2.12 discusses this issue in detail. To preview it here, we can say the level 1 and 3 BLAS require no extra operations or storage to support row-major operations on top of the legacy BLAS. Level 2 real routines also require no extra operations or storage. Some complex level 2 routines involving the conjugate transpose will require extra storage and operations in order to form explicit conjugates. However, this will always involve vectors, not the matrix. In the worst case, we will need n extra storage, and $3n$ sign changes.

One proposal was to accept arrays as arrays of pointers, instead of as a single pointer. The problems with this approach are manifold. First, the existing Fortran 77 BLAS could not be used, since they demand contiguous (though strided) storage. Second, this approach requires users of standard C 2D arrays or 1D arrays to allocate and assign the appropriate pointer array.

Beyond this, many of the vectors used in level 1 and level 2 BLAS come from rows or columns of two dimensional arrays. Elements within columns of row-major arrays are not uniformly strided, which means that a `n`*-element column vector would need* `n` *pointers to represent it. This then leads to vectors being accepted as arrays of pointers as well.*

Now, assuming both our one and two dimensional arrays are accepted as arrays of pointers, we have a problem when we wish to perform sub-array access. If we wish to pass an $m \times n$ subsection of a this array of pointers, starting at row i and column j, we must allocate m pointers, and assign them in a section of code such as:

```
float **A, **subA;

subA = malloc(m*sizeof(float*));
for (k=0; k != m; k++) subA[k] = A[i+k] + j;
cblas_rout(... subA ...);
```

The same operation must be done if we wish to use a row or column as a vector. This is not only an inconvenience, but can add up to a non-negligible performance loss as well.

A fix for these problems is that one and two dimensional arrays be passed as arrays of pointers, and then indices are passed in to indicate the sub-portion to access. Thus you have a call that looks like: `cblas_rout(... A, i, j, ...);`*. This solution still requires some additional tweaks to allow using two dimensional array rows and columns as vectors. Users presently using C 2D arrays or 1D arrays would have to malloc the* *array of pointers as shown in the preceding example in order to use this kind of interface. At any rate, a library accepting pointers to pointers cannot be supported on top of the Fortran 77 BLAS, while one supporting simple pointers can.*

If the programmer is utilizing the pointer to pointer style of array indexing, it is still possible to use this library providing that the user ensures that the operand matrix is contiguous, and that the rows are constantly strided. If this is the case, the user may pass the operand matrix to the library in precicely the same way as with a 2D C array: `cblas_rout(... &A[i][j] ...);`.

Example 1: making a library call with a C 2D array:

```
double A[50][25];  /* standard C 2D array */

cblas_rout(CblasRowMajor, ... &A[i][j], 25, ...);
```

Example 2: Legal use of pointer to pointer style programming and the CBLAS

```
double **A, *p;

A = malloc(M);
p = malloc(M*N*sizeof(double));
for (i=0; i < M; i++) A[i] = &p[i*N];

cblas_rout(CblasRowMajor, ... &A[i][j], N, ...);
```

Example 3: Illegal use of pointer to pointer style programming and the CBLAS

```
double **A, *p;

A = malloc(M);
p = malloc(M*N*sizeof(double));
for (i=0; i < M; i++) A[i] = malloc(N*sizeof(double));

cblas_rout(CblasRowMajor, ... &A[i][j], N, ...);
```

Note that Example 3 is illegal because the rows of A have no guaranteed stride.

B.2.7 Aliasing of arguments

Unless specified otherwise, only input-only arguments (specified with the `const` qualifier), may be legally aliased on a call to the C interface to the BLAS.

Considered methods

The ANSI C standard allows for the aliasing of output arguments. However, allowing this often carries a substantial performance penalty. This, along with the fact that Fortran 77 (which we hope to call for optimized libraries) does not allow aliasing of output arguments, led us to make this restriction.

B.2.8 C interface include file

The C interface to the BLAS will have a standard include file, called `cblas.h`, which minimally contains the definition of the CBLAS types and ANSI/ISO C prototypes for all BLAS routines. It is not an error to include this file multiple times. Refer to section B.2.11 for an example of a minimal `cblas.h`.

ADVICE TO THE IMPLEMENTOR:
Note that the vendor is not constrained to using precisely this include file; only the enumerated type definitions are fully specified. The implementor is free to make any other changes which are not apparent to the user. For instance, all matrix dimensions might be accepted as `size_t` *instead of* `int`*, or the implementor might choose to make some routines inline.*

B.2.9 Error checking

The C interface to the legacy BLAS must supply error checking corresponding to that provided by the reference Fortran 77 BLAS implementation.

B.2.10 Rules for obtaining the C interface from the Fortran 77

- The Fortran 77 routine name is changed to lower case, and prefixed by `cblas_`.
- All routines which accept two dimensional arrays (i.e., level 2 and 3), acquire a new parameter of type `CBLAS_ORDER` as their first argument, which determines if the two dimensional arrays are row or column major.
- *Character arguments* are replaced by the appropriate enumerated type, as shown in Section B.2.3.
- *Input arguments* are declared with the `const` modifier.
- *Non-complex scalar input arguments* are passed by value. This allows the user to put in constants when desired (eg., passing 10 on the command line for `N`).
- *Complex scalar input arguments* are passed as void pointers, since they do not exist as a predefined data type in ANSI/ISO C.
- *Array arguments* are passed by address.
- *Output scalar arguments* are passed by address.
- *Complex functions* become subroutines which return the result via a void pointer, added as the last parameter. The name is suffixed with `_sub`.

B.2.11 cblas.h include file

The `cblas.h` include file can be found on the BLAS Technical Forum webpage:

`http://www.netlib.org/blas/blast-forum/cblas.h`

B.2.12 Using Fortran 77 BLAS to support row-major BLAS operations

This section is not part of the standard per se. Rather, it exists as an advice to the implementor on how row-major BLAS operations may be implemented using column-major BLAS. This allows vendors to leverage years of Fortran 77 BLAS developement in producing the C BLAS.

Before this issue is examined in detail, a few general observations on array storage are helpful. We must distinguish between the matrix and the array which is used to store the matrix. The matrix, and its rows and columns, have mathematical meaning. The array is simply the method of storing the matrix, and its rows and columns are significant only for memory addressing.

Thus we see we can store the columns of a matrix in the rows of an array, for instance. When this occurs in the BLAS, the matrix is said to be stored in transposed form.

A row-major array stores elements along a row in contiguous storage, and separates the column elements by some constant stride (often the actual length of a row). Column-major arrays have contiguous columns, and strided rows. The importance of this is to note that a row-major array storing a matrix in the natural way, is a transposed column-major array (i.e., it can be thought of as a column-major array where the rows of the matrix are stored in the columns of the array).

Similarly, an upper triangular row-major array corresponds to a transposed lower triangular column-major array (the same is true in reverse [i.e., lower-to-upper], obviously). To see this, simply think of what a upper triangular matrix stored in a row-major array looks like. The first n entries contain the first matrix row, followed by a non-negative gap, followed by the second matrix row.

If this same array is viewed as column-major, the first n entries are a column, instead of a row, so that the columns of the array store the rows of the matrix (i.e., it is transposed). This means that if we wish to use the Fortran 77 (column-major) BLAS with triangular matrices coming from C (possibly row-major), we will be reversing the setting of `UPLO`, while simultaneously reversing the setting of `TRANS` (this gets slightly more complicated when the conjugate transpose is involved, as we will see).

Finally, note that if a matrix is symmetric or Hermitian, its rows are the same as its columns, so we may merely switch `UPLO`, without bothering with `TRANS`.

In the BLAS, there are two separate cases of importance. one dimensional arrays (storage for vectors) have the same meaning in both C and Fortran 77, so if we are solving a linear algebra problem who's answer is a vector, we will need to solve the same problem for both languages. However, if the answer is a matrix, in terms of calling routines which use column-major storage from one using row-major storage, we will want to solve the *transpose* of the problem.

To get an idea of what this means, consider a contrived example. Say we have routines for simple matrix-matrix and matrix-vector multiply. The vector operation is $y \leftarrow A \times x$, and the matrix operation is $C \leftarrow A \times B$. Now say we are implementing these as calls from row-major array storage to column-major storage. Since the matrix-vector multiply's answer is a vector, the problem we are solving is remains the same, but we must remember that our C array A is a Fortran 77 A^T. On the other hand, the matrix-matrix multiply has a matrix for a result, so when the differing array storage is taken into account, the problem we want to solve is $C^T \leftarrow B^T \times A^T$.

This last example demonstrates another general result. Some level 3 BLAS contain a `SIDE` parameter, determining which side a matrix is applied on. In general, if we are solving the transpose of this operation, the side parameter will be reversed.

With these general principles, it is possible to show that all that row-major level 3 BLAS can be expressed in terms of column-major BLAS without any extra array storage or extra operations. In the level 2 BLAS, no extra storage or array accesses are required for the real routines. Complex routines involving the conjugate transpose, however, may require a n-element temporary, and up to $3n$ more operations (vendors may avoid all extra workspace and operations by overloading the `TRANS` option for the level 2 BLAS: letting it also allow conjugation without doing the transpose). The level 1 BLAS, which deal exclusively with vectors, are unaffected by this storage issue.

With these ideas in mind, we will now show how to support a row-major BLAS on top of a column major BLAS. This information will be presented in tabular form. For brevity, row-major storage will be referred to as coming from C (even though column-major arrays can also come from C), while column-major storage will be referred to as F77.

Each table will show a BLAS invocation coming from C, the operation that the BLAS should perform, the operation required once F77 storage is taken into account (if this changes), and the call to the appropriate F77 BLAS. Not every possible combination of parameters is shown, since many are simply reflections of another (i.e., when we are applying the `Upper, NoTranspose` becomes `Lower, Transpose` rule, we will show it for only the upper case. In order to make the notation more concise, let us define $\overline{x}$ to be $conj(x)$.

Level 2 BLAS

GEMV

C call	`cblas_cgemv(CblasRowMajor, CblasNoTrans, m, n,` α`, A, lda, x, incx,` β`, y, incy)`
op	$y \leftarrow \alpha A x + \beta y$
F77 call	`CGEMV('T', n, m,` α`, A, lda, x, incx,` β`, y, incy)`

C call	`cblas_cgemv(CblasRowMajor, CblasTrans, m, n,` α`, A, lda, x, incx,` β`, y, incy)`
op	$y \leftarrow \alpha A^T x + \beta y$
F77 call	`CGEMV('N', n, m,` α`, A, lda, x, incx,` β`, y, incy)`

C call	`cblas_cgemv(CblasRowMajor, CblasConjTrans, m, n,` α`, A, lda, x, incx,` β`, y, incy)`
op	$y \leftarrow \alpha A^H x + \beta y \Rightarrow \overline{(\overline{y} \leftarrow \overline{\alpha} A^T \overline{x} + \overline{\beta}\overline{y})}$
F77 call	`CGEMV('N', n, m,` $\overline{\alpha}$`, A, lda,` $\overline{x}$`, 1,` $\overline{\beta}$`,` $\overline{y}$`, incy)`

Note that we switch the value of transpose to handle the row/column major ordering difference. In the last case, we will require n elements of workspace so that we may store the conjugated vector $\overline{x}$. Then, we set $y = \overline{y}$, and make the call. This gives us the conjugate of the answer, so we once again set $y = \overline{y}$. Therefore, we see that to support the conjugate transpose, we will need to allocate an n-element vector, and perform $2m + n$ extra operations.

SYMV

SYMV requires no extra workspace or operations.

C call	`cblas_csymv(CblasRowMajor, CblasUpper, n,` α`, A, lda, x, incx,` β`, y, incy)`
op	$y \leftarrow \alpha A x + \beta y \Rightarrow y \leftarrow \alpha A^T x + \beta y$
F77 call	`CSYMV('L', n,` α`, A, lda, x, incx,` β`, y, incy)`

HEMV

HEMV routine requires $3n$ conjugations, and n extra storage.

C call	`cblas_chemv(CblasRowMajor, CblasUpper, n,` α`, A, lda, x, incx,` β`, y, incy)`
op	$y \leftarrow \alpha A x + \beta y \Rightarrow y \leftarrow \alpha A^H x + \beta y \Rightarrow \overline{(\overline{y} \leftarrow \overline{\alpha} A^T \overline{x} + \overline{\beta}\overline{y})}$
F77 call	`CHEMV('L', n,` $\overline{\alpha}$`, A, lda,` $\overline{x}$`, incx,` $\overline{\beta}$`,` $\overline{y}$`, incy)`

TRMV/TRSV

C call	`cblas_ctrmv(CblasRowMajor, CblasUpper, CblasNoTrans, diag, n, A, lda, x, incx)`
op	$x \leftarrow Ax$
F77 call	`CTRMV('L', 'T', diag, n, A, lda, x, incx)`

C call	`cblas_ctrmv(CblasRowMajor, CblasUpper, CblasTrans, diag, n, A, lda, x, incx)`
op	$x \leftarrow A^T x$
F77 call	`CTRMV('L', 'N', diag, n, A, lda, x, incx)`

C call	`cblas_ctrmv(CblasRowMajor, CblasUpper, CblasConjTrans, diag, n, A, lda, x, incx)`
op	$x \leftarrow A^H x \Rightarrow \overline{(\overline{x} = A^T \overline{x})}$
F77 call	CTRMV('L', 'N', diag, n, A, lda, $\overline{x}$, incx)

Again, we see that we will need some extra operations when we are handling the conjugate transpose. We conjugate x before the call, giving us the conjugate of the answer we seek. We then conjugate this again to return the correct answer. This routine therefore needs $2n$ extra operations for the complex conjugate case.

The calls with the C array being `Lower` are merely the reflection of these calls, and thus are not shown. The analysis for TRMV is the same, since it involves the same principle of what a transpose of a triangular matrix is.

GER/GERU

This is our first routine that has a matrix as the solution. Recalling that this means we solve the transpose of the original problem, we get:

C call	cblas_cgeru(CblasRowMajor, m, n, α, x, incx, y, incy, A, lda)
C op	$A \leftarrow \alpha x y^T + A$
F77 op	$A^T \leftarrow \alpha y x^T + A^T$
F77 call	CGERU(n, m, α, y, incy, x, incx, A, lda)

No extra storage or operations are required.

GERC

C call	cblas_cgerc(CblasRowMajor, m, n, α, x, incx, y, incy, A, lda)
C op	$A \leftarrow \alpha x y^H + A$
F77 op	$A^T \leftarrow \alpha (x y^H)^T + A^T = \alpha \overline{y} x^T + A^T$
F77 call	CGERU(n, m, α, $\overline{y}$, incy, x, incx, A, lda)

Note that we need to allocate n-element workspace to hold the conjugated y, and we call `GERU`, not `GERC`.

HER

C call	cblas_cher(CblasRowMajor, CblasUpper, n, α, x, incx, A, lda)
C op	$A \leftarrow \alpha x x^H + A$
F77 op	$A^T \leftarrow \alpha \overline{x} x^T + A^T$
F77 call	CHER('L', n, α, $\overline{x}$, 1, A, lda)

Again, we have an n-element workspace and n extra operations.

HER2

C call	`cblas_cher2(CblasRowMajor, CblasUpper, n,` α`, x, incx, y, incy, A, lda)`
C op	$A \leftarrow \alpha x y^H + y(\alpha x)^H + A$
F77 op	$A^T \leftarrow \alpha \overline{y} x^T + \overline{\alpha x} y^T + A^T = \alpha \overline{y}(\overline{x})^H + \overline{x}(\alpha \overline{y})^H + A^T$
F77 call	`CHER2('L', n,` α`,` $\overline{y}$`, 1,` $\overline{x}$`, 1, A, lda)`

So we need $2n$ extra workspace and operations to form the conjugates of x and y.

SYR

C call	`cblas_ssyr(CblasRowMajor, CblasUpper, n,` α`, x, incx, A, lda)`
C op	$A \leftarrow \alpha x x^T + A$
F77 op	$A^T \leftarrow \alpha x x^T + A^T$
F77 call	`SSYR('L', n,` α`, x, incx, A, lda)`

No extra storage or operations required.

SYR2

C call	`cblas_ssyr2(CblasRowMajor, CblasUpper, n,` α`, x, incx, y, incy, A, lda)`
C op	$A \leftarrow \alpha x y^T + \alpha y x^T + A$
F77 op	$A^T \leftarrow \alpha y x^T + \alpha x y^T + A^T$
F77 call	`SSYR2('L', n,` α`, y, incy, x, incx, A, lda)`

No extra storage or operations required.

Level 3 BLAS

GEMM

C call	cblas_cgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha AB + \beta C$
F77 op	$C^T \leftarrow \alpha B^T A^T + \beta C^T$
F77 call	CGEMM('N', 'N', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasNoTrans, CblasTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha AB^T + \beta C$
F77 op	$C^T \leftarrow \alpha BA^T + \beta C^T$
F77 call	CGEMM('T', 'N', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasNoTrans, CblasConjTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha AB^H + \beta C$
F77 op	$C^T \leftarrow \alpha \overline{B} A^T + \beta C^T$
F77 call	CGEMM('C', 'N', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasTrans, CblasNoTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^T B + \beta C$
F77 op	$C^T \leftarrow \alpha B^T A + \beta C^T$
F77 call	CGEMM('N', 'T', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasTrans, CblasTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^T B^T + \beta C$
F77 op	$C^T \leftarrow \alpha BA + \beta C^T$
F77 call	CGEMM('T', 'T', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasTrans, CblasConjTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^T B^H + \beta C$
F77 op	$C^T \leftarrow \alpha \overline{B} A + \beta C^T$
F77 call	CGEMM('C', 'T', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasConjTrans, CblasNoTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^H B + \beta C$
F77 op	$C^T \leftarrow \alpha B^T \overline{A} + \beta C^T$
F77 call	CGEMM('N', 'C', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasConjTrans, CblasTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^H B^T + \beta C$
F77 op	$C^T \leftarrow \alpha B \overline{A} + \beta C^T$
F77 call	CGEMM('T', 'C', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasConjTrans, CblasConjTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^H B^H + \beta C$
F77 op	$C^T \leftarrow \alpha \overline{BA} + \beta C^T$
F77 call	CGEMM('C', 'C', n, m, k, α, B, ldb, A, lda, β, C, ldc)

SYMM/HEMM

C call	`cblas_chemm(CblasRowMajor, CblasLeft, CblasUpper, m, n,` α`, A, lda, B, ldb,` β`, C, ldc)`
C op	$C \leftarrow \alpha AB + \beta C$
F77 op	$C^T \leftarrow \alpha B^T A^T + \beta C^T$
F77 call	`CHEMM('R', 'L', n, m,` α`, A, lda, B, ldb,` β`, C, ldc)`

C call	`cblas_chemm(CblasRowMajor, CblasRight, CblasUpper, m, n,` α`, A, lda, B, ldb,` β`, C, ldc)`
C op	$C \leftarrow \alpha BA + \beta C$
F77 op	$C^T \leftarrow \alpha A^T B^T + \beta C^T$
F77 call	`CHEMM('L', 'L', n, m,` α`, A, lda, B, ldb,` β`, C, ldc)`

SYRK

C call	`cblas_csyrk(CblasRowMajor, CblasUpper, CblasNoTrans, n, k,` α`, A, lda,` β`, C, ldc)`
C op	$C \leftarrow \alpha AA^T + \beta C$
F77 op	$C^T \leftarrow \alpha AA^T + \beta C^T$
F77 call	`CSYRK('L', 'T', n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`

C call	`cblas_csyrk(CblasRowMajor, CblasUpper, CblasTrans, n, k,` α`, A, lda,` β`, C, ldc)`
C op	$C \leftarrow \alpha A^T A + \beta C$
F77 op	$C^T \leftarrow \alpha A^T A + \beta C^T$
F77 call	`CSYRK('L', 'N', n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`

In reading the above descriptions, it is important to remember a few things. First, the symmetric matrix is C, and thus we change `UPLO` to accommodate the differing storage of C. `TRANSPOSE` is then varied to handle the storage effects on A.

HERK

C call	`cblas_cherk(CblasRowMajor, CblasUpper, CblasNoTrans, n, k,` α`, A, lda,` β`, C, ldc)`
C op	$C \leftarrow \alpha AA^H + \beta C$
F77 op	$C^T \leftarrow \alpha \overline{A} A^T + \beta C^T$
F77 call	`CHERK('L', 'C', n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`

C call	`cblas_cherk(CblasRowMajor, CblasUpper, CblasConjTrans, n, k,` α`, A, lda,` β`, C, ldc)`
C op	$C \leftarrow \alpha A^H A + \beta C$
F77 op	$C^T \leftarrow \alpha A^T \overline{A} + \beta C^T$
F77 call	`CHERK('L', 'N', n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`

SYR2K

C call	`cblas_csyr2k(CblasRowMajor, CblasUpper, CblasNoTrans, n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`
C op	$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$
F77 op	$C^T \leftarrow \alpha BA^T + \alpha AB^T + \beta C^T = \alpha AB^T + \alpha BA^T + \beta C^T$
F77 call	`CSYR2K('L', 'T', n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`

C call	`cblas_csyr2k(CblasRowMajor, CblasUpper, CblasTrans, n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`
C op	$C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$
F77 op	$C^T \leftarrow \alpha B^T A + \alpha A^T B + \beta C^T = \alpha A^T B + \alpha B^T A + \beta C^T$
F77 call	`CSYR2K('L', 'N', n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`

Note that we once again wind up with an operation that looks the same from C and Fortran 77, saving that the C operations wishes to form C^T, instead of C. So once again we flip the setting of `UPLO` to handle the difference in the storage of C. We then flip the setting of `TRANS` to handle the storage effects for A and B.

HER2K

C call	`cblas_cher2k(CblasRowMajor, CblasUpper, CblasNoTrans, n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`
C op	$C \leftarrow \alpha AB^H + \overline{\alpha}BA^H + \beta C$
F77 op	$C^T \leftarrow \alpha\overline{B}A^T + \overline{\alpha}\overline{A}B^T + \beta C^T = \overline{\alpha}\overline{A}B^T + \alpha\overline{B}A^T + \beta C^T$
F77 call	`CHER2K('L', 'C', n, k,` $\overline{\alpha}$`, A, lda, B, ldb,` β`, C, ldc)`

C call	`cblas_cher2k(CblasRowMajor, CblasUpper, CblasConjTrans, n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`
C op	$C \leftarrow \alpha A^H B + \overline{\alpha}B^H A + \beta C$
F77 op	$C^T \leftarrow \alpha B^T\overline{A} + \overline{\alpha}A^T\overline{B} + \beta C^T = \overline{\alpha}A^T\overline{B} + \alpha B^T\overline{A} + \beta C^T$
F77 call	`CHER2K('L', 'N', n, k,` $\overline{\alpha}$`, A, lda, B, ldb,` β`, C, ldc)`

TRMM/TRSM

Because of their identical use of the `SIDE`, `UPLO`, and `TRANSA` parameters, TRMM and TRSM share the same general analysis. Remember that A is a triangular matrix, and thus when we handle its storage by flipping `UPLO`, we implicitly change its `TRANS` setting as well. With this in mind, we have:

C call	`cblas_ctrmm(CblasRowMajor, CblasLeft, CblasUpper, CblasNoTrans, diag, m, n,` α`, A, lda, B, ldb)`
C op	$B \leftarrow \alpha AB$
F77 op	$B^T \leftarrow \alpha B^T A^T$
F77 call	`CTRMM('R', 'L', 'N', diag, n, m,` α`, A, lda, B, ldb)`

C call	`cblas_ctrmm(CblasRowMajor, CblasLeft, CblasUpper, CblasTrans, diag, m, n,` α`, A, lda, B, ldb)`
C op	$B \leftarrow \alpha A^T B$
F77 op	$B^T \leftarrow \alpha B^T A$
F77 call	`CTRMM('R', 'L', 'T', diag, n, m,` α`, A, lda, B, ldb)`

C call	`cblas_ctrmm(CblasRowMajor, CblasLeft, CblasUpper, CblasConjTrans, diag, m, n,` α`, A, lda, B, ldb)`
C op	$B \leftarrow \alpha A^H B$
F77 op	$B^T \leftarrow \alpha B^T\overline{A}$
F77 call	`CTRMM('R', 'L', 'C', diag, n, m,` α`, A, lda, B, ldb)`

Banded routines

The above techniques can be used for the banded routines only if a C (row-major) banded array has some sort of meaning when expanded as a Fortran banded array. It turns out that when this is done, you get the transpose of the C array, just as in the dense case.

In Fortran 77, the banded array is an array whose rows correspond to the diagonals of the matrix, and whose columns contain the selected portion of the matrix column. To rephrase this, the diagonals of the matrix are stored in strided storage, and the relevant pieces of the columns of the matrix are stored in contiguous memory. This makes sense: in a column-based algorithm, you will want your columns to be contiguous for efficiency reasons.

In order to ensure our columns are contiguous, we will structure the banded array as shown below. Notice that the first K_U rows of the array store the superdiagonals, appropriately spaced to line up correctly in the column direction with the main diagonal. The last K_L rows contain the subdiagonals.

```
       ------  Super diagonal KU
  -----------  Super diagonal 2
 ------------  Super diagonal 1
-------------  main diagonal (D)
------------   Sub diagonal 1
-----------    Sub diagonal 2
------         Sub diagonal KL
```

If we have a row-major storage, and thus a row-oriented algorithm, we will similarly want our rows to be contiguous in order to ensure efficiency. The storage scheme that is thus dictated is shown below. Notice that the first K_L columns store the subdiagonals, appropriately padded to line up with the main diagonal along rows.

```
 KL   D   KU
      | | | |
    | | | | |
  | | | | | |
| | | | | |
| | | | |
| | | |
```

Now, let us contrast these two storage schemes. Both store the diagonals of the matrix along the non-contiguous dimension of the matrix. The column-major banded array stores the matrix columns along the contiguous dimension, whereas the row-major banded array stores the matrix rows along the contiguous storage.

This gives us our first hint as to what to do: rows stored where columns should be, indicated, in the dense routines, that we needed to set a transpose parameter. We will see that we can do this for the banded routines as well.

We can further note that in the column-major banded array, the first part of the non-contiguous dimension (i.e. the first rows) store superdiagonals, whereas the first part of the non-contiguous dimension of row-major arrays (i.e., the first columns) store the subdiagonals.

We now note that when you transpose a matrix, the superdiagonals of the matrix become the subdiagonals of the matrix transpose (and vice versa).

Along the contiguous dimension, we note that we skip K_U elements before coming to our first entry in a column-major banded array. The same happens in our row-major banded array, except that the skipping factor is K_L.

All this leads to the idea that when we have a row-major banded array, we can consider it as a transpose of the Fortran 77 column-major banded array, where we will swap not only m and n, but also K_U and K_L. An example should help demonstrate this principle. Let us say we have the matrix $A = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \end{bmatrix}$

If we express this entire array in banded form (a fairly dumb thing to do, but good for example purposes), we get $K_U = 3$, $K_L = 1$. In row-major banded storage this becomes: $C_b = \begin{bmatrix} X & 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 & X \end{bmatrix}$

So, we believe this should be the transpose if interpreted as a Fortran 77 banded array. The matrix transpose, and its Fortran 77 banded storage is shown below:

$$A^T = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} \Rightarrow F_b = \begin{bmatrix} X & 2 \\ 1 & 4 \\ 3 & 6 \\ 5 & 8 \\ 7 & X \end{bmatrix}$$

Now we simply note that since C_b is row major, and F_b is column-major, they are actually the same array in memory.

With the idea that row-major banded matrices produce the transpose of the matrix when interpreted as column-major banded matrices, we can use the same analysis for the banded BLAS as we used for the dense BLAS, noting that we must also always swap K_U and K_L.

Packed routines

Packed routines are much simpler than banded. Here we have a triangular, symmetric or Hermitian matrix which is packed so that only the relevant triangle is stored. Thus if we have an upper triangular matrix stored in column-major packed storage, the first element holds the relevant portion of the first column of the matrix, the next two elements hold the relevant portion of the second column, etc.

With an upper triangular matrix stored in row-major packed storage, the first N elements hold the first row of the matrix, the next $N-1$ elements hold the next row, etc.

Thus we see in the Hermitian and symmetric cases, to get a row-major packed array correctly interpreted by Fortran 77, we will simply switch the setting of `UPLO`. This will mean that the rows of the matrix will be read in as the columns, but this is not a problem, as we have seen before. In the symmetric case, since $A = A^T$ the column and rows are the same, so there is obviously no problem. In the Hermitian case, we must be sure that the imaginary component of the diagonal is not used, and it assumed to be zero. However, the diagonal element in a row when our matrix is upper will correspond to the diagonal element in a column when our matrix is called lower, so this is handled as well.

In the triangular cases, we will need to change both `UPLO` and `TRANS`, just as in the dense routines.

With these ideas in mind, the analysis for the dense routines may be used unchanged for packed.

Annex C

Journal of Development

C.1 Environmental Routine for Effective use of Cache, Pipelining and Registers

C.1.1 Introduction

It is well known that effective use of cache and registers can make a substantial difference in the performance of codes written to do the core algorithms of linear algebra. Existing computer languages could make it possible to write portable code without losing efficiency by providing a set of functions to probe the system parameters. A user would then construct a program to use the parameters to develop an optimal implementation on the target computer. With many optimizing compilers, a preprocessor could be used to get much of these benefits. It would not be expected to do as good a job, and would still require some selection of parameters for each new machine. The ideas apply to PC's, workstations, and to the code running on a single node of most new parallel machines.

Computers have been evolving in such a way that arithmetic speeds far exceed the rate with which operands from the main memory can be obtained for processing. Because it is cost effective to do so, many new parallel machines are using microprocessors for which this is the case. Since it is frequently the case that when something is used, it is going to be used again reasonably soon, systems are designed to use one or more levels of cache, (i.e. fast memory) where data used recently can be reused more quickly. This approach works quite well for most users, but when working with large dense matrices, data is never in the cache when needed. This can easily cost a factor of two or three in performance and in many cases significantly more.

We regard a computer's registers as a special case of a very small cache, since operations done on registers proceed more quickly than those using an operand that is in the fastest cache. (RISC architectures require that operands be in the registers.) But the differences between registers and cache memory require that different mechanisms be defined to make most effective use of both.

At every level of cache (including the registers), one wants to do as much computing as possible before the data in the cache is flushed for other data references. There is great scope for cleverness here in the design of algorithms. Since matrix multiplication is simple to understand we use it as an example to clarify how the ideas presented here might be used.

We believe that with the kind of features described here, it would not be difficult for those who know about such things to write compilers that would make it possible for those who know about such things to write portable code with no significant loss of efficiency. In addition it would be possible to write applications that can take advantage of these parameters and enhance the performance on a wide range of applications.

The most notable effort to deal with these problems is the Level 3 BLAS,[24] (it should be noted that we make no pretense to discussing all the factors that may be important to obtaining high performance on modern processors.

C.1.2 Language Extensions for the Cache

The cache system of a computer can be characterized by the following:

We expect $\sigma_L > \sigma_{L-1}$ and $\tau_L > \tau_{L-1}$, $L = 1, 2, \ldots, kache$, where we adopt the convention that L_0 corresponds to these values for the registers. We define σ_{kache} to be the largest amount of memory available, which in the case of virtual memory includes disk space. We make no further reference to the τ's, although there are certainly cases when such information might be of use.

In the case of Fortran, these values could be provided by environmental intrinsic functions, which take an argument of the type for which space is desired. In the case of C, these could be provided as part of the standard *math.h* header file. In the case of compilers being used in environments with different cache characteristics, it would be useful to have some means to specify

kache	The number of levels in the cache architecture.
σ_L	The size of the Lth cache.
τ_L	The access time to get data from the Lth cache divided by
	the time to copy one floating point register to another.
w_L	Number of consecutive items that get replaced when a new
	item is fetched to cache L.

this information in a configuration file. Finally if this were done using a preprocessor, one would provide the information for the system being used to the preprocessor.

Extensions that would allow one to gain the efficiency using the BLAS without writing machine specific code.

C.1.3 For Efficient LA Software

There are at least two things that might be done. Provide information through an interface about the cache structure of the machine, number of caches, their sizes, their access times, the size of the cache line, issues connected with pipelining, etc. Allow the programmer to declare variables and arrays (very small arrays, but still arrays) that are to be kept in registers. It is up to the compiler to pick the size of the arrays, and to unwind loops that refer to such "register" arrays. The compiler makes available the size of the arrays that it selects so that the programmer can reference them for purposes of writing loops, or for any other purpose.

C.1.4 Advantages of this approach

Code need only be written once, no dependence on computer vendors, or when a machine first comes out efficient implementations of important software will be available. If one has the stomach for it, it should be possible to write algorithms exactly as you would like to have them. Thus for example, one doesn't have to organize things to use a matrix multiplication if that is not the most effective way to do things.

C.1.5 Disadvantages of this approach

Code is significantly more difficult to write. It may require some cooperation from compiler vendors. What is proposed here can be done by a preprocessor if one can supply it with the cache information (probably not too hard) and the size that would be appropriate for the register arrays. This can be determined on a machine by machine basis by trying a few possibilities.

Characterizing a Cache System
The number of levels in the cache architecture.
The size of the Lth cache.
The access time to get data from the Lth cache divided by the time to copy one floating point register to another.
Number of consecutive items that get replaced when a new item is fetched to cache L.
Cache mapping: set associatively, direct

Machine Characteristics
Number of floating point registers
Number of floating point units
Number of caches
Cache size
Type of cache e.g. 2 way set associative, least recently used
Cache line size
Cycle time
Page size
Size of TLB (translation look-aside buffer)
Cycles for floating point operations, add, multiply, division
Number of processors
Fused floating point ops; multiply/add
Cycles from memory to stages in the cache
Pre-fetch and how many outstanding requests
Bandwidth from/to memory
Latency from memory/cache
Number of instructions issued per cycle

Five Parameters Associated with Memory Hierarchy
Access Time: Time for the CPU to fetch a value from memory – including delays through any intermediate levels.
Memory Size: The amount of memory of a given type in a system.
Cost Per Unit (byte): Cost per unit times size roughly equals total cost.
Transfer bandwidth: Units (bytes) per second transferred to the next level.
Unit of transfer: Number of units moved between adjacent levels in a single move.

Memory Hierarchy
Desired - no cost, very fast, large non-volatile
Actual Registers, cache, DRAM, Disk, tape, CD, Flash & EPROM
Registers - fast, local, volatile, VERY expensive, very small
Cache - fast, expensive, volatile, small
DRAM - medium size & speed, volatile
Distributed memory
Disk - slow, low cost per byte, non-volatile
Tape - very slow, very low cost, durable, non-volatile
CD - slow, “read only”, good data density, non-volatile
Flash & EPROM - small, not as fast a DRAM, non-volatile

C.2 Distributed-memory Dense BLAS

This document summarizes the discussions that took place during the meetings of the BLAS Technical Forum concerning the distributed-memory BLAS. The committee did not reach an agreement on how to specify an interface for such a set of routines, however it was felt that this document should keep a record of those discussions.

There has been much interest in the past few years in developing versions of the BLAS for distributed-memory computers [50, 28, 3, 29, 13, 17, 15, 18, 8, 51]. Some of this research proposed parallelizing the BLAS [45, 50, 17, 15, 11, 18, 49, 51], and some implemented a few important BLAS routines [50, 43, 17, 15, 11, 18, 49, 51], such as matrix-matrix multiplication [31, 5, 36, 44, 4, 16, 52] or triangular system solve [34, 40, 41, 10].

Based on this research work, it was agreed that an interface for the distributed-memory BLAS should have the following features:

- The calling sequence definitions should be simple and similar in all targeted programming languages.
- The interface should be effective for the developement of large and high-quality dense linear algebra software for distributed-memory computers.
- The interface should permit broad functionality to enable, facilitate and encourage the development of current and related research projects.

The main advantages of establishing a distributed-memory dense BLAS standard are portability and ease-of-use. In a distributed-memory environment in which the higher level routines and/or abstractions are built upon lower level message-passing and computational routines the benefits of standardization are particularly apparent. Furthermore, the definition of distributed-memory dense basic linear algebra subprograms provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.

The goal of the distributed-memory dense BLAS interface simply stated should be to develop a widely used standard for writing message-passing programs performing dense basic linear algebra operations. As such the interface should establish a practical, portable, efficient and flexible standard for distributed-memory dense basic linear algebra operations.

A complete list of goals follows.

- Design an Application Programming Interface (API) (not necessarily for compilers or a system implementation library) well suited for distributed-memory dense basic linear algebra computations.
- Allow efficient communication and computation: minimizing communication startup overhead and volume, while maximizing load balance and local computational performance.
- Allow for re-use of existing message-passing interface standard [30] as well as local basic linear algebra computational kernels such as the *de facto* standard BLAS.
- Allow for implementations that can be used in a heterogeneous environment.
- Define an interface that is not too different from current practice and provide extensions that allow greater flexibility.

- Define an interface that can be implemented on many vendors' platforms, with no significant changes in the underlying system software.
- Semantics of the interface should be language-and data-distribution independent.
- The interface should be designed to allow for thread-safety.

Such a distribute-memory BLAS standard should be intended for use by all those who want to write portable programs performing dense linear algebra operations in Fortran 77, Fortran 90, High Performance Fortran (HPF), C or C++. This includes individual application programmers, developers of dense linear algebra software designed to run on parallel machines, and creators of computational environment and tools. In order to be attrac- tive to this wide audience, the standard must provide a simple, easy-to-use interface for the basic user while not semantically precluding the high-performance computation and communication operations available on advanced computers.

The attractiveness of the distributed-memory dense BLAS at least partially stems from its wide portability as well as the common occurence of dense linear algebra operations in numerical simulations. These programs may run on distributed-memory multiprocessors, networks or clusters of workstations, and combinations of all of these. In addition, shared-memory implementations are possible. The message passing paradigm will not be made obsolete by architectures combining the shared- and distributed-memory views, or by increases in network speeds. It thus should be both possible and useful to implement such a standard on a great variety of machines, including those "machines", parallel or not, connected by a communication network.

The distributed-memory dense BLAS interface should provide many features intended to improve performance on scalable parallel computers with specialized interprocessor communication hardware. Thus, we expect that native, high-performance implementations of this interface could be provided on such machines. At the same time, implementations of such a standard on top of MPI or PVM will provide portability to workstation clusters and heterogeneous networks of workstations.

During the discussions it was agreed that the distributed-memory BLAS should include

- A set of basic dense linear algebra computational operations
- Data-redistribution operations
- Environmental management and inquiry
- Bindings for various widely used programming languages, including high level languages such as High Performance Fortran (HPF)

The distributed-memory dense BLAS interface should specify routines that operate on in-core dense matrices. On entry, these routines assume that the data has been distributed on the processors according to a specific data decomposition scheme that dictates the local storage of the data when it resides in the processors' memory. The data layout information as well as the local storage scheme for these different matrix operands is conveyed to the routines via a descriptor that could be a simple array of integers. The standard could mandate that the first entry of this array identifies the type of the descriptor, i.e., the data distribution scheme it describes. This allows to specify the distributed-memory dense BLAS interface indepen- dently from the data distribution.

The distributed-memory dense BLAS are executed by *processes*, rather than physical processors. In general there may be several processes running on a processor, in which case it is assumed that the runtime system handles the scheduling of processes. In the absence of such a runtime system, the distributed-memory dense BLAS assume one process per processor. A **process** is defined as a basic unit or thread of execution that minimally includes a stack, registers, and memory. Multiple processes may share a physical processor. The term processor refers to the actual hardware. Each process is treated as if it were a processor: a process executing a program or subprogram calling the distributed-memory dense BLAS must exist for the lifetime of the program's or subprogram's run. Its execution should affect other processes' execution only through the use of message-passing calls. With this in mind, the term process is used in all sections of this chapter unless otherwise specified.

The distributed-memory dense BLAS are thus executed by a collection of processes, that are enclosed in a **communication context**. Similarly, a communication context or simply context is associated with every global matrix. The use of a context provides the ability to have separate "universes" of message passing. This means that a collection of processes can safely communicate even if other (possibly overlapping) sets of processes are also communicating. Thus, a context is a powerful mechanism for avoiding unintentional nondeterminism in message passing and provides support for the design of safe, modular software libraries. In MPI, this concept is referred to as a *communicator*.

A context partitions the communication space. A message sent from one context cannot be received in another context. The use of separate communication contexts by distinct libraries (or distinct library routine invocations) insulates communication internal to a specific library routine from external communication that may be going on within the user's program.

In most respects, the terms *process collection* and *context* can be used interchangeably. For example, one may say that an operation is performed "in context X" or "in process collection X". The slight difference here is that the user may define two identical sets of processes (say, two 1×3 process grids, both of which use processes 0, 1, and 2), but each will be enclosed in its own context, so that they are distinct in operation, even though they are indistinguishable from a process collection standpoint.

Another example of the use of context might be to define a normal two-dimensional process grid within which most computation takes place. However, in certain portions of the code it may be more convenient to access the processes as a one-dimensional process grid, whereas at other times one may wish, for instance, to share information among nearest neighbors. In such cases, one will want each process to have access to three contexts: the two-dimensional process grid, the one-dimensional process grid, and a small process grid that contains the process and its nearest neighbors. Therefore, communication contexts allow one to

- create arbitrary groups of processes,
- create an indeterminate number of overlapping and/or disjoint collections of processes, and
- isolate a set of processes so that communication interference does not occur.

A distributed-memory dense BLAS function should create a grid of processes and its enclosing context. This routine returns a context handle, which is a simple integer, assigned by the message-passing library used by a given implementation of the distributed-memory dense BLAS to identify the commu- nication context. Subsequent distributed-memory dense BLAS will be passed these handles, which allow to determine from which context/process collec- tion a routine is being called. The user *should never alter or change these handles;* they are opaque data objects that are only meaningful for the distributed-memory dense BLAS routines.

A defined context consumes resources. It is therefore advisable to release contexts when they are no longer needed. When the entire distributed-memory dense BLAS system is shut down, all outstanding contexts are automatically freed.

Some systems, such as MPI, supply their own version of context. For portability reasons, one thus cannot assume that the communication contexts used by the user's program are usable by the distributed-memory dense BLAS. Therefore, the following interface allows to form a distributed-memory dense BLAS context in reference to a user's context.

The standard could mandate that the second entry of each descriptor is set to the value of the communication context identifying the collection of processes onto which the data is distributed.

The routines of the distributed-memory dense BLAS could require that all global data (vectors or matrices) be distributed across the processes prior to invoking the routines. The data layout and local storage scheme are specified by a particular implementation, but are strictly speaking not part of the standard. Global data is mapped to the local memories of processes assuming an implementation-dependent data distribution scheme.

A **descriptor** is associated with each global array. This descriptor stores the information required to establish the mapping between each global array entry and its corresponding process and memory location. Array descriptors corresponding to distinct layouts are differentiated by their first entry called the type of the descriptor. The data residing in the processes' memory is specified by a pointer. Splitting the local data from the layout's description allows to specify language-dependent interfaces for programming languages providing only simple data structures such as Fortran 77 without affecting the functionality or ease-of-use of the interface.

Most of the distributed-memory dense BLAS should operate within the same communication context, i.e., all distributed operands should be distributed on the same process grid. These operations are said to be **intra-context** operations. Only, very few distributed-memory dense BLAS perform inter-context operations, in which case this feature should clearly be mentioned in the procedure functionality.

Furthermore, all distributed operands involved in the operation should be distributed accordingly to the same decomposition scheme. In other words, the type entry of each descriptor must be equal.

All distributed-memory dense BLAS operations could be **collective**, that is, all processes in the context need to invoke the procedure even if certain processes are not involved in the operation. This situation may happen for example when some processes don't own any data to be operated on.

The distributed-memory dense BLAS manages **system memory** that is used for buffering messages and for storing internal representations of various distributed-memory dense BLAS objects such as communication contexts, local arrays of data, etc. This memory is not directly accessible to the user, and objects stored there are either private or opaque: their size and shape is not visible to the user. Opaque objects are accessed via **handles**, which exist in user space. Private objects cannot be accessed by the user, and are allocated and released within the same routine. distributed-memory dense BLAS that operate on opaque objects are passed handle arguments to access these objects. In addition to their use by distributed-memory dense BLAS calls for object access, handles can participate in assignment and comparisons.

In Fortran and C, all handles have type integer and correspond to the same objects. This means that the user's program can pass a C handle to a Fortran subprogram and conversely, such that in both languages the handle refers to the same object.

Opaque objects are allocated and deallocated by calls that are specific to each object type. These are listed in the sections where the object are described. The calls accept a handle argument of matching type. In an allocate call this is an ouput argument that returns a valid reference to the object. In a call to deallocate this is an input/output argument which returns with an "invalid handle" constant for each object type. Comparisons to this constant are used to test for validity of the handle.

A call to deallocate invalidates the handle and marks the object for deallocation. The object is not accesible to the user after the call. However, distributed-memory dense BLAS need not deallocate the object immediately. Any operation pending (at the time of the deallocate) that involves this object will complete normally; the object will be deallocated afterwards.

An opaque object and its handle are significant only at the process where the object was created, and cannot be transferred to another process.

This design hides the internal representation used for distributed-memory dense BLAS internal data structures, thus allowing similar calls in C and Fortran. It also avoids conflicts with the typing rules in these languages, and easily allows for future extensions of functionality. Note that the objects handles defined in the distributed-memory dense BLAS are exclusively used by the underlying message passing library. The user data itself remains directly accessible in the user's program.

The explicit separating of handles in user space, objects in system space, allows space-reclaiming, deallocation calls to be made at appropriate points in the user program. If the opaque objects were in user space, one would have to be very careful not to go out of scope before any pending operation requiring that object completed. The specified design allows an object to be marked for deallocation, the user program can then go out of scope, and the object itself still persists until any pending operations are complete. Again such a design cannot be applied in general to the user's data.

The requirement that handles support assignment/comparison is made since such operations are common. This restricts the domain of possible implementations. Moreover, such a design has been adopted by most message passing library interfaces such as the MPI.

The intended semantics of opaque objects is that each opaque object is separate from each other; each call to allocate such an object copies copies all the information required for that object. Implementations may avoid excessive copying by substituting referencing for copying. For example, a derived datatype may contain references to its components, rather then copies of its components. In such cases, the implementation must maintain reference counts, and allocate and deallocate objects such that the visible is as if the objects were copied. Such a design is particularly suitable for communication contexts, because the amount of data one has to keep track of is small. However, applying the same concept to the user's data forces the introduction of routines to manage logical templates, adding complexity, and was therefore ruled out.

There are several important language bindings issues not addressed by this document. This section does not discuss the interoperability of message passing between languages. It is fully expected that many implementations should have such features.

A descriptor is associated with each distributed matrix. The entries of the descriptor uniquely determine the mapping of the matrix entries onto the local processes' memories. Since vectors may be seen as a special case of distributed matrices or proper submatrices, the larger scheme just defined encompasses their description as well.

The local storage convention of the distributed matrix operands in every process's memory does not need to be specified by the standard. It is however recommended that convenient data structure are chosen by a given implementation allowing to rely on the sequential BLAS to perform the local computations within a process.

The distributed-memory dense BLAS should not provide mechanisms for dealing with failures in the communication and computation systems. If the distributed-memory dense BLAS is built on an unreliable underlying mechanism, then it is the job of the implementor(s) of the distributed-memory dense BLAS subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, such failures will be reflected as errors in the relevant communication or computation call.

Of course, distributed-memory dense BLAS programs can still be erroneous. A program error can occur when a distributed-memory dense BLAS routine is called with an invalid value for any of its arguments. The routine must report this fact and terminate the execution of the program. Each routine, on detecting an error, should call a common error-handling routine, passing to it the current communication context, the name of the routine and the number of the first argument that is in error. For efficiency purposes, the distributed-memory dense BLAS only perform a local validity check of their argument list. If an error is detected in at least one process of the current context, the program execution is stopped.

A global validity check of the input arguments passed to a distributed-memory dense BLAS routine must be performed in the user-level calling procedure. To demonstrate the need and cost of global checking, as well as the reason why this type of checking should not be performed in the distributed-memory dense BLAS, consider the following example: the value of a global input argument is legal but differs from one process to another. The results are unpredictable. In order to detect this kind of error situation, a synchronization point would be necessary, which may result in a significant performance degradation. Since every process must call the same routine to perform the desired operation successfully, it is natural and safe to restrict somewhat the amount of checking operations performed in the distributed-memory dense BLAS routines.

Specialized implementations may call system-specific exception-handling facilities, either via an auxiliary routine or directly from the routine. In addition, the testing programs can take advantage of this exception-handling mechanism by simulating specific erroneous input argument lists and then verifying that particular errors are correctly detected.

Resource errors can also occur when a program exceeds the amount of available system resources. The occurence of this type of error depends on the amount of available resources in the system and the resource allocation mechanism used; this may differ from system to system. A high-quality implementation will provide generous limits on the important resources so as to alleviate the portability problem this represents.

C.3 Fortran 95 Thin BLAS

C.3.1 Introduction

This paper presents a proposal for a specification of Fortran 95 thin BLAS.

A preliminary version of the F90 Blas proposal has been circulated informally (but not very widely) for about 4 years, and code which implements that version has been available in the Fortran 90 software repository maintained by NAG Ltd (**http://www.nag.co.uk**).

This proposal is designed to cover — as far as seems sensible — roughly the same functionality as the original Level 1, 2 and 3 (Fortran 77) BLAS. It does not address sparse matrices or vectors, nor does it explicitly address issues of parallel computation.

Many of the Fortran 77 Level 1 BLAS can be replaced by simple array expressions and assignments in Fortran 95, without loss of convenience or performance (assuming a reasonable degree of optimisation by the compiler). Fortran 95 also allows groups of related Level 2 and Level 3 BLAS to be merged together, each group being covered by a single interface.

C.3.2 Design of Fortran 95 Interfaces

Our proposed design utilizes the following features of the Fortran 95 language.

Generic interfaces: all routines are accessed through *generic* interfaces. A single generic name covers several specific instances whose arguments may differ in the following properties:

- **data type** (real or complex). However, the relevant arguments must be either all real or all complex. (We do not, for example, cater for multiplying a real matrix by a complex matrix, though this functionality could easily be added to the design if there was a need for it.)
- **precision** (or equivalently, kind-value). However, all real or complex arguments must have the same precision.
- **rank** Some arguments may either have rank 2 (to store a matrix) or rank 1 (to store a vector).
- **different argument list** Some of the arguments may not appear in a specific instance. In this case a pre-defined value or a pre-defined action is assumed. The following table contains the pre-defined value or action for the argument that may not be used. Some of these arguments are key arguments that will be described later.

argument	value or action if the argument is not used
alpha	1.0 or (1.0,0.0)
beta	0.0 or (0.0,0.0)
op_x	operate with x
lower	reference upper triangle only
right_side	operate on the left-hand side
unit_diag	non-unit triangular

Assumed-shape arrays: all array arguments are *assumed-shape* arrays, which must have the exact shape required to store the corresponding matrix or vector. Hence arguments to specify array-dimensions or problem-dimensions are not required. The routines assume that the supplied arrays have valid and consistent shapes (see Section C.3.5). Zero dimensions (implying empty arrays) are allowed.

Key arguments: in the Fortran 77 BLAS, we use character arguments to specify different options for the operation to be performed. In this proposal we suggest using key arguments. A key argument is a dummy argument whose actual argument must be a named constant defined by BLAS. The following table lists the key arguments, the related BLAS named constants and the equivalent F77 BLAS values.

dummy argument	named constant	meaning	F77 argument
op_x	*not used*	operate with x	TRANSx = 'N'
	blas_trans	operate with transpose x	TRANSx = 'T'
	blas_conj	operate with conjugate x	TRANSx = 'C'
	blas_conj_trans	operate with conjugate-transpose x	TRANSx = 'H'
lower	*not used*	reference upper triangle only	UPLO = 'U'
	blas_lower	reference lower triangle only	UPLO = 'L'
right_side	*not used*	operate on the left-hand side	SIDE = 'L'
	blas_right	operate on the right-hand side	SIDE = 'R'
unit_diag	*not used*	non-unit triangular	DIAG = 'N'
	blas_unit_diag	unit triangular	DIAG = 'U'

C.3.3 Interfaces for Real Data

The primary aim of this paper is to convey the flavour of the different generic interfaces.

Therefore we first describe the interfaces as they apply to *real* data. The extra complications which arise when they apply to complex data will be considered in Section C.3.4.

We summarize each interface in the form of a `subroutine` statement (or in one case a `function` statement), in which all the arguments might appear. (This is a convenient way to think of the interface, although such a statement using the generic interface name never appears in the code.) Arguments which need not be supplied are enclosed in square brackets, for example:

```
subroutine trmm( [alpha,] a, [op_a,] b, [lower,] [right_side,] [unit_diag] )
```

This is followed by a table which lists the different variants of the operation, depending either on the ranks of some of the arguments or on the key arguments.

The following table shows the values used in the tables and the related named constant for the key arguments.

dummy argument	value in table	named constant
op_x	'T'	blas_trans
	'C'	blas_conj
	'C/T'	blas_conj_trans
right_side	'R'	blas_right

Routines using conventional storage for matrices

By conventional storage, we mean storing a matrix in a 2-dimensional array,

```
subroutine gemm( [alpha,] a, [op_a,] b, [op_b], [beta,] c )
```

rank of `a`	rank of `b`	rank of `c`	`op_a`	`op_b`	operation	F77 BLAS
2	2	2			$C \leftarrow \alpha AB + \beta C$	`_GEMM`
2	2	2		'T'	$C \leftarrow \alpha AB^T + \beta C$	`_GEMM`
2	2	2	'T'		$C \leftarrow \alpha A^T B + \beta C$	`_GEMM`
2	2	2	'T'	'T'	$C \leftarrow \alpha A^T B^T + \beta C$	`_GEMM`
2	1	1			$c \leftarrow \alpha Ab + \beta c$	`_GEMV`
2	1	1	'T'		$c \leftarrow \alpha A^T b + \beta c$	`_GEMV`
1	1	2			$C \leftarrow \alpha ab^T + \beta C$	`_GER_`

```
subroutine symm( [alpha,] a, b, [beta,] c, [lower,] [right_side] )
subroutine hemm( [alpha,] a, b, [beta,] c, [lower,] [right_side] )
```

rank of `b`	rank of `c`	`right_side`	operation	F77 BLAS
2	2		$C \leftarrow \alpha AB + \beta C$	`_SYMM`
2	2	'R'	$C \leftarrow \alpha BA + \beta C$	`_SYMM`
1	1		$c \leftarrow \alpha Ab + \beta c$	`_SYMV`

where A is a symmetric matrix.

```
subroutine syrk( [alpha,] a, [op_a,] [beta,] c, [lower] )
subroutine herk( [alpha,] a, [op_a,] [beta,] c, [lower] )
```

rank of `a`	`op_a`	operation	F77 BLAS
2		$C \leftarrow \alpha AA^T + \beta C$	`_SYRK`
2	'T'	$C \leftarrow \alpha A^T A + \beta C$	`_SYRK`
1		$C \leftarrow \alpha aa^T + \beta C$	`_SYR1`

where C is a symmetric matrix.

```
subroutine syr2k( [alpha,] a, [op_a,] b, [beta,] c, [lower] )
subroutine he2rk( [alpha,] a, [op_a,] b, [beta,] c, [lower] )
```

rank of `a`	rank of `b`	`op_a`	operation	F77 BLAS
2	2		$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$	`_SYR2K`
2	2	'T'	$C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$	`_SYR2K`
1	1		$C \leftarrow \alpha ab^T + \alpha ba^T + \beta C$	`_SYR2`

where C is a symmetric matrix.

```
subroutine trmm( [alpha,] a, [op_a,] b, [lower,] [right_side,] [unit_diag] )
```

rank of `b`	`op_a`	`right_side`	operation	F77 BLAS
2			$B \leftarrow \alpha AB$	`_TRMM`
2	'T'		$B \leftarrow \alpha A^T B$	`_TRMM`
2		'R'	$B \leftarrow \alpha BA$	`_TRMM`
2	'T'	'R'	$B \leftarrow \alpha BA^T$	`_TRMM`
1			$b \leftarrow \alpha Ab$	`_TRMV`
1	'T'		$b \leftarrow \alpha A^T b$	`_TRMV`

```
subroutine trsm( [alpha,] a, [op_a,] b, [lower,] [right_side,] [unit_diag] )
```

rank of b	op_a	right_side	operation	F77 BLAS
2			$B \leftarrow \alpha A^{-1} B$	_TRSM
2	'T'		$B \leftarrow \alpha A^{-T} B$	_TRSM
2		'R'	$B \leftarrow \alpha B A^{-1}$	_TRSM
2	'T'	'R'	$B \leftarrow \alpha B A^{-T}$	_TRSM
1			$b \leftarrow \alpha A^{-1} b$	_TRSV
1	'T'		$b \leftarrow \alpha A^{-T} b$	_TRSV

Routines using packed storage for matrices

By *packed* storage, we mean storing the upper or lower triangle of a symmetric or triangular matrix in a 1-dimensional array (i.e. a vector).

```
subroutine spmv( [alpha,] a, b, [beta,] c, [lower] )
subroutine hpmv( [alpha,] a, b, [beta,] c, [lower] )
```

operation	F77 BLAS
$c \leftarrow \alpha A b + \beta c$	_SPMV

where A is a symmetric matrix.

```
subroutine spr1( [alpha,] a, [beta,] c, [lower] )
subroutine hpr1( [alpha,] a, [beta,] c, [lower] )
```

operation	F77 BLAS
$C \leftarrow \alpha a a^T + \beta C$	_SPR1

where C is a symmetric matrix.

```
subroutine syr2( [alpha,] a, b, [beta,] c, [lower] )
subroutine he2r( [alpha,] a, b, [beta,] c, [lower] )
```

operation	F77 BLAS
$C \leftarrow \alpha a b^T + \alpha b a^T + \beta C$	_SYR2

where C is a symmetric matrix.

```
subroutine tpmv( [alpha,] a, [op_a,] b, [lower,] [unit_diag] )
```

op_a	operation	F77 BLAS
	$b \leftarrow \alpha A b$	_TPMV
'T'	$b \leftarrow \alpha A^T b$	_TPMV

where A is a triangular matrix.

```
subroutine tpsv( [alpha,] a, [op_a,] b, [lower,] [unit_diag] )
```

op_a	operation	F77 BLAS
	$b \leftarrow \alpha A^{-1} b$	_TPSV
'T'	$b \leftarrow \alpha A^{-T} b$	_TPSV

where A is a triangular matrix.

Routines for band matrices

```
subroutine gbmv( [alpha,] a, [op_a,] b, [beta,] c, kd )
```

op_a	operation	F77 BLAS
	$C \leftarrow \alpha A b + \beta c$	_GBMV
'T'	$C \leftarrow \alpha A^T b + \beta c$	_GBMV

where A is a general band matrix with `kd` superdiagonals supplied.

```
subroutine sbmv( [alpha,] a, b, [beta,] c, [lower] )
subroutine hbmv( [alpha,] a, b, [beta,] c, [lower] )
```

operation	F77 BLAS
$c \leftarrow \alpha A b + \beta c$	_SPMV

where A is a symmetric band matrix.

```
subroutine tbmv( [alpha,] a, [op_a,] , [lower,] [unit_diag] )
```

op_a	operation	F77 BLAS
	$b \leftarrow \alpha A b$	_TBMV
'T'	$b \leftarrow \alpha A^T b$	_TBMV

where A is a triangular band matrix.

```
subroutine tpsv( [alpha,] a, [op_a,] , [lower,] [unit_diag] )
```

op_a	operation	F77 BLAS
	$b \leftarrow \alpha A^{-1} b$	_TBSV
'T'	$b \leftarrow \alpha A^{-T} b$	_TBSV

where A is a triangular band matrix.

Level 1 routines

```
function nrm2( x )
```

Operation: return $\|x\|_2$.

```
subroutine swap( x, y )
```

Operation: $x \leftrightarrow y$.

```
subroutine rot( x, y, c, s )
```

```
subroutine rotg( a, b, c, s )
```

C.3.4 Interfaces for Complex Data

In this section we show the subroutine `gemm` for complex arguments. The generic interface is that described for real arguments.

rank of a	rank of b	rank of c	op_a	op_b	operation	F77 BLAS
2	2	2			$C \leftarrow \alpha AB + \beta C$	_GEMM
2	2	2		'T'	$C \leftarrow \alpha AB^T + \beta C$	_GEMM
2	2	2		'C/T'	$C \leftarrow \alpha AB^H + \beta C$	_GEMM
2	2	2	'T'		$C \leftarrow \alpha A^T B + \beta C$	_GEMM
2	2	2	'T'	'T'	$C \leftarrow \alpha A^T B^T + \beta C$	_GEMM
2	2	2	'T'	'C/T'	$C \leftarrow \alpha A^T B^H + \beta C$	_GEMM
2	2	2	'C/T'		$C \leftarrow \alpha A^H B + \beta C$	_GEMM
2	2	2	'C/T'	'T'	$C \leftarrow \alpha A^H B^T + \beta C$	_GEMM
2	2	2	'C/T'	'C/T'	$C \leftarrow \alpha A^H B^H + \beta C$	_GEMM
2	1	1			$c \leftarrow \alpha Ab + \beta c$	_GEMV
2	1	1	'T'		$c \leftarrow \alpha A^T b + \beta c$	_GEMV
2	1	1	'C/T'		$c \leftarrow \alpha A^H b + \beta c$	_GEMV
1	1	2			$C \leftarrow \alpha ab^T + \beta C$	_GERU
1	1	2		'C'	$C \leftarrow \alpha ab^H + \beta C$	_GERC

C.3.5 Error checking

We propose that the Fortran 95 thin BLAS perform no checks on their arguments.

C.3.6 Comparison with the Fortran 77 BLAS

We consider in more detail each Level of BLAS in turn. In referring to particular operations performed by the BLAS, we use the traditional BLAS names, except that we omit the initial letter (S, D, C, Z) which indicates the data type — for example, `SWAP`. The resulting names are also the generic names which we propose for the Fortran 95 interfaces.

Level 1

We include in this proposal only the following:

```
SWAP
ROT
NRM2
ROTG
```

`ROT` and `ROTG` have been extended to cover complex rotations.

Level 2

We propose to combine many of the Level 2 BLAS with the corresponding Level 3 BLAS in a single generic interface, the different instances being distinguished by the ranks of some of the arguments. In order to do this, we propose to remove some minor inconsistencies between the specifications of the Level 2 and Level 3 routines:

We propose adding one new routine:

`REFG`

to generate an elementary reflector (that is, a Householder matrix), following the same specification as the LAPACK auxiliary routine xLARTG.

The scope of the proposed BLAS has been extended slightly compared with the Fortran 77 BLAS: for example, we propose Level 1 BLAS for generating an elementary reflector (Householder matrix), and for generating and applying complex plane rotations; we also propose Level 2 BLAS for complex symmetric matrices. On the other hand, many of the Fortran 77 Level 1 BLAS can be replaced in Fortran 95 by simple array constructs, and they have been omitted.

For the thin BLAS we propose that the code does not do any checks on the arguments.

We propose generic interfaces that cover — wherever relevant — both Level 2 and Level 3 BLAS (for example, xTRSV and xTRSM), and have modified the specification of some Level 2 BLAS to make them more consistent with the Level 3 BLAS (for example, xTRSV now has an argument `alpha`).

For each procedure we specify a number of arguments that must be supplied and another set of arguments that need not be supplied. We specify a value or action for each argument which need not be supplied.

We propose that the thin BLAS contain a specific instance for each possible case and no checks or branching is used within the code.

We propose that the early implementations for the thin BLAS will contain simple calls to the reliable and tested F77 BLAS.

For example, the generic `gemm` will consist of the following specific procedures:

- 36 specific procedures each of which calls the F77 BLAS procedure `ZGEMM` (3 settings for each of `op_a` and `op_b`, and 2 settings for each of `alpha` and `beta`).
- 12 specific procedures each of which calls the F77 BLAS procedure `ZGEMV` (3 settings for `op_a`, and 2 settings for each of `alpha` and `beta`).
- 4 specific procedures each of which calls the F77 BLAS procedure `ZGERU` (2 settings for each of `alpha` and `beta`).
- 4 specific procedures each of which calls the F77 BLAS procedure `ZGERC` (2 settings for each of `alpha` and `beta`).
- 36 specific procedures each of which calls the F77 BLAS procedure `DGEMM` (3 settings for each of `op_a` and `op_b`, and 2 settings for each of `alpha` and `beta`). Only 16 procedures are needed, but we allow for `op_a = blas_conj_trans` for similarity with the complex case.
- 12 specific procedures each of which calls the F77 BLAS procedure `DGEMV` (3 settings for `op_a`, and 2 settings for each of `alpha` and `beta`). Only 8 procedures are needed, but we allow for `op_a = blas_conj_trans` for similarity with the complex case.
- 4 specific procedures each of which calls the F77 BLAS procedure `DGER` (2 settings for each of `alpha` and `beta`).
- 4 specific procedures each of which calls the F77 BLAS procedure `DGER` (2 settings for each of `alpha` and `beta`). These are similar to the previous case but have been added to allow `op_b = blas_conj` (as in the complex case for `ZGERC`).

Appendix C.3.8 contains a list of these specific procedures (only double precision procedures are listed).

A proposed document for this procedure is given in a separate document.

C.3.7 Conclusion

Our principal purpose in presenting this specification at this meeting is to provide additional input to the discussion about different levels of genericity in the interface to linear algebra routines. The thin BLAS are designed principally as building-blocks for software developers and for the BLAS itself.

C.3.8 Further Details: Specific procedures for gemm

This appendix contains a list of the specific procedures for the generic procedure gemm.

```
!
! 36 procedures each calls the F77 BLAS subroutine ZGEMM
!  a, b and c are rank-2
!
!            alpha      op_a       a     op_b        b beta c   operation
!
! zgemm_301 (alpha,blas_conj_trans,a,blas_conj_trans,b,beta,c)  C < alpha A(H) B(H) + beta C
! zgemm_302 (alpha,blas_conj_trans,a,blas_conj_trans,b,     c)  C < alpha A(H) B(H) + C
! zgemm_303 (alpha,blas_conj_trans,a,blas_trans     ,b,beta,c)  C < alpha A(H) B(T) + beta C
! zgemm_304 (alpha,blas_conj_trans,a,blas_trans     ,b,     c)  C < alpha A(H) B(T) + C
! zgemm_305 (alpha,blas_conj_trans,a,                b,beta,c)  C < alpha A(H) B + beta C
! zgemm_306 (alpha,blas_conj_trans,a,                b,     c)  C < alpha A(H) B + C
! zgemm_307 (alpha,blas_trans     ,a,blas_conj_trans,b,beta,c)  C < alpha A(T) B(H) + beta C
! zgemm_308 (alpha,blas_trans     ,a,blas_conj_trans,b,     c)  C < alpha A(T) B(H) + C
! zgemm_309 (alpha,blas_trans     ,a,blas_trans     ,b,beta,c)  C < alpha A(T) B(T) + beta C
! zgemm_310 (alpha,blas_trans     ,a,blas_trans     ,b,     c)  C < alpha A(T) B(T) + C
! zgemm_311 (alpha,blas_trans     ,a,                b,beta,c)  C < alpha A(T) B + beta C
! zgemm_312 (alpha,blas_trans     ,a,                b,     c)  C < alpha A(T) B + C
! zgemm_313 (alpha,                a,blas_conj_trans,b,beta,c)  C < alpha A B(H) + beta C
! zgemm_314 (alpha,                a,blas_conj_trans,b,     c)  C < alpha A B(H) + C
! zgemm_315 (alpha,                a,blas_trans     ,b,beta,c)  C < alpha A B(T) + beta C
! zgemm_316 (alpha,                a,blas_trans     ,b,     c)  C < alpha A B(T) + C
! zgemm_317 (alpha,                a,                b,beta,c)  C < alpha A B + beta C
! zgemm_318 (alpha,                a,                b,     c)  C < alpha A B + C
! zgemm_319 (      blas_conj_trans,a,blas_conj_trans,b,beta,c)  C < A(H) B(H) + beta C
! zgemm_320 (      blas_conj_trans,a,blas_conj_trans,b,     c)  C < A(H) B(H) + C
! zgemm_321 (      blas_conj_trans,a,blas_trans     ,b,beta,c)  C < A(H) B(T) + beta C
! zgemm_322 (      blas_conj_trans,a,blas_trans     ,b,     c)  C < A(H) B(T) + C
! zgemm_323 (      blas_conj_trans,a,                b,beta,c)  C < A(H) B + beta C
! zgemm_324 (      blas_conj_trans,a,                b,     c)  C < A(H) B + C
! zgemm_325 (      blas_trans     ,a,blas_conj_trans,b,beta,c)  C < A(T) B(H) + beta C
! zgemm_326 (      blas_trans     ,a,blas_conj_trans,b,     c)  C < A(T) B(H) + C
! zgemm_327 (      blas_trans     ,a,blas_trans     ,b,beta,c)  C < A(T) B(T) + beta C
! zgemm_328 (      blas_trans     ,a,blas_trans     ,b,     c)  C < A(T) B(T) + C
! zgemm_329 (      blas_trans     ,a,                b,beta,c)  C < A(T) B + beta C
! zgemm_330 (      blas_trans     ,a,                b,     c)  C < A(T) B + C
! zgemm_331 (                      a,blas_conj_trans,b,beta,c)  C < A B(H) + beta C
! zgemm_332 (                      a,blas_conj_trans,b,     c)  C < A B(H) + C
! zgemm_333 (                      a,blas_trans     ,b,beta,c)  C < A B(T) + beta C
! zgemm_334 (                      a,blas_trans     ,b,     c)  C < A B(T) + C
! zgemm_335 (                      a,                b,beta,c)  C < A B + beta C
! zgemm_336 (                      a,                b,     c)  C < A B + C
!
!
! 12 procedures each calls the F77 BLAS subroutine ZGEMV
!  a is rank-2, and b and c are rank-1
!
!            alpha      op_a       a     op_b        b beta c   operation
!
! zgemv_201 (alpha,blas_conj_trans,a,                b,beta,c)  c < alpha A(H) b + beta c
! zgemv_202 (alpha,blas_conj_trans,a,                b,     c)  c < alpha A(H) b + c
! zgemv_203 (alpha,blas_trans     ,a,                b,beta,c)  c < alpha A(T) b + beta c
! zgemv_204 (alpha,blas_trans     ,a,                b,     c)  c < alpha A(T) b + c
! zgemv_205 (alpha,                a,                b,beta,c)  c < alpha A b + beta c
! zgemv_206 (alpha,                a,                b,     c)  c < alpha A b + c
! zgemv_207 (      blas_conj_trans,a,                b,beta,c)  c < A(H) b + beta c
! zgemv_208 (      blas_conj_trans,a,                b,     c)  c < A(H) b + c
! zgemv_209 (      blas_trans     ,a,                b,beta,c)  c < A(T) b + beta c
! zgemv_200 (      blas_trans     ,a,                b,     c)  c < A(T) b + c
! zgemv_211 (                      a,                b,beta,c)  c < A b + beta c
! zgemv_212 (                      a,                b,     c)  c < A b + c
!
!
! 4 procedures each calls the F77 BLAS subroutine ZGERU
```

```
!  c is rank-2, and a and b are rank-1
!
!            alpha      op_a       a     op_b        b beta c   operation
!
! zgeru_201 (alpha,                a,                b,beta,c)  C < alpha a b(T) + beta C
! zgeru_202 (alpha,                a,                b,     c)  C < alpha a b(T) + C
! zgeru_203 (                      a,                b,beta,c)  C < a b(T) + beta C
! zgeru_204 (                      a,                b,     c)  C < a b(T) + C
!
!
! 4 procedures each calls the F77 BLAS subroutine ZGERC
!  c is rank-2, and a and b are rank-1
!
!            alpha      op_a       a     op_b        b beta c   operation
!
! zgerc_201 (alpha,blas_conj       a,                b,beta,c)  C < alpha a b(H) + beta C
! zgerc_202 (alpha,blas_conj       a,                b,     c)  C < alpha a b(H) + C
! zgerc_203 (      blas_conj       a,                b,beta,c)  C < a b(H) + beta C
! zgerc_204 (      blas_conj       a,                b,     c)  C < a b(H) + C
!
!
!
! 36 procedures each calls the F77 BLAS subroutine DGEMM
!  a, b and c are rank-2
!
!            alpha      op_a       a     op_b        b beta c   operation
!
! dgemm_301 (alpha,blas_conj_trans,a,blas_conj_trans,b,beta,c)  C < alpha A(H) B(H) + beta C
! dgemm_302 (alpha,blas_conj_trans,a,blas_conj_trans,b,     c)  C < alpha A(H) B(H) + C
! dgemm_303 (alpha,blas_conj_trans,a,blas_trans     ,b,beta,c)  C < alpha A(H) B(T) + beta C
! ...
! ...
! dgemm_334 (                      a,blas_trans     ,b,     c)  C < A B(T) + C
! dgemm_335 (                      a,                b,beta,c)  C < A B + beta C
! dgemm_336 (                      a,                b,     c)  C < A B + C
!
!
! 12 procedures each calls the F77 BLAS subroutine DGEMV
!  a is rank-2, and b and c are rank-1
!
!            alpha      op_a       a     op_b        b beta c   operation
!
! dgemv_201 (alpha,blas_conj_trans,a,                b,beta,c)  c < alpha A(H) b + beta c
! dgemv_202 (alpha,blas_conj_trans,a,                b,     c)  c < alpha A(H) b + c
! ...
! dgemv_211 (                      a,                b,beta,c)  c < A b + beta c
! dgemv_212 (                      a,                b,     c)  c < A b + c
!
!
! 8 procedures each calls the F77 BLAS subroutine DGER
!  c is rank-2, and a and b are rank-1
!
!            alpha      op_a       a     op_b        b beta c   operation
!
! dger_201  (alpha,                a,                b,beta,c)  C < alpha a b(T) + beta C
! dger_202  (alpha,                a,                b,     c)  C < alpha a b(T) + C
! dger_203  (                      a,                b,beta,c)  C < a b(T) + beta C
! dger_204  (                      a,                b,     c)  C < a b(T) + C
! dger_205  (alpha,blas_conj       a,                b,beta,c)  C < alpha a b(H) + beta C
! dger_206  (alpha,blas_conj       a,                b,     c)  C < alpha a b(H) + C
! dger_207  (      blas_conj       a,                b,beta,c)  C < a b(H) + beta C
! dger_208  (      blas_conj       a,                b,     c)  C < a b(H) + C
```

Procedure gemm

Description

`gemm` is a generic procedure which performs one of following operations:

rank of `a`	rank of `b`	rank of `c`	`op_a`	`op_b`	operation	F77 BLAS
2	2	2			$C \leftarrow \alpha AB + \beta C$	`_GEMM`
2	2	2		'T'	$C \leftarrow \alpha AB^T + \beta C$	`_GEMM`
2	2	2		'C/T'	$C \leftarrow \alpha AB^H + \beta C$	`_GEMM`
2	2	2	'T'		$C \leftarrow \alpha A^T B + \beta C$	`_GEMM`
2	2	2	'T'	'T'	$C \leftarrow \alpha A^T B^T + \beta C$	`_GEMM`
2	2	2	'T'	'C/T'	$C \leftarrow \alpha A^T B^H + \beta C$	`_GEMM`
2	2	2	'C/T'		$C \leftarrow \alpha A^H B + \beta C$	`_GEMM`
2	2	2	'C/T'	'T'	$C \leftarrow \alpha A^H B^T + \beta C$	`_GEMM`
2	2	2	'C/T'	'C/T'	$C \leftarrow \alpha A^H B^H + \beta C$	`_GEMM`
2	1	1			$c \leftarrow \alpha Ab + \beta c$	`_GEMV`
2	1	1	'T'		$c \leftarrow \alpha A^T b + \beta c$	`_GEMV`
2	1	1	'C/T'		$c \leftarrow \alpha A^H b + \beta c$	`_GEMV`
1	1	2			$C \leftarrow \alpha ab^T + \beta C$	`_GER_`
1	1	2		'C'	$C \leftarrow \alpha ab^H + \beta C$	`_GER_`

(If A is real, then $A^H = A^T$.)

Usage

```
CALL gemm([alpha], [op_a], a, [op_b], b, [beta], c)
```

One or more of the arguments in square brakets can be dropped. The order of the supplied arguments must remain unchanged.

Interfaces

Distinct interfaces are provided for each of the combinations of the following cases:

Real / complex data

Real data: `alpha`, `a`, `b`, `beta` and `c` are of type real(kind=wp).

Complex data: `alpha`, `a`, `b`, `beta` and `c` are of type complex(kind=wp).

different ranks

f77_gemm: `a`, `b` and `c` are rank-2 arrays.

f77_gemv: `a` is a rank-2 array while `b` and `c` are rank-1 arrays.

f77_ger: `c` is a rank-2 array while `a` and `b` are rank-1 arrays.

Arguments

All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '`x`(n)' is used in the argument descriptions to specify that the array `x` must have exactly n elements.

The procedure derives the values of the following problem parameters from the shape of the supplied arrays.

m — the first dimension of `c`, if `c` is rank-2 (m =`SIZE(c,1)`), or the size of `c` if it is rank-1 (m =`SIZE(c)`)

n — the second dimension of `c` if it is rank-2 (n =`SIZE(c,2)`)

k — the intermediate dimension

Mandatory arguments

One or more of the arguments `alpha`, `op_a`, `op_b` and `beta` can be dropped. The order of the supplied arguments must remain unchanged.

alpha — real(kind=wp)/ complex(kind=wp), intent(in)

Input: the value of α if different from one.

Note: if α is exactly one, you need not supply this argument.

op_a — a "key" argument, intent(in)

Input: if `op_a` is supplied, it specifies whether the operation involves the transpose A^T or its conjugate-transpose A^H ($= A^T$ if A is real). In this case `op_a` must have one of the following values (which are named constants, each of a different derived type, defined by the BLAS, and accessible from the module `blas`):

`blas_trans`: if the operation involves the transpose A^T rather than the matrix A;

`blas_conj_trans`: if the operation involves the conjugate-transpose A^H rather than the matrix A.

For further explanation of "key" arguments, see the ????.

Note: for real matrices, `blas_conj_trans` is equivalent to `blas_trans`.

Constraints: `op_a` *must not* be supplied if `a` is rank-1 or if the operation does not involve the transpose or the conjugate-transpose of A.

a(m) / **a**(p,q) — real(kind=*wp*)/ complex(kind=*wp*), intent(in)

Input: the matrix A or vector a.
If `a` is rank-2 then:

if `op_a` is not supplied, the shape of `a` must be (m,k);

if `op_a` is supplied, the shape of `a` must be (k,m).

op_b — a "key" argument, intent(in)

Input: if `op_b` is supplied and `b` is rank-2, it specifies whether the operation involves the transpose B^T or its conjugate-transpose B^H ($= B^T$ if B is real). In this case `op_b` must have one of the following values (which are named constants, each of a different derived type, defined by the BLAS, and accessible from the module `blas`):

`blas_trans`: if the operation involves the transpose B^T rather than the matrix B;

`blas_conj_trans`: if the operation involves the conjugate-transpose B^H rather than the matrix B.

If `op_b` is supplied and `b` is rank-1, it specifies that the operation involves the conjugate of b^T (b^H) rather than b^T. In this case `op_b` must have be `blas_conj` (which is a named constant of a derived type, defined by the BLAS, and accessible from the module `blas`).

For further explanation of "key" arguments, see the ????.

Note: for real matrices, `blas_conj_trans` is equivalent to `blas_trans`. For real arrays `blas_conj` does not have any effect.

Constraints: `op_b` *must not* be supplied if the operation does not involve the conjugate of b, the transpose of B or the conjugate-transpose of B.

b(r) / **b**(r,s) — real(kind=*wp*)/ complex(kind=*wp*), intent(in)

Input: the matrix B or vector b.
If `b` is rank-1 then:

if `a` is rank-1, the shape of `b` must be (m);

if `a` is rank-2, the shape of `b` must be `SIZE(op_a(a),2)`.

If `b` is rank-2 then:

if `op_b` is not supplied, the shape of `b` must be (k,n);

if `op_b` is supplied, the shape of `b` must be (n,k).

beta — real(kind=*wp*)/ complex(kind=*wp*), intent(in)

Input: the value of β if different from zero.

Note: if β is exactly zero, you need not supply this argument.

c(m) / **c**(m,n) — real(kind=*wp*)/ complex(kind=*wp*), intent(inout)

Input: the matrix C or vector c. If `beta` is not supplied `c` need not be initialized.

Output: the matrix C or vector c after applying the operation.

Examples of usage

One or more of the arguments `alpha`, `op_a`, `op_b` and `beta` can be dropped. The order of the supplied arguments must remain unchanged.

To perform $C \leftarrow \alpha AB^H$ use the call:

```
CALL gemm (alpha,a,blas_conj_trans,b,c)
```

To perform $C \leftarrow AB + \beta C$ use the call:

```
CALL gemm (a,b,beta,c)
```

To perform $C \leftarrow \alpha AB + \beta C$ use the call:

```
CALL gemm (alpha,a,b,beta,c)
```

To perform $C \leftarrow A^T B^H + \beta C$ use the call:

```
CALL gemm (blas_trans,a,blas_her,b,beta,c)
```

To perform $c \leftarrow \alpha A^T b + \beta c$ use the call:

```
CALL gemm (alpha,blas_trans,a,b,beta,c)
```

To perform $C \leftarrow ab^H + \beta C$ use the call:

```
CALL gemm (a,b,blas_conj,beta,c)
```

C.4 Interval BLAS

C.4.1 Introduction

Interval computation refers to performing computations with intervals. Computing with intervals guarantees that interval results contain the set of all possible correct answers. Valid implementations of interval arithmetic produce correct bounds on the set of all possible correct answers, including the effects of accumulated roundoff errors. Recent advances in interval algorithms have generated interest in using these methods in increasing numbers of applications. This motivates us to establish the standard for interval BLAS described in this chapter.

Intervals

A nonempty *mathematical interval* $[a, b]$ is the set $\{x \in \Re | a \leq x \leq b\}$ where $a \leq b$. A *machine interval* $[a^*, b^*]$ is a mathematical interval whose endpoints are machine representable numbers. We say that $[a^*, b^*]$ is a machine representation of $[a, b]$ if $[a^*, b^*]$ *contains* $[a, b]$ i.e. $a^* \leq a$ and $b \leq b^*$. We say that the machine interval $[a^*, b^*]$ is a *tight representation* of a mathematical interval $[a, b]$ if and only if a^* is the greatest machine representable number which is less than or equal to a, and b^* is the least machine representable number which is greater than or equal to b.
The *empty interval* $\emptyset$, which does not contain any real number, is required in the interval BLAS. For machines in compliance with the IEEE-standard, we recommend the use of [NaN_empty, NaN_empty] to represent the empty interval, where NaN_empty is a unique non-default quiet not-a-number that is used to represent the empty interval *only.*
interval.

Notation

Both scalar (floating point number) and interval arguments are used for the specification of routines in this chapter. *Interval vectors* and *interval matrices* are vectors and matrices whose entries are intervals. The notation used in this chapter is consistent with other chapters, but we use **boldface letters** to specify interval arguments. We also use overline and underline to specify the greatest lower bound and the least upper bound of an interval variable, respectively. For example, if $\mathbf{x}$ is an interval vector, then $\mathbf{x} = [\underline{x}, \overline{x}]$.

Interval arithmetic

Interval arithmetic on mathematical intervals is defined as follows.

> *Let* $\mathbf{a}$ *and* $\mathbf{b}$ *be two mathematical intervals. Let* op *be one of the arithmetic operations* $+, -, \times, \div$. *Then* $\mathbf{a}$ op $\mathbf{b} \equiv \{a \text{ op } b : a \in \mathbf{a}, b \in \mathbf{b}\}$, *provided that* $0 \notin \mathbf{b}$ *if* op *represents* $\div$.
>
> *Advice to users:* The above definition of division implies that the user is responsible to trapping and dealing with any division by an interval containing zero.

Table C.1 gives explicit implementations of these four basic interval arithmetic operations and other operations on mathematical intervals used in this chapter. We use the notation $\mathbf{a} = [\underline{a}, \overline{a}]$ and $\mathbf{b} = [\underline{b}, \overline{b}]$.
All operations in the interval BLAS are necessarily performed on machine intervals. Arithmetic on machine intervals must satisfy the following property:

Operation	$\mathbf{a} \neq \emptyset$ and $\mathbf{b} \neq \emptyset$	$\mathbf{a} = \emptyset$ or/and $\mathbf{b} = \emptyset$
Addition $\mathbf{a} + \mathbf{b}$	$[\underline{a} + \underline{b}, \overline{a} + \overline{b}]$	$\emptyset$
Subtraction $\mathbf{a} - \mathbf{b}$	$[\underline{a} - \overline{b}, \overline{a} - \underline{b}]$	$\emptyset$
Multiplication $\mathbf{a} * \mathbf{b}$	$[\min\{\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}\}, \max\{\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}\}]$	$\emptyset$
Cancellation $\mathbf{a} \ominus \mathbf{b}$	$[\underline{a} - \underline{b}, \overline{a} - \overline{b}]$ if $(\underline{a} - \underline{b}) \leq (\overline{a} - \overline{b})$; Otherwise, $\emptyset$	$\emptyset$
Division $\frac{\mathbf{a}}{\mathbf{b}}, (0 \notin \mathbf{b})$	$[\min\{\frac{\underline{a}}{\underline{b}}, \frac{\underline{a}}{\overline{b}}, \frac{\overline{a}}{\underline{b}}, \frac{\overline{a}}{\overline{b}}\}, \max\{\frac{\underline{a}}{\underline{b}}, \frac{\underline{a}}{\overline{b}}, \frac{\overline{a}}{\underline{b}}, \frac{\overline{a}}{\overline{b}}\}]$	$\emptyset$
Convex Hull $\mathbf{a}, \mathbf{b}$	$[\min\{\underline{a}, \underline{b}\}, \max\{\overline{a}, \overline{b}\}]$	$\mathbf{b}$ if $\mathbf{a} = \emptyset$; or $\mathbf{a}$ if $\mathbf{b} = \emptyset$
Intersection $\mathbf{a} \cap \mathbf{b}$	$[\max\{\underline{a}, \underline{b}\}, \min\{\overline{a}, \overline{b}\}]$ if $\max\{\underline{a}, \underline{b}\} \leq \min\{\overline{a}, \overline{b}\}$; Otherwise, $\emptyset$	$\emptyset$
Disjoint	*True* if $\mathbf{a} \cap \mathbf{b} = \emptyset$; *False*, otherwise	*True*
Absolute value $\|\mathbf{a}\|$	$\max\{\lvert\underline{a}\rvert, \lvert\overline{a}\rvert\}$	NaN_empty
Midpoint $\mathbf{a}$	$(\underline{a} + \overline{a})/2$	NaN_empty
Width $\mathbf{a}$	$\overline{a} - \underline{a}$	NaN_empty

Table C.1: Elementary interval operations

> *Containment Condition:* Let $\mathbf{a} = [\underline{a}, \overline{a}]$ and $\mathbf{b} = [\underline{b}, \overline{b}]$ be intervals. Let $\mathbf{c} = [\underline{c}, \overline{c}]$ be the interval result of computing $\mathbf{a}$ `op` $\mathbf{b}$ where `op` is defined in Table C.1. If $\mathbf{c}$ is nonempty, then $\mathbf{c}$ must contain the exact mathematical interval $\mathbf{a}$ `op` $\mathbf{b}$.

In other words, interval arithmetic on nonempty machine intervals requires that we round down the lower bound and round up the upper bound to guarantee that the machine interval result contains the true mathematical interval result. This is needed to propagate guaranteed error bounds. A good implementation will round down and round up to the nearest possible floating point numbers, in order to get the narrowest possible machine intervals. But coarser rounding is enough to get a correct implementation. For more information on interval arithmetic specifications, one may refer to [14].

> *Advice to implementors:* In implementations of interval BLAS, a warning message should be provided to users whenever there is no finite machine interval that satisfies the containment condition during computations.

With interval arithmetic, one may automatically bound truncation error, round-off error, and even error in the original data to obtain machine intervals that are guaranteed to contain the true mathematical result of a computation. However, simply changing floating point numbers in an algorithm into intervals and all floating point operations into interval operations may result in such wide intervals that the output is useless in practice. For example, $[-100, 200]$ is a correct but probably useless bound for a true result of 3.1416. To apply the interval BLAS routines effectively, appropriate algorithms should be used that attempt to keep interval widths narrow. Many such algorithms are available in the literature. Readers may find a list of reference books, websites, software packages, and applications in [2, 12, 1].

C.4.2 Functionality

This chapter defines the functionality and language bindings for both the interval BLAS routines, and for selected mathematical operations on: intervals; interval vectors; and, dense, banded, and triangular interval matricies. Neither sparse data structures, nor complex intervals are treated.

Sections C.4.2 – C.4.2 outline the functionality of the proposed routines in tabular form. Sections C.2–C.4.5 present the language bindings for the proposed routines in the functionality tables.

Interval Vector Operations

Table C.2 lists interval vector reduction operations. Table C.3 lists interval vector operations. Table C.4 lists interval vector operations which involve only data movement.

Dot product	$\mathbf{r} \leftarrow \beta\mathbf{r} + \alpha\mathbf{x}^T\mathbf{y}$	DOT_I
Vector norms	$r \leftarrow \|\|\mathbf{x}\|\|_1, r \leftarrow \|\|\mathbf{x}\|\|_2$	
	$r \leftarrow \|\|\mathbf{x}\|\|_\infty$	NORM_I
Sum	$\mathbf{r} \leftarrow \sum_i \mathbf{x}_i$	SUM_I
Max magnitude & location	$k, \mathbf{x}_k;\ k = \arg\ \max_i\{\|\underline{x}_i\|, \|\overline{x}_i\|\}$	AMAX_VAL_I
Min absolute value & location	$k, \mathbf{x}_k;\ k = \arg\ \min_i\{\|\underline{x}_i\|, \|\overline{x}_i\|\}$	AMIN_VAL_I
Sum of squares	$(\mathbf{a}, \mathbf{b}) \leftarrow \sum_i \mathbf{x}_i^2,\ \mathbf{a} \cdot \mathbf{b}^2 = \sum_i \mathbf{x}_i^2$	SUMSQ_I

Table C.2: Reduction Operations

Reciprocal scale	$\mathbf{x} \leftarrow \mathbf{x}/\alpha$	RSCALE_I
Scaled interval vector accumulation	$\mathbf{y} \leftarrow \alpha\mathbf{x} + \beta\mathbf{y}$	AXPBY_I
Scaled interval vector accumulation	$\mathbf{w} \leftarrow \alpha\mathbf{x} + \beta\mathbf{y}$	WAXPBY_I
Scaled interval vector cancellation	$\mathbf{y} \leftarrow \alpha\mathbf{x} \ominus \beta\mathbf{y}$	CANCEL_I
Scaled interval vector cancellation	$\mathbf{w} \leftarrow \alpha\mathbf{x} \ominus \beta\mathbf{y}$	WCANCEL_I

Table C.3: Interval Vector Operations

Copy	$\mathbf{y} \leftarrow \mathbf{x}$	COPY_I
Swap	$\mathbf{y} \leftrightarrow \mathbf{x}$	SWAP_I
Permute vector	$\mathbf{x} \leftarrow P\mathbf{x}$	PERMUTE_I

Table C.4: Data Movement with Interval Vector Operations

Interval Matrix-Vector Operations

Table C.5 lists interval matrix-vector operations.

Matrix vector product	$\mathbf{y} \leftarrow \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$	GE,GB,SY,SB,SP	MV_I
	$\mathbf{y} \leftarrow \alpha\mathbf{A}^T\mathbf{x} + \beta\mathbf{y}$	GE,GB	MV_I
	$\mathbf{x} \leftarrow \mathbf{T}\mathbf{x}, \mathbf{x} \leftarrow \mathbf{T}^T\mathbf{x}$	TR, TB, TP	MV_I
Triangular solve	$\mathbf{x} \leftarrow \alpha\mathbf{T}^{-1}\mathbf{x}, \mathbf{x} \leftarrow \alpha\mathbf{T}^{-T}\mathbf{x}$	TR, TB, TP	SV_I
Rank one updates	$\mathbf{A} \leftarrow \alpha\mathbf{x}\mathbf{y}^T + \beta\mathbf{A}$	GE,SY,SP	R_I

Table C.5: Interval Matrix-vector Operations

Interval Matrix Operations

Table C.6 lists single interval matrix operations and interval matrix operations that involve $O(n^2)$ floating point operations. The matrix $\mathbf{T}$ represents an upper or lower triangular interval matrix. $\mathbf{D}$ represents a diagonal interval matrix. Table C.7 lists the interval matrix-matrix operations that involve $O(n^3)$ floating point operations and Table C.8 lists those operations that involve only data movement.

Matrix norms	$r \leftarrow \|\|\mathbf{A}\|\|_1, r \leftarrow \|\|\mathbf{A}\|\|_F,$	GE,GB,SY,SB,	_NORM_I
	$r \leftarrow \|\|\mathbf{A}\|\|_\infty, r \leftarrow \|\|\mathbf{A}\|\|_{\max}$	SP,TR,TB,TP	
Diagonal scaling	$\mathbf{A} \leftarrow \mathbf{D}\mathbf{A}, \mathbf{A} \leftarrow \mathbf{A}\mathbf{D}$	GE, GB	_DIAG_SCALE_I
Two sided diagonal scaling	$\mathbf{A} \leftarrow \mathbf{D}_1\mathbf{A}\mathbf{D}_2$	GE, GB	_LRSCALE_I
Two sided diagonal scaling	$\mathbf{A} \leftarrow \mathbf{D}\mathbf{A}\mathbf{D}$	SY, SB, SP	_LRSCALE_I
	$\mathbf{A} \leftarrow \mathbf{A} + \mathbf{B}\mathbf{D}$	GE, GB	
Matrix acc and scale	$\mathbf{B} \leftarrow \alpha\mathbf{A} + \beta\mathbf{B},$	GE,GB,SY,SB,	_ACC_I
	$\mathbf{B} \leftarrow \alpha\mathbf{A}^T + \beta\mathbf{B}$	SP,TR,TB,TP	
Matrix add and scale	$\mathbf{C} \leftarrow \alpha\mathbf{A} + \beta\mathbf{B}$	GE,GB,SY,SB,	_ADD_I
		SP,TR,TB,TP	

Table C.6: Matrix Operations – $O(n^2)$ floating point operations

Matrix matrix product	$\mathbf{C} \leftarrow \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C}$, $\mathbf{C} \leftarrow \alpha\mathbf{A}^T\mathbf{B} + \beta\mathbf{C}$,	GE,GB,SY,SB	MM_I
	$\mathbf{C} \leftarrow \alpha\mathbf{A}\mathbf{B}^T + \beta\mathbf{C}$, $\mathbf{C} \leftarrow \alpha\mathbf{A}^T\mathbf{B}^T + \beta\mathbf{C}$		
	$\mathbf{C} \leftarrow \alpha\mathbf{B}\mathbf{A} + \beta\mathbf{C}$, $\mathbf{C} \leftarrow \alpha\mathbf{B}^T\mathbf{A} + \beta\mathbf{C}$,	GB	MM_I
	$\mathbf{C} \leftarrow \alpha\mathbf{B}\mathbf{A}^T + \beta\mathbf{C}$, $\mathbf{C} \leftarrow \alpha\mathbf{B}^T\mathbf{A}^T + \beta\mathbf{C}$		
Triangular multiply	$\mathbf{B} \leftarrow \alpha\mathbf{T}\mathbf{B}$, $\mathbf{B} \leftarrow \alpha\mathbf{B}\mathbf{T}$	TR, TB	MM_I
	$\mathbf{B} \leftarrow \alpha\mathbf{T}^T\mathbf{B}$, $\mathbf{B} \leftarrow \alpha\mathbf{B}\mathbf{T}^T$		
Triangular solve	$\mathbf{B} \leftarrow \alpha\mathbf{T}^{-1}\mathbf{B}$, $\mathbf{B} \leftarrow \alpha\mathbf{B}\mathbf{T}^{-1}$	TR, TB	SM_I
	$\mathbf{B} \leftarrow \alpha\mathbf{T}^{-T}\mathbf{B}$, $\mathbf{B} \leftarrow \alpha\mathbf{B}\mathbf{T}^{-T}$		

Table C.7: Matrix Operations – $O(n^3)$ floating point operations

Matrix copy	$\mathbf{B} \leftarrow \mathbf{A}$	GE,GB,SY,SB,SP,TR,TB,TP	_COPY_I
	$\mathbf{B} \leftarrow \mathbf{A}^T$	GE, GB	_COPY_I
Matrix transpose	$\mathbf{A} \leftarrow \mathbf{A}^T$	GE	_TRANS_I
Permute matrix	$\mathbf{A} \leftarrow \mathbf{PA}, \mathbf{A} \leftarrow \mathbf{AP}$	GE	_PERMUTE_I

Table C.8: Data Movement with Interval Matrices

Set Operations Involving Interval Vectors

Table C.9 lists set operations for interval vectors.

Enclosed	$\mathbf{x}$ is enclosed in $\mathbf{y}$ if $\mathbf{x} \subseteq \mathbf{y}$	ENCV_I
Interior	$\mathbf{x}$ is enclosed in the interior of $\mathbf{y}$	INTERIORV_I
Disjoint	$\mathbf{x}$ and $\mathbf{y}$ are disjoint if $\mathbf{x} \cap \mathbf{y} = \emptyset$	DISJV_I
Intersection	$\mathbf{y} \leftarrow \mathbf{x} \cap \mathbf{y}, \mathbf{z} \leftarrow \mathbf{x} \cap \mathbf{y}$	INTERV_I, WINTERV_I
Hull	the convex hull of $\mathbf{x}$ and $\mathbf{y}$	HULLV_I, WHULLV_I

Table C.9: Set Operations for Interval Vectors

Set Operations Involving Interval Matrices

Table C.10 lists set operations for interval matrices.

Enclosed	$\mathbf{A}$ is enclosed in $\mathbf{B}$ if $\mathbf{A} \subseteq \mathbf{B}$	GE,GB,SY,SB, SP,TR,TB,TP	_ENCM_I
Interior	$\mathbf{A}$ is enclosed in the interior of $\mathbf{B}$	GE,GB,SY,SB, SP,TR,TB,TP	_INTERIORM_I
Disjoint	$\mathbf{A}$ and $\mathbf{B}$ are disjoint if $\mathbf{A} \cap \mathbf{B} = \emptyset$	GE,GB,SY,SB, SP,TR,TB,TP	_DISJM_I
Intersection	$\mathbf{B} \leftarrow \mathbf{A} \cap \mathbf{B}$, $\mathbf{C} \leftarrow \mathbf{A} \cap \mathbf{B}$	GE,GB,SY,SB, SP,TR,TB,TP	_INTERM_I, _WINTERM_I
Hull	the convex hull of $\mathbf{A}$ and $\mathbf{B}$	GE,GB,SY,SB, SP,TR,TB,TP	_HULLM_I, _WHULLM_I

Table C.10: Set Operations for Interval Matrices

Utility Functions Involving Interval Vectors

Table C.11 lists some utility operations for interval vectors.

Utility Functions Involving Interval Matrices

Table C.12 lists some utility operations for interval matrices.

Empty element	k if $\mathbf{x}_k = \emptyset$; or -1	EMPTYELEV_I
Left endpoint	$v \leftarrow \underline{x}$	INFV_I
Right endpoint	$v \leftarrow \overline{x}$	SUPV_I
Midpoint	$v \leftarrow (\underline{x} + \overline{x})/2$	MIDV_I
Width	$v \leftarrow \overline{x} - \underline{x}$	WIDTHV_I
Construct	$\mathbf{x} \leftarrow u, v$	CONSTRUCTV_I

Table C.11: Utility Operations for Interval Vectors

Empty element	if $\mathbf{A}$ has an empty interval element	GE,GB,SY,SB, SP,TR,TB,TP	_EMPTYELEM_I
Left endpoint	$C \leftarrow \underline{A}$	GE,GB,SY,SB, SP,TR,TB,TP	_INFM_I
Right endpoint	$C \leftarrow \overline{A}$	GE,GB,SY,SB, SP,TR,TB,TP	_SUPM_I
Midpoint	$C \leftarrow (\underline{A} + \overline{A})/2$	GE,GB,SY,SB, SP,TR,TB,TP	_MIDM_I
Width	$C \leftarrow \overline{A} - \underline{A}$	GE,GB,SY,SB,	_WIDTHM_I
Construct	$\mathbf{A} \leftarrow B, C$	GE,GB,SY,SB, SP,TR,TB,TP	_CONSTRUCTM_I

Table C.12: Utility Operations

C.4.3 Interface Issues

Naming Conventions

The naming conventions are the same as described in section 2.3.1 except that the suffix _I (or _i) is added to indicate an interval BLAS routine.

Interface Issues for Fortran 95

Design of the Fortran 95 Interfaces

The Fortran 95 binding is defined in a module. The specific interfaces in this module should declare the default interval data type as `TYPE(INTERVAL)`.

> *Advice to implementors:* In the Fortran 95 interfaces, it is assumed that `INTERVAL` is a derived type. However, in compilers that support an intrinsic interval type, it is recommended that an alternate module that contains appropriately modified declarations also be supplied. For example, `TYPE(INTERVAL), INTENT(IN) :: ALPHA` could become `INTERVAL, INTENT(IN) :: ALPHA` in a recommended alternate module.

The Fortran 95 interval BLAS routines are consistent with regard to generic interfaces, precision, rank, assumed-shape arrays, derived types, operator arguments and `CMACH` values, and error handling as described in section 2.4 of this document. However, in the interval BLAS, α and β are intervals; and their default values are `alpha = [1,1], beta = [0,0]`.
Error handling is as defined in section 2.4.6.

Format of the Fortran 95 bindings

Each interface is summarized in the form of a SUBROUTINE statement (or in few cases a FUNCTION statement), in which all of the potential arguments appear. Arguments which need not be supplied are grouped after the mandatory arguments and enclosed in square brackets, for example:

```
SUBROUTINE axpby_i( x, y [, alpha] [, beta] )
  TYPE(INTERVAL) (<wp>), INTENT (IN) :: x (:)
  TYPE(INTERVAL) (<wp>), INTENT (INOUT) :: y (:)
  TYPE(INTERVAL) (<wp>), INTENT (IN), OPTIONAL :: alpha, beta
```

Variables in interval BLAS routines should be specified as INTEGER, REAL, TYPE(INTERVAL) or types defined in MODULE blas_operator_arguments. The precision of a real or interval variable is denoted by <wp> where

```
<wp> ::= KIND(1.0) | KIND(1.0D0)
```

Interface Issues for Fortran 77

The interval BLAS Fortran 77 binding is consistent with ANSI standard Fortran 77 except the following:

- Subroutine names are not limited to six significant characters.
- Subroutine names contain one or more underscores.
- Subroutines may use the INCLUDE statement for include files.

In interval BLAS Fortran 77 binding, α and β are intervals and their default values are: ALPHA = [1.0, 1.0] and BETA = [0.0, 0.0]. Without assuming an intrinsic interval data type, an interval, say α, will be declared as REAL or DOUBLE PRECISION ALPHA(2); an interval vector will be stored as REAL or DOUBLE PRECISION X(2,*); and a general interval matrix will be defined as REAL or DOUBLE PRECISION A(2,LDA, *).

> *Advice to implementors:* On Fortran 77 compilers that have an intrinsic interval data type, an interval vector will be stored as INTERVAL X(*), and a general interval matrix will be defined as INTERVAL A(LDA, *).

The Fortran 77 interval BLAS routines are consistent with regard to indexing of vector and matrix operands, operator arguments and CMACH values, array arguments, matrix storage schemes, and error handling as described in section 2.5 of this document but with interval variables.
Error handling is as defined in section 2.5.6.

Format of the Fortran 77 bindings

Each interface is summarized in the form of a SUBROUTINE statement (or a FUNCTION statement). For example:

```
SUBROUTINE BLAS_xAXPBY_I( N, ALPHA, X, INCX, BETA, Y, INCY )
  INTEGER    INCX, INCY, N
  <type>     ALPHA(2), BETA(2)
  <type>     X(2,*), Y(2,*)
```

Floating point variables are denoted by the keyword <type> which may be REAL or DOUBLE PRECISION, and should agree with the x letter in the naming convention of the routine.

Interface Issues for C

The interface is expressed in terms of ANSI/ISO C. *All interval arguments are accepted as* `float *` or `double *`. An interval element consists of two consecutive memory locations of the underlying data type (i.e., `float` or `double`), where the first location contains the lower bound of the interval, and the second contains the upper bound of the interval.
The C interval BLAS routines are consistent with regard to indexing of vector and matrix operands, operator arguments and `CMACH` values, array arguments, matrix storage schemes, and error handling that described in section 2.6 of this document but with interval variables. The default value for intervals `alpha` and `beta` are `alpha` $=[1.0,1.0]$ and `beta` $=[0.0,0.0]$.
Error handling is as defined in section 2.6.9.

Format of the C bindings

Each interval BLAS routine is summarized in the form of an ANSI/ISO C prototype. For example:

```
void BLAS_xaxpby_i( int n, <interval> alpha, const <interval_array> x,
                    int incx, <interval> beta, <interval_array> y,
                    int incy)
```

In the C binding, we use the keywords <interval> and <interval_array> to indicate if an argument is a single interval or an interval vector/matrix. In fact, <interval> and <interval_array> can be `float *` or `double *`. A real number, not an interval, will be indicated by the keyword `SCALAR`. A vector/matrix of real numbers, not intervals, will be specified by `RARRAY`. The precisions of `SCALAR`, `RARRAY` can be `float` or `double`. They will agree with the `x` letter in the naming convention of the routine. However, in some routines, not all floating point variables will be the same type. If this is the case, then a variable may be denoted by the keywords `SCALAR_IN` or `SCALAR_INOUT`. `SCALAR_IN` can be `float` or `double`; and `SCALAR_INOUT` and `RARRAY` can be `float *` or `double *`.

C.4.4 Numerical Accuracy and Environmental Enquiry

The semantics of interval arithmetic require us to have another environmental enquiry function to supplement the routine FPINFO described in sections 1.6 and 2.7. Here we will specify the additional routine FPINFO_I to determine how tightly the containment property of interval arithmetic is maintained.
To establish notation, let $\mathbf{a}=[\underline{a},\overline{a}]$ and $\mathbf{b}=[\underline{b},\overline{b}]$ be machine intervals, let `op` be one of the operations $+$, $-$, $\ominus$, $\times$ and $\div$, let $\mathbf{c}=[\underline{c},\overline{c}]=\mathbf{a}\ \texttt{op}\ \mathbf{b}$ be the exact mathematical interval result of $\mathbf{a}\ \texttt{op}\ \mathbf{b}$, and let $\mathbf{c}^*=[\underline{c}^*,\overline{c}^*]=fl(\mathbf{a}\ \texttt{op}\ \mathbf{b})$ be the machine interval computed containing $\mathbf{c}$. Let $\epsilon_I>0$ be defined as the smallest number such that for all $\mathbf{a}$, $\mathbf{b}$ and `op` where overflow and underflow do not occur in computing $\mathbf{c}^*$, then

$$
\begin{aligned}
\underline{c} &\geq \min\{\underline{c}^*(1+\epsilon_I),\underline{c}^*(1-\epsilon_I)\} \\
\overline{c} &\leq \max\{\overline{x}^*(1+\epsilon_I),\overline{c}^*(1-\epsilon_I)\}
\end{aligned}
$$

In other words, ϵ_I measures how much the exact mathematical interval bounds are rounded out to get the machine interval result. When the machine interval is tight, i.e. as narrow as possible, then $\epsilon_I=BASE^{1-T}$, where $BASE$ and T are values returned by FPINFO. But ϵ_I could be larger depending on the implementation, leading us to the following environmental enquiry:

Value of CMACH	Name of value returned by FPINFO_I	Description
blas_base	BASE	base of the machine
blas_t_i	T_I	effective number of base BASE digits, such that $\epsilon_I = BASE^{1-T_I}$
blas_rnd_i	RND_I	when interval arithmetic is implemented with correct IEEE-style directed rounding
blas_eps_i	EPS_I	ϵ_I as defined above.

C.4.5 Language Bindings

Each specification of a routine will correspond to an operation outlined in the functionality tables. Operations are organized analogous to the order in which they are presented in the functionality tables. The format of the language bindings is as described in section 2.8.

Overview

- Reduction Operations (section C.4.5)
  - DOT_I (Dot product)
  - NORM_I (Interval vector norms)
  - SUM_I (Sum)
  - AMIN_VAL_I (Min absolute value & location)
  - AMAX_VAL_I (Max absolute value & location)
  - SUMSQ_I (Sum of squares)
- Interval Vector Operations (section C.4.5)
  - RSCALE_I (Reciprocal Scale)
  - AXPBY_I (Scaled vector accumulation)
  - WAXPBY_I (Scaled vector addition)
  - CANCEL_I (Scaled cancellation)
  - WCANCEL_I (Scaled cancellation)
- Data Movement with Interval Vectors (section C.4.5)
  - COPY_I (Interval vector copy)
  - SWAP_I (Interval vector swap)
  - PERMUTE_I (Permute interval vector)
- Interval Matrix-Vector Operations (section C.4.5)
  - {GE,GB}MV_I (Interval matrix vector product)
  - {SY,SB,SP}MV_I (Interval symmetric matrix vector product)
  - {TR,TB,TP}MV_I (Interval triangular matrix vector product)
  - {TR,TB,TP}SV_I (Interval triangular solve)
  - GER_I (Rank one update)

  - {SY,SP}R_I (Symmetric rank one update)

- Interval Matrix Operations (section C.4.5)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_NORM_I (Interval matrix norms)
  - {GE,GB}_DIAG_SCALE_I (Diagonal scaling)
  - {GE,GB}_LRSCALE_I (Two-sided diagonal scaling)
  - {SY,SB,SP}_LRSCALE_I (Two-sided diagonal scaling of a symmetric interval matrix)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_ACC_I (Matrix accumulation and scale)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_ADD_I (Matrix add and scale)
- Interval Matrix-Matrix Operations (section C.4.5)
  - GEMM_I (General interval Matrix Matrix product)
  - SYMM_I (Symmetric interval matrix matrix product)
  - TRMM_I (Triangular interval matrix matrix multiply)
  - TRSM_I (Interval triangular solve)
- Data Movement with Interval Matrices (section C.4.5)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_COPY_I (Matrix copy)
  - GE_TRANS_I (Matrix transposition)
  - GE_PERMUTE_I (Permute an interval matrix)
- Set Operations Involving Interval Vectors (section C.4.5)
  - ENCV_I (Checks if an interval vector is enclosed in another interval vector)
  - INTERIORV_I (Checks if an interval vector is enclosed in the interior of another interval vector)
  - DISJV_I (Checks if two interval vectors are disjoint)
  - INTERV_I (Intersection of an interval vector with another)
  - WINTERV_I (Intersection of two interval vectors)
  - HULLV_I (Convex hull of an interval vector with another)
  - WHULLV_I (Convex hull of two interval vectors)
- Set Operations Involving Interval Matrices (section C.4.5)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_ENCM_I (Checks if an interval matrix is enclosed in another interval matrix)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_INTERIORM_I (Checks if an interval matrix is enclosed in the interior of another interval matrix)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_DISJM_I (Checks if two interval matrices are disjoint)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_INTERM_I (Elementwise intersection of an interval matrix with another)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_WINTERM_I (Elementwise intersection of two interval matrices)

  - {GE,GB,SY,SB,SP,TR,TB,TP}_HULLM_I (Convex hull of an interval matrix with another)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_WHULLV_I (Convex hull of two interval matrices)

- Utility Functions Involving Interval Vectors (section C.4.5)
  - EMPTYELEV_I (Empty entry and location)
  - INFV_I (The left endpoint of an interval vector)
  - SUPV_I (The right endpoint of an interval vector)
  - MIDV_I (The approximate midpoint of an interval vector)
  - WIDTHV_I (The elementwise width of an interval vector)
  - CONSTRUCTV_I (Constructs an interval vector from two floating point vectors)

- Utility Functions Involving Interval Matrices (section C.4.5)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_EMPTYELEM_I (Empty entry and location)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_INFM_I (The left endpoint of an interval matrix)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_SUPM_I (The right endpoint of an interval matrix)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_MIDM_I (The approximate midpoint of an interval matrix)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_WIDTHM_I (Elementwise width of an interval matrix)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_CONSTRUCTM_I (Constructs an interval matrix from two given floating point matrices)

- Environmental Enquiry (section C.4.5)
  - FPINFO_I (Environmental enquiry)

Reduction Operations

DOT_I (Dot Product) $\mathbf{r} \leftarrow \beta \mathbf{r} + \alpha \mathbf{x}^T \mathbf{y}$

The routine `DOT_I` adds the scaled dot product of two interval vectors $\mathbf{x}$ and $\mathbf{y}$ into a scaled interval $\mathbf{r}$. The routine returns immediately if `n` is less than zero, or, if `beta` is equal to [1,1] and either `alpha` is equal to [0,0] or `n` is equal to zero. If `alpha` is equal to [0,0] then $\mathbf{x}$ and $\mathbf{y}$ are not read. Similarly, if `beta` is equal to [0,0], $\mathbf{r}$ is not referenced. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.
179, one.

- Fortran 95 binding:

```
SUBROUTINE dot_i( x, y, r [, alpha] [,beta] )
  TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:), y(:)
  TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
  TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: r
where
  x and y have shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xDOT_I( N, ALPHA, X, INCX, BETA, Y, INCY, R )
INTEGER          INCX, INCY, N
<type>           ALPHA( 2 ), BETA( 2 ), R( 2 )
<type>           X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xdot_i( int n, const <interval> alpha, const <interval_array> x,
                  int incx, const <interval> beta, const <interval_array> y,
                  int incy, <interval> r );
```

Advice to users: The scaling parameters `alpha` and `beta` are intervals. If any one of them is a real number in applications, the user needs to convert it into its interval representation first, and then use the routine.

NORM_I (Interval vector norms) $r \leftarrow ||\mathbf{x}||_1,\ ||\mathbf{x}||_2, ||\mathbf{x}||_\infty$

The routine NORM_I computes the $||\cdot||_1$, $||\cdot||_2$, or $||\cdot||_\infty$ of a vector x depending on the value passed as the norm operator argument.
If `n` is less than or equal to zero, this routine returns immediately with the output scalar `r` set to zero. The resulting scalar `r` is always real and its value is as defined in section 2.1.1, provided that $|\mathbf{x}_i| = \max\{|\underline{x}_i|, |\overline{x}_i|\}$.
As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
REAL (<wp>) FUNCTION norm_i ( x [, norm] )
  TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
  TYPE(blas_norm_type),  INTENT(IN), OPTIONAL :: norm
where
  x has shape (n)
```

- Fortran 77 binding:

```
<type>   FUNCTION BLAS_xNORM_I( NORM, N, X, INCX )
INTEGER            INCX, N, NORM
<type>             X( 2, * )
```

- C binding:

```
void BLAS_xnorm_i( enum blas_norm_type norm, int n, const <interval_array> x,
                   int incx, SCALAR_INOUT r );
```

Advice to implementors: In finite precision floating point arithmetic, an upper bound, preferably the least machine representable upper bound, for the mathematical value should be returned for the norms.

SUM_I (Sum the entries of an interval vector) $\mathbf{r} \leftarrow \sum_{i=0}^{n-1} \mathbf{x}_i$

The routine SUM_I returns the sum of the entries of an interval vector **x**. If n is less than or equal to zero, this routine returns immediately with the output interval **r** set to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE sum_i( x, r )
  TYPE(INTERVAL)(<wp>), INTENT(IN) :: x(:)
  TYPE(INTERVAL)(<wp>), INTENT(OUT) :: r
where
  x has shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xSUM_I( N, INCX, X, R )
INTEGER          INCX, N
<type>           X( 2, * )
<type>           R( 2 )
```

- C binding:

```
void BLAS_xsum_i( int n, int incx, const <interval_array> x, <interval> r );
```

AMIN_VAL_I ($\min_{0\leq i<n}\{|\underline{x}_i|, |\overline{x}_i|\}$ & location) $k, r \leftarrow \min\{|\underline{x}_k|, |\overline{x}_k|\} = r = \min_{0\leq i<n}\{|\underline{x}_i|, |\overline{x}_i|\}$

The routine AMIN_VAL_I finds the index of the component of an interval vector such that the absolute value of the lower or upper bounds of the component is the smallest among the absolute values of the lower and upper bounds of all components of the interval vector. When the value of the n argument is less than or equal to zero, the routine should initialize the output k to negative one or zero, and r to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE amin_val_i( x, k, r )
  TYPE(INTERVAL)(<wp>), INTENT(IN) :: x(:)
  INTEGER, INTENT(OUT) :: k
  REAL (<wp>), INTENT(OUT) :: r
where
  x has shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xAMIN_VAL_I( N, X, INCX, K, R )
INTEGER           INCX, K, N
<type>            X( 2, * )
<type>            R
```

- C binding:

```
void BLAS_xamin_val_i( int n, const <interval_array> x, int incx, int k,
                       SCALAR_INOUT r );
```

AMAX_VAL_I (Max absolute value & location) $k, r \leftarrow \max\{|\underline{x}_k|, |\overline{x}_k|\} = r = \max\limits_{0\le i<n}\{|\underline{x}_i|, |\overline{x}_i|\}$

The routine `AMAX_VAL_I` finds the index of the component of an interval vector such that the absolute value of the lower or upper bounds of the component has the largest value among the absolute values of the lower and upper bounds of all components of the interval vector. When the value of the `n` argument is less than or equal to zero, the routine should initialize the output `k` to negative one or zero, and `r` to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE amax_val_i( x, k, r )
  TYPE(INTERVAL)(<wp>), INTENT(IN) :: x(:)
  INTEGER, INTENT(OUT) :: k
  REAL (<wp>), INTENT(OUT) :: r
where
  x has shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_AMAX_VAL_I( N, X, INCX, K, R )
INTEGER           INCX, K, N
<type>            X( 2, * )
<type>            R
```

- C binding:

```
void BLAS_xamax_val_i( int n, const <interval_array> x, int incx, int k,
                       SCALAR_INOUT  r );
```

SUMSQ_I (Sum of squares) $(scl, ssq) \leftarrow \sum \mathbf{x}_i^2$

The routine `SUMSQ_I` returns the intervals *scl* and *ssq* such that

$$scl^2 * ssq = scale^2 * sumsq + \sum_{i=0}^{n-1} \mathbf{x}_i^2.$$

The value of *sumsq* is assumed to be at least unity and the value of *ssq* will then satisfy $1.0 \leq ssq \leq (sumsq + n)$. It is assumed that *scale* is to be non-negative, and *scl* returns the value

$$scl = \max_{0 \leq i < n} (scale, |\mathbf{x}_i|).$$

scale and *sumsq* must be supplied on entry in `scl` and `ssq` respectively. `scl` and `ssq` are overwritten by *scl* and *ssq* respectively. If `n` is less than or equal to zero, this routine returns immediately with `scl` and `ssq` unchanged. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE sumsq_i( x, ssq, scl )
  TYPE(INTERVAL)(<wp>), INTENT(IN) :: x(:)
  TYPE(INTERVAL)(<wp>), INTENT(INOUT) :: ssq, scl
where
  x has shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xSUMSQ_I( N, X, INCX, SSQ, SCL )
INTEGER          INCX, N
<type>           X( 2, * )
<type>           SCL( 2 ), SSQ( 2 )
```

- C binding:

```
void BLAS_xsumsq_i( int n, const <interval_array> x, int incx, <interval> ssq,
                    <interval> scl );
```

Interval Vector Operations

RSCALE_I (Reciprocal Scale of an interval vector) $\mathbf{x} \leftarrow \mathbf{x}/\alpha$

The routine `RSCALE_I` updates the entries of an interval vector $\mathbf{x}$ by the scale interval $1/\alpha$ provided that $0 \notin \alpha$. If n is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE rscale_i( alpha, x )
  TYPE(INTERVAL)(<wp>), INTENT(INOUT) :: x(:)
  TYPE(INTERVAL)(<wp>), INTENT(IN) :: alpha
where
  x has shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xRSCALE_I( N, ALPHA, X, INCX )
INTEGER          INCX, N
<type>           ALPHA( 2 )
<type>           X( 2, * )
```

- C binding:

```
void BLAS_xrscale_i( int n, <interval> alpha, <interval_array> x, int incx );
```

AXPBY_I (Scaled vector accumulation) $\mathbf{y} \leftarrow \alpha\mathbf{x} + \beta\mathbf{y}$

The routine AXPBY_I scales the interval vector $\mathbf{x}$ by the interval α and the interval vector $\mathbf{y}$ by β, adds these two vectors to one another and stores the result in the vector $\mathbf{y}$. If n is less than or equal to zero, or if α is equal to [0,0] and β equal to [1,1], this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
 SUBROUTINE axpby_i( x, y [, alpha] [, beta] )
  <type>(<wp>), INTENT (IN) :: x (:)
  <type>(<wp>), INTENT (INOUT) :: y (:)
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
where
  x and y have shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xAXPBY_I( N, ALPHA, X, INCX, BETA, Y, INCY )
INTEGER            INCX, INCY, N
<type>             ALPHA( 2 ), BETA( 2 )
<type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xaxpby_i( int n, <interval> alpha, <interval_array> x, int incx,
                    <interval> beta, <interval_array> y, int incy );
```

WAXPBY_I (Scaled vector addition) $\mathbf{w} \leftarrow \alpha\mathbf{x} + \beta\mathbf{y}$

The routine WAXPBY_I scales the interval vector $\mathbf{x}$ by the interval α and the interval vector $\mathbf{y}$ by β, adds these two vectors to one another and stores the result in the vector $\mathbf{w}$. If n is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy or incw less than zero is permitted. However, if either incx or incy or incw is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
     SUBROUTINE waxpby_i( x, y, w [, alpha] [, beta] )
      <type>(<wp>), INTENT (IN) :: x(:), y(:)
      <type>(<wp>), INTENT (OUT) :: w(:)
      <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
    where
      x, y and w have shape (n)
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xWAXPBY_I( N, ALPHA, X, INCX, BETA, Y, INCY, W,
   $                           INCW )
    INTEGER            INCW, INCX, INCY, N
    <type>             ALPHA( 2 ), BETA( 2 )
    <type>             W( 2, * ), X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_wxaxpby_i( int n, <interval> alpha, const <interval_array> x,
                     int incx, <interval> beta, const <interval_array> y,
                     int incy, <interval_array> w, int incw );
```

CANCEL_I (Scaled cancellation) $\mathbf{y} \leftarrow \alpha\mathbf{x} \ominus \beta\mathbf{y}$

The operation cancel, $\ominus$, between two intervals $\mathbf{a}$ and $\mathbf{b}$ is defined as $\mathbf{a} \ominus \mathbf{b} = [\underline{a} - \underline{b}, \overline{a} - \overline{b}]$ if $(\underline{a} - \underline{b}) \leq (\overline{a} - \overline{b})$; Otherwise, $\emptyset$. The routine `CANCEL_I` scales the interval vector $\mathbf{x}$ by the interval α and the interval vector $\mathbf{y}$ by β, updates $\mathbf{y}_i$ with $\alpha\mathbf{x}_i \ominus \beta\mathbf{y}_i$, $\forall 0 \leq i < n$. If `n` is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
     SUBROUTINE cancel_i( x, y [, alpha] [, beta] )
      <type>(<wp>), INTENT (IN) :: x (:)
      <type>(<wp>), INTENT (INOUT) :: y (:)
      <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
     where
       x and y have shape (n)
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xCANCEL_I( N, ALPHA, X, INCX, BETA, Y, INCY )
    INTEGER            INCX, INCY, N
    <type>             ALPHA( 2 ), BETA( 2 )
    <type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xcancel_i( int n, <interval> alpha, <interval_array> x, int incx,
                     <interval> beta, <interval_array> y, int incy );
```

WCANCEL_I (Scaled cancellation) $\mathbf{w} \leftarrow \alpha\mathbf{x} \ominus \beta\mathbf{y}$

The operation cancel, $\ominus$, between two intervals $\mathbf{a}$ and $\mathbf{b}$ is defined as $\mathbf{a} \ominus \mathbf{b} = [\underline{a} - \underline{b}, \overline{a} - \overline{b}]$ if $(\underline{a} - \underline{b}) \leq (\overline{a} - \overline{b})$; Otherwise, $\emptyset$. The routine WCANCEL_I scales the interval vector $\mathbf{x}$ by the interval α and the interval vector $\mathbf{y}$ by β, stores $\alpha\mathbf{x}_i \ominus \beta\mathbf{y}_i$ in $\mathbf{w}_i$ for $0 \leq i < n$. If n is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy or incw less than zero is permitted. However, if either incx or incy or incw is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE wcancel_i( x, y, w [, alpha] [, beta] )
  <type>(<wp>), INTENT (IN) :: x(:), y(:)
  <type>(<wp>), INTENT (OUT) :: w(:)
  <type>(<wp>), INTENT (IN), OPTIONAL :: alpha, beta
where
  x, y, and w have shape (n)
```

- Fortran 77 binding:

```
 SUBROUTINE BLAS_xWCANCEL_I( N, ALPHA, X, INCX, BETA, Y, INCY, W,
$                            INCW )
 INTEGER            INCW, INCX, INCY, N
 <type>             ALPHA( 2 ), BETA( 2 )
 <type>             W( 2, * ), X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xwcancel_i( int n, <interval> alpha, <interval_array> x, int incx,
                      <interval> beta, <interval_array> y, int incy,
                      <interval_array> w, int incw );
```

Data Movement with Interval Vectors

COPY_I (Interval vector copy) $\mathbf{y} \leftarrow \mathbf{x}$

The routine COPY_I copies the interval vector $\mathbf{x}$ into the interval vector $\mathbf{y}$. If n is less than or equal to zero, the routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
      SUBROUTINE copy_i( x, y )
        TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
        TYPE(INTERVAL) (<wp>), INTENT(OUT) :: y(:)
      where
        x and y have shape (n)
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xCOPY_I( N, X, INCX, Y, INCY )
      INTEGER            INCX, INCY, N
      <type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xcopy_i( int n, const <interval_array> x, int incx,
                   <interval_array> y, int incy );
```

SWAP_I (Interval vector swap) $\mathbf{y} \leftrightarrow \mathbf{x}$

The routine SWAP_I interchanges the interval vectors $\mathbf{x}$ and $\mathbf{y}$. If n is less than or equal to zero, the routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
      SUBROUTINE swap_i( x, y )
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: x(:), y(:)
      where
        x and y have shape (n)
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xSWAP_I( N, X, INCX, Y, INCY )
      INTEGER            INCX, INCY, N
      <type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xswap_i( int n, <interval_array> x, int incx, <interval_array> y,
                   int incy );
```

PERMUTE_I (Permute interval vector) $\mathbf{x} \leftarrow P\mathbf{x}$

The routine PERMUTE_I permutes the entries of an interval vector $\mathbf{x}$ according to the permutation vector P. If n is less than or equal to zero, the routine returns immediately. As described in section 2.5.3, the value incx or incp less than zero is permitted. However, if either incx or incp is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
      SUBROUTINE permute_i(x, p )
        INTEGER, INTENT(IN) :: p(:)
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: x(:)
      where
        x and p have shape (n)
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xPERMUTE_I( N, P, INCP, X, INCX )
      INTEGER            INCP, INCX, N
      INTEGER            P( * )
      <type>             X( 2, * )
```

- C binding:

```
void BLAS_xpermute_i( int n, const int *p, int incp, <interval_array> x,
                      int incx );
```

Interval Matrix-Vector Operations

{GE,GB}MV_I (Interval matrix-vector multiplication) $\mathbf{y} \leftarrow \alpha \mathbf{A}\mathbf{x} + \beta \mathbf{y}, \mathbf{y} \leftarrow \alpha \mathbf{A}^T \mathbf{x} + \beta \mathbf{y}$

The routines multiply the interval vector **x** by a general (or general band) interval matrix **A** or its transpose, scales the resulting interval vector and adds it to the scaled interval vector operand **y**. If `m` or `n` is less than or equal to zero or if `beta` is equal to [1,1] and `alpha` is equal to [0,0], the routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. For the routine GEMV_I, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine GBMV_I, if kl or ku is less than zero, or if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
      SUBROUTINE gemv_i( a, x, y [, transa] [, alpha] [, beta] )
General Band:
      SUBROUTINE gbmv_i( a, m, kl, x, y [, transa] [, alpha] [, beta] )
all:
        TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), x(:)
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: y(:)
        INTEGER INTENT(IN) :: m, kl
        TYPE(blas_trans_type), INTENT(IN), OPTIONAL :: transa
        TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      where
        a has shape (m,n) for general matrix
                    (l,n) for general banded matrix ( l > kl)
        x and y have shape n if transa = blas_no_trans (the default}
                           m if transa /= blas_no_trans
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGEMV_I( TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA,
     $                         Y, INCY )
General Band:
      SUBROUTINE BLAS_xGBMV_I( TRANS, M, N, KL, KU, ALPHA, A, LDA, X, INCX,
     $                         BETA, Y, INCY )
all:
      INTEGER            INCX, INCY, KL, KU, LDA, M, N, TRANS
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             A( 2, LDA, * ), X( 2, * ), Y( 2, * )
```

- C binding:

```
General:
void BLAS_xgemv_i( enum blas_order_type order, enum blas_trans_type trans,
                   int m, int n, <interval> alpha, const <interval_array> a,
                   int lda, const <interval_array> x, int incx, <interval> beta,
                   <interval_array> y, int incy );
General Band:
void BLAS_xgbmv_i( enum blas_order_type order, enum blas_trans_type trans,
                   int m, int n, int kl, int ku, <interval> alpha,
                   const <interval_array> a, int lda, const <interval_array> x,
                   int incx, <interval> beta, <interval_array> y, int incy );
```

{SY,SB,SP}MV_I (Interval symmetric matrix vector product) $\mathbf{y} \leftarrow \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$ with $\mathbf{A} = \mathbf{A}^T$

The routines multiply an interval vector $\mathbf{x}$ by a symmetric interval matrix $\mathbf{A}$, scales the resulting interval vector and adds it to the scaled interval vector operand $\mathbf{y}$. If `n` is less than or equal to zero or if `beta` is equal to one and `alpha` is equal to zero, the routine returns immediately. The operator argument `uplo` specifies if the matrix operand is an upper or lower triangular part of the symmetric matrix. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. For the routine SYMV_I, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine SBMV_I, if lda is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
Symmetric:
      SUBROUTINE symv_i( a, x, y [, uplo] [, alpha] [, beta] )
Symmetric Band:
      SUBROUTINE sbmv_i( a, x, y [, uplo] [, alpha] [, beta] )
Symmetric Packed:
      SUBROUTINE spmv_i( ap, x, y [, uplo] [, alpha] [, beta] )
```

```
all:
        TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), ap(:), x(:)
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: y(:)
        TYPE(blas_uplo_type),  INTENT(IN), OPTIONAL :: uplo
        TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      where
        x and y have shape (n)
       SY  a has shape (n,n)
       SB  a has shape (k+1,n), where k = band width
       SP  ap has shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
Symmetric:
      SUBROUTINE BLAS_xSYMV_I( UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y,
     $                      INCY )
Symmetric Band:
      SUBROUTINE BLAS_xSBMV_I( UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA,
     $                      Y, INCY )
Symmetric Packed:
      SUBROUTINE BLAS_xSPMV_I( UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY )
all:
      INTEGER            INCX, INCY, K, LDA, N, UPLO
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             A( 2, LDA, * ) or AP( 2, * ), X( 2, * ),
     $                   Y( 2, * )
```

- C binding:

```
Symmetric:
void BLAS_xsymv_i( enum blas_order_type order, enum blas_uplo_type uplo, int n,
                   <interval> alpha, const <interval_array> a, int lda,
                   const <interval_array> x, int incx, <interval> beta,
                   <interval_array> y, int incy );
Symmetric Band:
void BLAS_xsbmv_i( enum blas_order_type order, enum blas_uplo_type uplo, int n,
                   int k, <interval> alpha, const <interval_array> a, int lda,
                   const <interval_array> x, int incx, <interval> beta,
                   <interval_array> y, int incy );
Symmetric Packed:
void BLAS_xspmv_i( enum blas_order_type order, enum blas_uplo_type uplo, int n,
                   <interval> alpha, const <interval_array> ap,
                   const <interval_array> x, int incx, <interval> beta,
                   <interval_array> y, int incy );
```

{TR,TB,TP}MV_I (Interval triangular matrix vector product) $\mathbf{x} \leftarrow \alpha \mathbf{T}\mathbf{x}, \mathbf{x} \leftarrow \alpha \mathbf{T}^T\mathbf{x}$

The routines multiply an interval vector **x** by a general triangular interval matrix **T** or its transpose, and copies the resulting vector in the vector operand **x**. If n is less than or equal to zero, the routine returns immediately. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler. For the routine TRMV_I, if ldt is less than one or ldt is less than n, an error flag is set and passed to the error handler. For the routine TBMV_I, if ldt is less than k plus one, an error flag is set and passed to the error handler.
The operator argument uplo specifies whether the matrix operand is upper or lower triangular. The operator argument diag specifies whether or not the matrix operand has unit diagonal entries.

- Fortran 95 binding:

```
Triangular:
      SUBROUTINE trmv_i( t, x [, uplo] [, transt] [, diag] [, alpha] )
Triangular Band:
      SUBROUTINE tbmv_i( t, x [, uplo] [, transt] [, diag] [, alpha] )
Triangular Packed:
      SUBROUTINE tpmv_i( tp, x [, uplo] [, transt] [, diag] [, alpha] )
all:
        TYPE(INTERVAL) (<wp>), INTENT(IN) :: t(:,:), tp(:)
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: x(:)
        TYPE(blas_uplo_type),  INTENT(IN), OPTIONAL :: uplo
        TYPE(blas_trans_type), INTENT(IN), OPTIONAL :: transt
        TYPE(blas_diag_type),  INTENT(IN), OPTIONAL :: diag
        TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha
      where
        x has shape (n)
       TR  t has shape (n,n)
       TB  t has shape (k+1,n) where k = band width
       TP  tp has shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
Triangular:
      SUBROUTINE BLAS_xTRMV_I( UPLO, TRANS, DIAG, N, ALPHA, T, LDT, X,
     $                         INCX )
Triangular Band:
      SUBROUTINE BLAS_xTBMV_I( UPLO, TRANS, DIAG, N, K, ALPHA, T, LDT, X,
     $                         INCX )
Triangular Packed:
      SUBROUTINE BLAS_xTPMV_I( UPLO, TRANS, DIAG, N, ALPHA, TP, X, INCX )
all:
      INTEGER            DIAG, INCX, K, LDA, N, TRANS, UPLO
      <type>             ALPHA( 2 )
      <type>             T( 2, LDA, * ) or TP( 2, * ), X( 2, * )
```

- C binding:

```
Triangular:
void BLAS_xtrmv_i( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, enum blas_diag_type diag, int n,
                   <interval> alpha, const <interval_array> t, int ldt,
                   <interval_array> x, int incx );
Triangular Band:
void BLAS_xtbmv_i( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, enum blas_diag_type diag, int n,
                   <interval> alpha, const <interval_array> t, int ldt,
                   <interval_array> x, int incx );
Triangular Packed:
void BLAS_xtpmv_i( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, enum blas_diag_type diag, int n,
                   <interval> alpha, const <interval_array> tp,
                   <interval_array> x, int incx );
```

{TR,TB,TP}SV_I (Interval triangular solve with a vector) $\mathbf{x} \leftarrow \alpha \mathbf{T}^{-1}\mathbf{x}$, $\mathbf{x} \leftarrow \alpha \mathbf{T}^{-T}\mathbf{x}$

These routines bound one of the systems of equations $\mathbf{x} \leftarrow \alpha \mathbf{T}^{-1}\mathbf{x}$ or $\mathbf{x} \leftarrow \alpha \mathbf{T}^{-T}\mathbf{x}$, where $\mathbf{x}$ is an inverval vector and the matrix $\mathbf{T}$ is a upper or lower triangular (or triangular banded or triangular packed) interval matrix. If n is less than or equal to zero, this function returns immediately. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler. If ldt is less than one or ldt is less than n, an error flag is set and passed to the error handler.

Advice to users and implementors: Checking for singularity, or near singularity is not specified for these triangular solvers. Users should perform such a test before calling the triangular solver if their applications require such a test.

- Fortran 95 binding:

```
Triangular:
      SUBROUTINE trsv_i( t, x [, uplo] [, transt] [, diag] [, alpha] )
Triangular Band:
      SUBROUTINE tbsv_i( t, x [, uplo] [, transt] [, diag] [, alpha] )
Triangular Packed:
      SUBROUTINE tpsv_i( tp, x [, uplo] [, transt] [, diag] [, alpha] )
all:
        TYPE(INTERVAL) (<wp>), INTENT(IN) :: t(:,:), tp(:)
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: x(:)
        TYPE(blas_uplo_type),  INTENT(IN), OPTIONAL :: uplo
        TYPE(blas_trans_type), INTENT(IN), OPTIONAL :: transt
        TYPE(blas_diag_type),  INTENT(IN), OPTIONAL :: diag
        TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha
      where
        x has shape (n)
       TR  t has shape (n,n)
```

```
        TB  t has shape (k+1,n) where k = band width
        TP  tp has shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
Triangular:
      SUBROUTINE BLAS_xTRSV_I( UPLO, TRANS, DIAG, N, ALPHA, T, LDT, X,
     $                         INCX )
Triangular Band:
      SUBROUTINE BLAS_xTBSV_I( UPLO, TRANS, DIAG, N, K, ALPHA, T, LDT,
     $                         X, INCX )
Triangular Packed:
      SUBROUTINE BLAS_xTPSV_I( UPLO, TRANS, DIAG, N, ALPHA, TP, X, INCX )
all:
      INTEGER            DIAG, INCX, K, LDT, N, TRANS, UPLO
      <type>             ALPHA( 2 )
      <type>             T( 2, LDA, * ) or TP( 2, * ), X( 2, * )
```

- C binding:

```
Triangular:
void BLAS_xtrsv_i( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, enum blas_diag_type diag, int n,
                   const <interval> alpha, const <interval_array> t, int ldt,
                   <interval_array> x, int incx );
Triangular Band:
void BLAS_xtbsv_i( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, enum blas_diag_type diag, int n,
                   int k, const <interval> alpha, const <interval_array> t,
                   int ldt, <interval_array> x, int incx );
Triangular Packed:
void BLAS_xtpsv_i( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, enum blas_diag_type diag, int n,
                   const <interval> alpha, const <interval_array> tp,
                   <interval_array> x, int incx );
```

GER_I (Rank one update) $\mathbf{A} \leftarrow \alpha \mathbf{x}\mathbf{y}^T + \beta \mathbf{A}$

This routine performs the operation $\mathbf{A} \leftarrow \alpha \mathbf{x}\mathbf{y}^T + \beta \mathbf{A}$, where α and β are intervals, $\mathbf{x}$ and $\mathbf{y}$ are interval vectors, and $\mathbf{A}$ is an interval matrix. This routine returns $\mathbf{A}$ immediately if $\alpha = [0, 0]$ and $\beta = [1, 1]$. If m or n is less than or equal to zero, this function returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or lda is less than m, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
      SUBROUTINE ger_i( a, x, y [, alpha] [, beta] )
      TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: a(:,:)
      TYPE(INTERVAL) (<wp>), INTENT(IN)    :: x(:), y(:)
      TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
      x and y have shape (n)
      a has shape (n,n)
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xGER_I( M, N, ALPHA, X, INCX, Y, INCY, BETA, A, LDA )
      INTEGER            INCX, INCY, LDA, M, N
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             A( 2, LDA, * ), X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xger_i( int m, int n, <interval> alpha, const <interval_array> x,
                  int incx, const <interval_array> y, int incy, <interval> beta,
                  <interval_array> a, int lda );
```

{SY,SP}R_I (Symmetric rank one update) $\mathbf{A} \leftarrow \alpha \mathbf{x}\mathbf{x}^T + \beta \mathbf{A}$ with $\mathbf{A} = \mathbf{A}^T$

This routine performs the symmetric update $\mathbf{A} \leftarrow \alpha \mathbf{x}\mathbf{y}^T + \beta \mathbf{A}$, where α and β are intervals, $\mathbf{x}$ is an interval vector, and $\mathbf{A}$ is a symmetric interval matrix. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or lda is less than n, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
Symmetric:
      SUBROUTINE syr_i( a, x [, uplo] [, alpha] [, beta] )
Symmetric Packed:
      SUBROUTINE spr_i( ap, x [, uplo] [, alpha] [, beta] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: a(:,:), ap(:)
      TYPE(INTERVAL) (<wp>), INTENT(IN)    :: x(:)
      TYPE(blas_uplo_type),  OPTIONAL :: uplo
      TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
      x has shape (n)
     SY  a has shape (n,n)
     SP  ap has shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
Symmetric:
      SUBROUTINE BLAS_xSYR_I( UPLO, N, ALPHA, X, INCX, BETA, A, LDA )
Symmetric Packed:
      SUBROUTINE BLAS_xSPR_I( UPLO, N, ALPHA, X, INCX, BETA, AP )
all:
      INTEGER            INCX, LDA, N, UPLO
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             A( 2, LDA, * ) or AP( 2, * ), X( 2, * )
```

- C binding:

```
Symmetric:
void BLAS_xsyr_i( enum blas_order_type order, enum blas_uplo_type uplo, int n,
                  <interval> alpha, const <interval_array> x, int incx,
                  <interval> beta, <interval_array> a, int lda );
Symmetric Packed:
void BLAS_xspr_i( enum blas_order_type order, enum blas_uplo_type uplo, int n,
                  <interval> alpha, const <interval_array> x, int incx,
                  <interval> beta, <interval_array> ap );
```

Interval Matrix Operations

{GE,GB,SY,SB,SP,TR,TB,TP}_NORM_I (Interval matrix norms)

$$r \leftarrow ||\mathbf{A}||_1, \; ||\mathbf{A}||_F, \; ||\mathbf{A}||_\infty, \text{ or } ||\mathbf{A}||_{\max}$$

These routines compute the one-norm, Frobenius-norm, infinity-norm, or max-norm of a general interval matrix **A** depending on the value passed as the norm operator argument. This routine returns immediately with the output scalar r set to zero if m (for nonsymmetric matrices) or n is less than or equal to zero. For the routine GE_NORM_I, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine GB_NORM_I, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler. For the routines SY_NORM_I and TR_NORM_I, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routines SB_NORM_I and TB_NORM_I, if lda is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
      REAL (<wp>)  FUNCTION ge_norm_i( a [, norm] )
General Band:
      REAL (<wp>)  FUNCTION gb_norm_i( a, m, kl [, norm] )
Symmetric:
      REAL (<wp>)  FUNCTION sy_norm_i( a [, norm] [, uplo] )
Symmetric Band:
      REAL (<wp>)  FUNCTION sb_norm_i( a [, norm] [, uplo] )
Symmetric Packed:
      REAL (<wp>)  FUNCTION sp_norm_i( ap [, norm] [, uplo] )
```

```
Triangular:
      REAL (<wp>)  FUNCTION tr_norm_i( a [, norm] [, uplo] [, diag] )
Triangular Band:
      REAL (<wp>)  FUNCTION tb_norm_i( a [, norm] [, uplo] [, diag] )
Triangular Packed:
      REAL (<wp>)  FUNCTION tp_norm_i( ap [, norm] [, uplo] [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:)
      INTEGER, INTENT(IN) :: m, kl
      TYPE(blas_norm_type), INTENT(IN), OPTIONAL :: norm
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
      a has shape (m,n) for general matrix
                  (l,n) for general banded matrix (l > kl)
                  (n,n) for symmetric or triangular
                  (k+1,n) for symmetric banded, or triangular
                          banded (k=band width)
      ap has shape (n*(n+1)/2).
```

- Fortran 77 binding:

```
General:
      <type>  FUNCTION BLAS_xGE_NORM_I( NORM, M, N, A, LDA )
General Band:
      <type>  FUNCTION BLAS_xGB_NORM_I( NORM, M, N, KL, KU, A, LDA )
Symmetric:
      <type>  FUNCTION BLAS_xSY_NORM_I( NORM, UPLO, N, A, LDA )
Symmetric Band:
      <type>  FUNCTION BLAS_xSB_NORM_I( NORM, UPLO, N, K, A, LDA )
Symmetric Packed:
      <type>  FUNCTION BLAS_xSP_NORM_I( NORM, UPLO, N, AP )
Triangular:
      <type>  FUNCTION BLAS_xTR_NORM_I( NORM, UPLO, DIAG, N, A, LDA )
Triangular Band:
      <type>  FUNCTION BLAS_xTB_NORM_I( NORM, UPLO, DIAG, N, K, A, LDA )
Triangular Packed:
      <type>  FUNCTION BLAS_xTP_NORM_I( NORM, UPLO, DIAG, N, AP )
all:
      INTEGER            DIAG, K, KL, KU, LDA, M, N, NORM, UPLO
      <type>             A( 2, LDA, * ) or AP( 2, * )
```

- C binding:

```
General:
void BLAS_xge_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                      int m, int n, const <interval_array> a, int lda,
```

```
                             SCALAR_INOUT r );
      General Band:
      void BLAS_xgb_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                            int m, int n, int kl, int ku, const <interval_array> a,
                            int lda, SCALAR_INOUT r );
      Symmetric:
      void BLAS_xsy_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                            enum blas_uplo_type uplo, int n,
                            const <interval_array> a, int lda, SCALAR_INOUT  r );
      Symmetric Band:
      void BLAS_xsb_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                            enum blas_uplo_type uplo, int n, int k,
                            const <interval_array> a, int lda, SCALAR_INOUT r );
      Symmetric Packed:
      void BLAS_xsp_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                            enum blas_uplo_type uplo, int n,
                            const <interval_array> ap, SCALAR_INOUT r );
      Triangular:
      void BLAS_xtr_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                            enum blas_uplo_type uplo, enum blas_diag_type diag,
                            int n, const <interval_array> a, int lda,
                            SCALAR_INOUT r );
      Triangular Band:
      void BLAS_xtb_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                            enum blas_uplo_type uplo, enum blas_diag_type diag,
                            int n, int k, const <interval_array> a, int lda,
                            SCALAR_INOUT  r );
      Triangular Packed:
      void BLAS_xtp_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                            enum blas_uplo_type uplo, enum blas_diag_type diag,
                            int n, const <interval_array> ap, SCALAR_INOUT  r );
```

Advice to implementors: In finite precision floating point arithmetic, an upper bound, preferably the least machine representable upper bound, for the mathematical value should be returned for the norms.

{GE,GB}_DIAG_SCALE_I (Diagonal scaling an interval matrix) $\mathbf{A} \leftarrow \mathbf{DA}, \mathbf{AD}$ with $\mathbf{D}$ diagonal

These routines scale a general (or banded) interval matrix $\mathbf{A}$ on the left side or the right side by a diagonal interval matrix $\mathbf{D}$. This routine returns immediately if m or n is less than or equal to zero. As described in section 2.5.3, the value incd less than zero is permitted. However, if incd is equal to zero, an error flag is set and passed to the error handler. For the routine GE_DIAG_SCALE_I, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine GB_DIAG_SCALE_I, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_diag_scale_i( d, a [, side )
General Band:
      SUBROUTINE gb_diag_scale_i( d, a, m, kl [, side] )
all:
      TYPE(INTERVAL) (<wp>), INTENT (IN) :: d(:)
      TYPE(INTERVAL) (<wp>), INTENT (INOUT) :: a(:,:)
      INTEGER, INTENT(IN) :: m, kl
      TYPE(blas_side_type), INTENT (IN), OPTIONAL :: side
    where
      a has shape (m,n) for general matrix
                  (l,n) for general banded matrix (l > kl)
      d has shape (p) where p = m if side = blas_left_side
                            p = n if side = blas_right_side
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_DIAG_SCALE_I( SIDE, M, N, D, INCD, A, LDA )
General Band:
      SUBROUTINE BLAS_xGB_DIAG_SCALE_I( SIDE, M, N, KL, KU, D, INCD, A,
     $                                  LDA )
all:
      INTEGER            INCD, KL, KU, LDA, M, N, SIDE
      <type>             A( 2, LDA, * ), D( 2, * )
```

- C binding:

```
General:
void BLAS_xge_diag_scale_i( enum blas_order_type order,
                            enum blas_side_type side, int m, int n,
                            const <interval_array> d, int incd,
                            <interval_array> a, int lda );
General Band:
void BLAS_xgb_diag_scale_i( enum blas_order_type order,
                            enum blas_side_type side, int m, int n, int kl,
                            int ku, const <interval_array> d, int incd,
                            <interval_array> a, int lda );
```

{GE,GB}_LRSCALE_I (Two-sided diagonal scaling) $\mathbf{A} \leftarrow \mathbf{D}_L\mathbf{A}\mathbf{D}_R$ with $\mathbf{D}_L, \mathbf{D}_R$ diagonal

These routines scale a general (or banded) interval matrix $\mathbf{A}$ on the left side by an interval diagonal matrix $\mathbf{D}_L$ and on the right side by an interval diagonal matrix $\mathbf{D}_R$. This routine returns immediately if m or n is less than or equal to zero. As described in section 2.5.3, the value incdl or incdu less than zero is permitted. However, if either incdl or incdu is equal to zero, an error flag is set and passed to the error handler. For the routine GE_LRSCALE_I, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine GB_LRSCALE_I, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_lrscale_i( dl, dr, a )
General Band:
      SUBROUTINE gb_lrscale_i( dl, dr, a, m, kl )
all:
        TYPE(INTERVAL) (<wp>), INTENT(IN) :: dl(:), dr(:)
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: a(:,:)
        INTEGER, INTENT(IN) :: m, kl
      where
        a has shape (m,n) for general matrix
                    (l,n) for general banded matrix (l > kl)
        dl has shape (m)
        dr has shape (n)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_LRSCALE_I( M, N, DL, INCDL, DR, INCDR, A, LDA )
General Band:
      SUBROUTINE BLAS_xGB_LRSCALE_I( M, N, KL, KU, DL, INCDL, DR, INCDR, A,
     $                               LDA )
all:
      INTEGER            INCDL, INCDR, KL, KU, LDA, M, N
      <type>             A( 2, LDA, * ), DL( 2, * ), DR( 2, * )
```

- C binding:

```
General:
void BLAS_xge_lrscale_i( enum blas_order_type order, int m, int n,
                         const <interval_array> dl, int incdl,
                         const <interval_array> dr, int incdr,
                         <interval_array> a, int lda );
General Band:
void BLAS_xgb_lrscale_i( enum blas_order_type order, int m, int n,
                         int kl, int ku, const <interval_array> dl,
                         int incdl, const <interval_array> dr, int incdr,
                         <interval_array> a, int lda );
```

{SY,SB,SP}_LRSCALE_I (Two-sided diagonal scaling) $\mathbf{A} \leftarrow \mathbf{DAD}$ with $\mathbf{A} = \mathbf{A}^T$.

These routines perform a two-sided scaling on a symmetric (or symmetric banded or symmetric packed) interval matrix **A** by an interval diagonal matrix **D**. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incd less than zero is permitted. However, if incd is equal to zero, an error flag is set and passed to the error handler. For the routines SY_LRSCALE_I and SP_LRSCALE_I, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine SB_LRSCALE_I, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
Symmetric:
      SUBROUTINE sy_lrscale_i( d, a [, uplo] )
Symmetric Band:
      SUBROUTINE sb_lrscale_i( d, a [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_lrscale_i( d, ap [, uplo] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: d(:)
      TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: a(:,:) | ap(:)
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
    where
      a has shape (n,n) for symmetric
                  (k+1,n) for symmetric banded (k=band width)
      ap has shape (n*(n+1)/2).
      d has shape (n)
```

- Fortran 77 binding:

```
Symmetric:
      SUBROUTINE BLAS_xSY_LRSCALE_I( UPLO, N, D, INCD, A, LDA )
Symmetric Band:
      SUBROUTINE BLAS_xSB_LRSCALE_I( UPLO, N, K, D, INCD, A, LDA )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_LRSCALE_I( UPLO, N, D, INCD, AP )
all:
      INTEGER            INCD, K, LDA, N, UPLO
      <type>             A( 2, LDA, * ) or AP( 2, * ), D( 2, * )
```

- C binding:

```
Symmetric:
void BLAS_xsy_lrscale_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         int n, const <interval_array> d, int incd,
                         <interval_array> a, int lda );
Symmetric Band:
void BLAS_xsb_lrscale_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         int n, int k, const <interval_array> d, int incd,
```

```
                             <interval_array> a, int lda );
Symmetric Packed:
void BLAS_xsp_lrscale_i( enum blas_order_type order, enum blas_uplo_type uplo,
                             int n, const <interval_array> d, int incd,
                             <interval_array> ap );
```

{GE,SY,SB,SP}_ACC_I (Matrix accumulation and scale) $\mathbf{B} \leftarrow \alpha\mathbf{A} + \beta\mathbf{B}, \mathbf{B} \leftarrow \alpha\mathbf{A}^T + \beta\mathbf{B}$.

These routines scale an interval matrix $\mathbf{A}$ (or its transpose) and scale an interval matrix $\mathbf{B}$ and accumulate the result in the interval matrix $\mathbf{B}$. Matrices $\mathbf{A}$ (or $\mathbf{A}^T$) and $\mathbf{B}$ have the same storage format. This routine returns immediately if m (for nonsymmetric matrices) or n or k (for symmetric band matrices) is less than or equal to zero. For the routine GE_ACC_I, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine SY_ACC_I, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine SB_ACC_I, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_acc_i( a, b [, transa] [, alpha] [, beta] )
Symmetric:
      SUBROUTINE sy_acc_i( a, b [, uplo], [, transa] [, alpha] [, beta] )
Symmetric Band:
      SUBROUTINE sb_acc_i( a, b [, uplo], [, transa] [, alpha] [, beta] )
Symmetric Packed:
      SUBROUTINE sp_acc_i( ap, bp [, uplo], [, transa] [, alpha] [, beta] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: b(:,:) | bp(:)
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:)
      TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      TYPE(blas_trans_type), INTENT(IN), OPTIONAL :: transa
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
    where
       GE  a and b have shape (m,n) if transa = blas_no_trans (the default)
           a has shape (n,m) and b has shape (m,n) if transa /= blas_no_trans
       SY  a and b have shape (n,n)
       SB  a and b have shape (p+1,n) (p = band width)
       SP  ap and bp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_ACC_I( TRANS, M, N, ALPHA, A, LDA, BETA, B, LDB )
Symmetric:
      SUBROUTINE BLAS_xSY_ACC_I( UPLO, TRANS, N, ALPHA, A, LDA, BETA, B,
```

```
     $                           LDB )
Symmetric Band:
      SUBROUTINE BLAS_xSB_ACC_I( UPLO, TRANS, N, K, ALPHA, A, LDA, BETA,
     $                           B, LDB )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_ACC_I( UPLO, TRANS, N, ALPHA, AP, BETA, BP )
all:
      INTEGER            K, LDA, LDB, M, N, TRANS, UPLO
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDB, * )
     $                   or BP( 2, * )
```

- C binding:

```
General:
void BLAS_xge_acc_i( enum blas_order_type order, enum blas_trans_type trans,
                     int m, int n, <interval> alpha, const <interval_array> a,
                     int lda, <interval> beta, <interval_array> b, int ldb );
Symmetric:
void BLAS_xsy_acc_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_trans_type trans, int n, <interval> alpha,
                     const <interval_array> a, int lda, <interval> beta,
                     <interval_array> b, int ldb );
Symmetric Band:
void BLAS_xsb_acc_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_trans_type trans, int n, int k, <interval> alpha,
                     const <interval_array> a, int lda, <interval> beta,
                     <interval_array> b, int ldb );
Symmetric Packed:
void BLAS_xsp_acc_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_trans_type trans, int n, <interval> alpha,
                     const <interval_array> ap, <interval> beta,
                     <interval_array> bp );
```

{GB,TR,TB,TP}_ACC_I (Matrix accumulation and scale) $\mathbf{B} \leftarrow \alpha\mathbf{A} + \beta\mathbf{B}$.

These routines scale interval matrices **A** and **B** and accumulate the result in the matrix **B**. Matrices A and B have the same storage format. This routine returns immediately if m or kl or ku (for general band matrices) or n or k (for triangular band matrices) is less than or equal to zero. For the routine GB_ACC_I, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler. For the routines TR_ACC_I and TP_ACC_I, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine TB_ACC_I, if lda is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General Band:
```

```
          SUBROUTINE gb_acc_i( a, m, kl, b [, alpha] [, beta] )
    Triangular:
          SUBROUTINE tr_acc_i( a, b [, uplo], [, diag] [, alpha] [, beta] )
    Triangular Band:
          SUBROUTINE tb_acc_i( a, b [, uplo], [, diag] [, alpha] [, beta] )
    Triangular Packed:
          SUBROUTINE tp_acc_i( ap, bp [, uplo], [, diag] [, alpha] [, beta] )
    all:

          TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:)
          INTEGER, INTENT(IN) :: m, kl
          TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: b(:,:) | bp(:)
          TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
          TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
          TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
        where
          a and b have shape (l,n) for general banded matrix (l > kl)
          a and b have shape (n,n) for triangular matrix
          a and b have shape (p+1,n) for triangular banded matrix
          ap and bp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
    General Band:
          SUBROUTINE BLAS_xGB_ACC_I( M, N, KL, KU, ALPHA, A, LDA, BETA, B,
         $                           LDB )
    Triangular:
          SUBROUTINE BLAS_xTR_ACC_I( UPLO, DIAG, N, ALPHA, A, LDA, BETA, B,
         $                           LDB )
    Triangular Band:
          SUBROUTINE BLAS_xTB_ACC_I( UPLO, DIAG, N, K, ALPHA, A, LDA, BETA,
         $                           B, LDB )
    Triangular Packed:
          SUBROUTINE BLAS_xTP_ACC_I( UPLO, DIAG, N, ALPHA, AP, BETA, BP )
    all:
          INTEGER           DIAG, K, KL, KU, LDA, LDB, M, N, UPLO
          <type>            ALPHA( 2 ), BETA( 2 )
          <type>            A( 2, LDA, * ) or AP( 2, * ), B( 2, LDB, * )
         $                  or BP( 2, * )
```

- C binding:

```
    General Band:
    void BLAS_xgb_acc_i( enum blas_order_type order, int m, int n, int kl, int ku,
                         <interval> alpha, <interval_array> a, int lda,
                         <interval> beta, <interval_array> b, int ldb );
    Triangular:
```

```
void BLAS_xtr_acc_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_diag_type diag, int n, <interval> alpha,
                     <interval_array> a, int lda, <interval> beta,
                     <interval_array> b, int ldb );
Triangular Band:
void BLAS_xtb_acc_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_diag_type diag, int n, int k, <interval> alpha,
                     <interval_array> a, int lda, <interval> beta,
                     <interval_array> b, int ldb );
Triangular Packed:
void BLAS_xtp_acc_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_diag_type diag, int n, <interval> alpha,
                     <interval_array> ap, <interval> beta,
                     <interval_array> bp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_ADD_I (Matrix add and scale) $\mathbf{C} \leftarrow \alpha\mathbf{A} + \beta\mathbf{B}$

These routines scale two interval matrices **A** and **B** and store the sum in the matrix **C**. Matrices A, B, and C have the same storage format. This routine returns immediately if m or kl or ku (for general band matrices) or n or k (for symmetric or triangular band matrices) is less than or equal to zero. For the routine GE_ADD_I, if lda is less than one or less than m, an error flag is set and passed to the error handler. For the routine GB_ADD_I, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler. For the routines SY_ADD_I, TR_ADD_I, SP_ADD_I, and TP_ADD_I, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routines SB_ADD_I and TB_ADD_I, if lda is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_add_i( a, b, c [, alpha] [, beta] )
General Band:
      SUBROUTINE gb_add_i( a, m, kl, b, c [, alpha] [, beta] )
Symmetric:
      SUBROUTINE sy_add_i( a, b, c [, uplo], [, alpha] [, beta] )
Symmetric Band:
      SUBROUTINE sb_add_i( a, b, c [, uplo], [, alpha] [, beta] )
Symmetric Packed:
      SUBROUTINE sp_add_i( ap, bp, cp [, uplo], [, alpha] [, beta] )
Triangular:
      SUBROUTINE tr_add_i( a, b, c [, uplo], [, diag] [, alpha] [, beta] )
Triangular Band:
      SUBROUTINE tb_add_i( a, b, c [, uplo], [, diag] [, alpha] [, beta] )
Triangular Packed:
      SUBROUTINE tp_add_i( ap, bp, cp [, uplo], [, diag] [, alpha] [, beta] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:), b(:,:) | bp(:)
```

```
      INTEGER, INTENT(IN) :: m, kl
      TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: c(:,:) | cp(:)
      TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
     assuming A, B and C all the same (general, banded or packed) with
     the same size.
      a, b and c have shape (m,n) for general matrix
                            (l,n) for general banded matrix (l > kl)
                            (n,n) for symmetric or triangular
                            (k+1,n) for symmetric banded or triangular
                                    banded (k=band width)
      ap, bp and cp have shape (n*(n+1)/2).
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_ADD_I( M, N, ALPHA, A, LDA, BETA, B, LDB, C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_ADD_I( M, N, KL, KU, ALPHA, A, LDA, BETA, B, LDB,
     $                           C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_ADD_I( UPLO, N, ALPHA, A, LDA, BETA, B, LDB, C,
     $                           LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_ADD_I( UPLO, N, K, ALPHA, A, LDA, BETA, B, LDB,
     $                           C, LDC )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_ADD_I( UPLO, N, ALPHA, AP, BETA, BP, CP )
Triangular:
      SUBROUTINE BLAS_xTR_ADD_I( UPLO, DIAG, N, ALPHA, A, LDA, BETA, B,
     $                           LDB, C, LDC )
Triangular Band:
      SUBROUTINE BLAS_xTB_ADD_I( UPLO, DIAG, N, K, ALPHA, A, LDA, BETA, B,
     $                           LDB, C, LDC )
Triangular Packed:
      SUBROUTINE BLAS_xTP_ADD_I( UPLO, DIAG, N, ALPHA, AP, BETA, BP, CP )
all:
      INTEGER            DIAG, K, KL, KU, LDA, LDB, M, N, TRANS, UPLO
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDB, * )
     $                   or BP( 2, * ), C( 2, LDC, * ) or CP( 2, * )
```

- C binding:

```
General:
```

```
void BLAS_xge_add_i( enum blas_order_type order, int m, int n, <interval> alpha,
                     const <interval_array> a, int lda, <interval> beta,
                     const <interval_array> b, int ldb, <interval_array> c,
                     int ldc );
General Band:
void BLAS_xgb_add_i( enum blas_order_type order, int m, int n, int kl, int ku,
                     <interval> alpha, const <interval_array> a, int lda,
                     <interval> beta, const <interval_array> b, int ldb,
                     <interval_array> c, int ldc );
Symmetric:
void BLAS_xsy_add_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, <interval> alpha, const <interval_array> a,
                     int lda, <interval> beta, const <interval_array> b,
                     int ldb, <interval_array> c, int ldc );
Symmetric Band:
void BLAS_xsb_add_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, int k, <interval> alpha, const <interval_array> a,
                     int lda, <interval> beta, const <interval_array> b,
                     int ldb, <interval_array> c, int ldc );
Symmetric Packed:
void BLAS_xsp_add_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, <interval> alpha, const <interval_array> ap,
                     <interval> beta, const <interval_array> bp,
                     <interval_array> cp );
Triangular:
void BLAS_xtr_add_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_diag_type diag, int n, <interval> alpha,
                     const <interval_array> a, int lda, <interval> beta,
                     const <interval_array> b, int ldb, <interval_array> c,
                     int ldc );
Triangular Band:
void BLAS_xtb_add_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, enum blas_diag_type diag, int k, <interval> alpha,
                     const <interval_array> a, int lda, <interval> beta,
                     const <interval_array> b, int ldb, <interval_array> c,
                     int ldc );
Triangular Packed:
void BLAS_xtp_add_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, enum blas_diag_type diag, <interval> alpha,
                     const <interval_array> ap, <interval> beta,
                     const <interval_array> bp, <interval_array> cp );
```

Interval Matrix-Matrix Operations

In the following specifications, $\mathrm{op}(X)$ denotes X or X^T where X is a matrix.

GEMM_I (General interval matrix matrix product) $\mathbf{C} \leftarrow \alpha\ \mathrm{op}(\mathbf{A})\ \mathrm{op}(\mathbf{B}) + \beta\mathbf{C}$.

This routine performs a general interval matrix matrix multiply $\mathbf{C} \leftarrow \alpha \text{ op}(\mathbf{A}) \text{ op}(\mathbf{B}) + \beta\mathbf{C}$, where α and β are intervals, and $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are general interval matrices. This routine returns immediately if m or n or k is less than or equal to zero. If lda is less than one or less than m, or if ldb is less than one or less than k, or if ldc is less than one or less than m, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
    SUBROUTINE gemm_i( a, b, c [, transa] [, transb] [, alpha] [, beta] )
    TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), b(:,:)
    TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: c(:,:)
    TYPE(blas_trans_type), INTENT(IN), OPTIONAL :: transa, transb
    TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
  where
    c has shape (m,n)
    a has shape (m,k) if transa = blas_no_trans (the default)
                (k,m) if transa /= blas_no_trans
    b has shape (k,n) if transb = blas_no_trans (the default)
                (n,k) if transb /= blas_no_trans
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xGEMM_I( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA,
   $                        B, LDB, BETA, C, LDC )
    INTEGER            K, LDA, LDB, LDC, M, N, TRANSA, TRANSB
    <type>             ALPHA( 2 ), BETA( 2 )
    <type>             A( 2, LDA, * ), B( 2, LDB, * ),
   $                   C( 2, LDC, * )
```

- C binding:

```
void BLAS_xgemm_i( enum blas_order_type order, enum blas_trans_type transa,
                   enum blas_trans_type transb, int m, int n, int k,
                   <interval> alpha,  const <interval_array> a, int lda,
                   const <interval_array> b, int ldb, <interval> beta,
                   <interval_array> c, int ldc );
```

SYMM_I (Symmetric interval matrix matrix product) $\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta\mathbf{C}$ or $\mathbf{C} \leftarrow \alpha\mathbf{BA} + \beta\mathbf{C}$.

These routines perform one of the symmetric interval matrix operations $\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta\mathbf{C}$ or $\mathbf{C} \leftarrow \alpha\mathbf{BA} + \beta\mathbf{C}$ where α and β are intervals, $\mathbf{A}$ is a symmetric interval matrix, and $\mathbf{B}$ and $\mathbf{C}$ are general interval matrices. This routine returns immediately if m or n is less than or equal to zero. For side equal to blas_left_side, and if lda is less than one or less than m, or if ldb is less than one or less than m, or if ldc is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if lda is less than one or less than n, or if ldb is less than one or less than n, or if ldc is less than one or less than n, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
      SUBROUTINE symm_i( a, b, c [, side] [, uplo] [, alpha] [, beta] )
      TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: c(:,:)
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), b(:,:)
      TYPE(blas_side_type), INTENT(IN), OPTIONAL :: side
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
      c has shape (m,n), b same shape as c
       SY  a has shape (m,m) if side = blas_left_side (the default)
           a has shape (n,n) if side /= blas_left_side
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xSYMM_I( SIDE, UPLO, M, N, ALPHA, A, LDA, B,
     $                         LDB, BETA, C, LDC )
      INTEGER            LDA, LDB, LDC, M, N, SIDE, UPLO
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             A( 2, LDA, * ), B( 2, LDB, * ),
     $                   C( 2, LDC, * )
```

- C binding:

```
void BLAS_xsymm_i( enum blas_order_type order, enum blas_side_type side,
                   enum blas_uplo_type uplo, int m, int n, <interval> alpha,
                   const <interval_array> a, int lda, const <interval_array> b,
                   int ldb, <interval> beta, <interval_array> c, int ldc );
```

TRMM_I (Triangular interval matrix matrix product)

$$\mathbf{B} \leftarrow \alpha\mathbf{TB}, \mathbf{B} \leftarrow \alpha\mathbf{T}^T\mathbf{B}, \mathbf{B} \leftarrow \alpha\mathbf{BT}, \text{ or } \mathbf{B} \leftarrow \alpha\mathbf{BT}^T$$

These routines perform one of the interval matrix operations $\mathbf{B} \leftarrow \alpha\mathbf{TB}, \mathbf{B} \leftarrow \alpha\mathbf{T}^T\mathbf{B}$, $\mathbf{B} \leftarrow \alpha\mathbf{BT}$, or $\mathbf{B} \leftarrow \alpha\mathbf{BT}^T$, where α is an interval, $\mathbf{T}$ is a unit, or non-unit, upper or lower triangular interval matrix, and $\mathbf{B}$ is a general interval matrix. This routine returns immediately if m or n is less than or equal to zero. For side equal to blas_left_side, and if ldt is less than one or less than m, or if ldb is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if ldt is less than one or less than n, or if ldb is less than one or less than m, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
      SUBROUTINE trmm_i( t, b [, side] [, uplo] [, transt] [, diag] &
                         [, alpha] )
      TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: b(:,:)
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: t(:,:)
```

```
          TYPE(blas_side_type),  INTENT(IN), OPTIONAL :: side
          TYPE(blas_trans_type), INTENT(IN), OPTIONAL :: transt
          TYPE(blas_diag_type),  INTENT(IN), OPTIONAL :: diag
          TYPE(blas_uplo_type),  INTENT(IN), OPTIONAL :: uplo
          TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
        where
          b has shape (m,n)
           TR  t has shape (m,m) if side = blas_left_side (the default)
               t has shape (n,n) if side /= blas_left_side
```

- Fortran 77 binding:

```
          SUBROUTINE BLAS_xTRMM_I( SIDE, UPLO, TRANST, DIAG, M, N, ALPHA,
         $                         T, LDT, B, LDB )
          INTEGER            DIAG, LDA, LDB, M, N, SIDE, TRANST, UPLO
          <type>             ALPHA( 2 )
          <type>             T( 2, LDA, * ), B( 2, LDB, * )
```

- C binding:

```
void BLAS_xtrmm_i( enum blas_order_type order, enum blas_side_type side,
                   enum blas_uplo_type uplo, enum blas_trans_type transt,
                   enum blas_diag_type diag, int m, int n, <interval> alpha,
                   const <interval_array> t, int ldt, <interval_array> b,
                   int ldb );
```

TRSM_I (Interval triangular solve)

$$\mathbf{B} \leftarrow \alpha\mathbf{T}^{-1}\mathbf{B},\ \mathbf{B} \leftarrow \alpha(\mathbf{T}^{-1})^T\mathbf{B},\ \mathbf{B} \leftarrow \alpha\mathbf{B}\mathbf{T}^{-1} \text{ or } \mathbf{B} \leftarrow \alpha\mathbf{B}(\mathbf{T}^{-1})^T$$

These routines bound one of the matrix equations $\mathbf{B} \leftarrow \alpha\mathbf{T}^{-1}\mathbf{B}$, $\mathbf{B} \leftarrow \alpha(\mathbf{T}^{-1})^T\mathbf{B}$, $\mathbf{B} \leftarrow \alpha\mathbf{B}\mathbf{T}^{-1}$ or $\mathbf{B} \leftarrow \alpha\mathbf{B}(\mathbf{T}^{-1})^T$ where α is an interval, $\mathbf{B}$ is a general interval matrix, and $\mathbf{T}$ is a a unit, or non-unit, upper or lower triangular interval matrix. This routine returns immediately if m or n is less than or equal to zero. For side equal to blas_left_side, and if ldT is less than one or less than m, or if ldb is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if ldt is less than one or less than n, or if ldb is less than one or less than m, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
          SUBROUTINE trsm_i( t, b [, side] [, uplo] [, transt] [, diag] [, alpha] )
          TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: b(:,:)
          TYPE(INTERVAL) (<wp>), INTENT(IN) :: t(:,:)
          TYPE(blas_side_type),  INTENT(IN), OPTIONAL :: side
          TYPE(blas_trans_type), INTENT(IN), OPTIONAL :: transt
          TYPE(blas_diag_type),  INTENT(IN), OPTIONAL :: diag
          TYPE(blas_uplo_type),  INTENT(IN), OPTIONAL :: uplo
```

```
    TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
  where
    b has shape (m,n)
     TR  t has shape (m,m) if side = blas_left_side (the default)
         t has shape (n,n) if side /= blas_left_side
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xTRSM_I( SIDE, UPLO, TRANST, DIAG, M, N, ALPHA,
   $                         ALPHA, T, LDT, B, LDB )
    INTEGER            DIAG, LDB, LDT, M, N, SIDE, TRANST, UPLO
    <type>             ALPHA( 2 )
    <type>             T( 2, LDA, * ), B( 2, LDB, * )
```

- C binding:

```
void BLAS_xtrsm_i( enum blas_order_type order, enum blas_side_type side,
                   enum blas_uplo_type uplo, enum blas_trans_type transt,
                   enum blas_diag_type diag, int m, int n, <interval> alpha,
                   const <interval_array> t, int ldt, <interval_array> b,
                   int ldb );
```

Data Movement with Interval Matrices

{GE,GB,SY,SB,SP,TR,TB,TP}_COPY_I (Matrix copy) $\mathbf{B} \leftarrow \mathbf{A}, \mathbf{B} \leftarrow \mathbf{A}^T$

This routine copies an interval matrix (or its transpose) $\mathbf{A}$ and stores the result in an interval matrix $\mathbf{B}$. Matrices A (or $\mathbf{A}^T$) and B have the same storage format. This routine returns immediately if m (for nonsymmetric matrices), n, k (for symmetric band matrices), or kl or ku (for general band matrices), is less than or equal to zero. For the routine GE_COPY_I, if trans equal to blas_no_trans, and if lda is less than one or less than m, or if ldb is less than one or less than m, an error flag is set and passed to the error handler. For the routine GE_COPY_I, if trans equal to blas_trans, and if lda is less than one or less than m, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For the routine GB_COPY_I, if lda is less than kl plus ku plus one, or if ldb is less than kl plus ku plus one, an error flag is set and passed to the error handler. For the routines SY_COPY_I and TR_COPY_I, if lda is less than one or less than n, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For the routines SB_COPY_I and TB_COPY_I, if lda is less than k plus one, or if ldb is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_copy_i( a, b [, transa] )
General Band:
      SUBROUTINE gb_copy_i( a, kl, b [, transa] )
Symmetric:
```

```
            SUBROUTINE sy_copy_i( a, b [, uplo] )
      Symmetric Band:
            SUBROUTINE sb_copy_i( a, b [, uplo] )
      Symmetric Packed:
            SUBROUTINE sp_copy_i( ap, bp [, uplo] )
      Triangular:
            SUBROUTINE tr_copy_i( a, b [, uplo], [, trans] [, diag] )
      Triangular Band:
            SUBROUTINE tb_copy_i( a, b [, uplo], [, trans] [, diag] )
      Triangular Packed:
            SUBROUTINE tp_copy_i( ap, bp [, uplo], [, trans] [, diag] )
      all:
            TYPE(INTERVAL) (<wp>), INTENT(OUT) :: b(:,:) | bp(:)
            TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:)
            INTEGER, INTENT(IN) :: kl
            TYPE(blas_trans_type),INTENT(IN), OPTIONAL :: trans
            TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
            TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
          where
           If trans = blas_no_trans (the default)
              a, b have shape (m,n) for general matrix
                              (l,n) for general banded matrix (l > kl)
                              (n,n) for symmetric or triangular
                              (k+1,n) for symmetric banded or triangular
                                      banded (k=band width)
              ap and bp have shape (n*(n+1)/2).

           If trans \= blas_no_trans
              a has shape (m,n) and b has shape (n,m) for general matrix
                          (l,n) and b has shape (l,m) for general banded matrix (l>kl)
              a and b have shape (n,n) for symmetric or triangular
                                 (k+1,n) for symmetric banded or triangular
                                         banded (k=band width)
              ap and bp have shape (n*(n+1)/2).
```

- Fortran 77 binding:

```
      General:
            SUBROUTINE BLAS_xGE_COPY_I( TRANS, M, N, A, LDA, B, LDB )
      General Band:
            SUBROUTINE BLAS_xGB_COPY_I( TRANS, M, N, KL, KU, A, LDA, B, LDB )
      Symmetric:
            SUBROUTINE BLAS_xSY_COPY_I( UPLO, N, A, LDA, B, LDB )
      Symmetric Band:
            SUBROUTINE BLAS_xSB_COPY_I( UPLO, N, K, A, LDA, B, LDB )
      Symmetric Packed:
            SUBROUTINE BLAS_xSP_COPY_I( UPLO, N, AP, BP )
      Triangular:
```

```
      SUBROUTINE BLAS_xTR_COPY_I( UPLO, TRANS, DIAG, N, A, LDA, B, LDB )
Triangular Band:
      SUBROUTINE BLAS_xTB_COPY_I( UPLO, TRANS, DIAG, N, K, A, LDA, B, LDB )
Triangular Packed:
      SUBROUTINE BLAS_xTP_COPY_I( UPLO, TRANS, DIAG, N, AP, BP )
all:
      INTEGER            DIAG, LDA, LDB, N, K, KL, KU, TRANS, UPLO
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDB, * )
     $                   or BP( 2, * )
```

- C binding:

```
General:
void BLAS_xge_copy_i( enum blas_order_type order, enum blas_trans_type transa,
                      int m, int n, const <interval_array> a, int lda,
                      <interval_array> b, int ldb );
General Band:
void BLAS_xgb_copy_i( enum blas_order_type order, enum blas_trans_type transa,
                      int m, int n, int kl, int ku, const <interval_array> a,
                      int lda, <interval_array> b, int ldb );
Symmetric:
void BLAS_xsy_copy_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, const <interval_array> a, int lda,
                      <interval_array> b, int ldb );
Symmetric Band:
void BLAS_xsb_copy_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, int k, const <interval_array> a, int lda,
                      <interval_array> b, int ldb );
Symmetric Packed:
void BLAS_xsp_copy_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, const <interval_array> ap, <interval_array> bp );
Triangular:
void BLAS_xtr_copy_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> a, int lda,
                      <interval_array> b, int ldb );
Triangular Band:
void BLAS_xtb_copy_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, int k, const <interval_array> a, int lda,
                      <interval_array> b, int ldb );
Triangular Packed:
void BLAS_xtp_copy_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> ap, <interval_array> bp );
```

GE_TRANS_I (Matrix transposition) $\mathbf{A} \leftarrow \mathbf{A}^T$

This routine performs the transposition of a square interval matrix $\mathbf{A}$ and overwrites the matrix $\mathbf{A}$. This routine returns immediately if n is less than or equal to zero. If lda is less than one or less than n, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
        SUBROUTINE  ge_trans_i( a )
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: a(:,:)
      where
        a has shape (n,n)
```

- Fortran 77 binding:

```
        SUBROUTINE BLAS_xGE_TRANS_I( N, A, LDA )
        INTEGER            LDA, N
        <type>             A( 2, LDA, * )
```

- C binding:

```
void BLAS_xge_trans_i( int n, <interval_array> a, int lda );
```

GE_PERMUTE_I (Permute an interval matrix) $\mathbf{A} \leftarrow P\mathbf{A}$ or $\mathbf{A} \leftarrow \mathbf{A}P$

This routine permutes the rows or columns of an interval matrix $\mathbf{A}$ by the permutation matrix P. This routine returns immediately if m or n is less than or equal to zero. As described in section 2.5.3, the value incp less than zero is permitted. However, if incp is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or less than m, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
        SUBROUTINE  ge_permute_i( p, a [, side] )
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: a(:,:)
        INTEGER, INTENT(IN) :: p(:)
        TYPE(blas_side_type), INTENT(IN), OPTIONAL :: side
      where
        a has shape (m,n)
        p has shape (k) where k = m if side = blas_left_side
                              k = n if side = blas_right_side
```

- Fortran 77 binding:

```
        SUBROUTINE BLAS_xGE_PERMUTE_I( SIDE, M, N, P, INCP, A, LDA )
        INTEGER            INCP, LDA, M, N, SIDE
        INTEGER            P( * )
        <type>             A( 2, LDA, * )
```

The value of INCP may be positive or negative. A negative value of INCP applies the permutation in the opposite direction.

- C binding:

```
void BLAS_xge_permute_i( enum blas_order_type order, enum blas_side_type side,
                         int m, int n, const int *p, int incp,
                         <interval_array> a, int lda );
```

The value of incp may be positive or negative. A negative value of incp applies the permutation in the opposite direction.

Set Operations Involving Interval Vectors

ENCV_I (Checks if an interval vector is enclosed in another interval vector) True if $\mathbf{x} \subseteq \mathbf{y}$

This routine checks if an interval vector $\mathbf{x}$ is enclosed in another interval vector $\mathbf{y}$. We say that an interval vector $\mathbf{x}$ is enclosed in $\mathbf{y}$, denoted as $\mathbf{x} \subseteq \mathbf{y}$, if and only if $\underline{y_i} \leq \underline{x_i} \leq \overline{x_i} \leq \overline{y_i} \forall i$. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
   LOGICAL FUNCTION encv_i(x, y)
   TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:), y(:)
 where
   x and y have shape (n)
```

- Fortran 77 binding:

```
 LOGICAL FUNCTION BLAS_xENCV_I( N, X, INCX, Y, INCY )
 INTEGER           N, INCX, INCY
 <type>            X( 2, * ), Y( 2, * )
```

- C binding:

```
int BLAS_xencv_i( int n, const <interval_array> x, int incx,
                  const <interval_array> y, int incy );
```

INTERIORV_I (If an interval vector is in the interior of another interval vector) True if $\mathbf{x} \subset \mathbf{y}$

This routine checks if an interval vector $\mathbf{x}$ is enclosed in the interior of another interval vector $\mathbf{y}$. We say that an interval vector $\mathbf{x}$ is enclosed in the interior of $\mathbf{y}$, denoted as $\mathbf{x} \subset \mathbf{y}$, if and only if $\underline{y_i} < \underline{x_i} \leq \overline{x_i} < \overline{y_i} \forall i$. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
LOGICAL FUNCTION interiorv_i(x, y)
TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:), y(:)
where
  x and y have shape (n)
```

- Fortran 77 binding:

```
LOGICAL FUNCTION BLAS_xINTERIORV_I( N, X, INCX, Y, INCY )
INTEGER            N, INCX, INCY
<type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
int BLAS_xinteriorv_i( int n, const <interval_array> x, int incx,
                       const <interval_array> y, int incy );
```

DISJV_I (Checks if two interval vectors disjoint) True if $\mathbf{x} \cap \mathbf{y} = \emptyset$

This routine checks if two interval vectors $\mathbf{x}$ and $\mathbf{y}$ are disjoint, which means that $\mathbf{x}_i \cap \mathbf{y}_i = \emptyset$ for some i. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
LOGICAL FUNCTION disjv_i(x, y)
TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:), y(:)
where
  x and y have shape (n)
```

- Fortran 77 binding:

```
LOGICAL FUNCTION BLAS_xDISJV_I( N, X, INCX, Y, INCY )
INTEGER            N, INCX, INCY
<type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
int BLAS_xdisjv_i( int n, const <interval_array> x, int incx,
                   const <interval_array> y, int incy );
```

INTERV_I (Intersection of an interval vector with another) $\mathbf{y} \leftarrow \mathbf{x} \cap \mathbf{y}$.

This routine finds the intersection of two interval vectors $\mathbf{x}$ and $\mathbf{y}$, and stores the result in $\mathbf{y}$. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
     SUBROUTINE interv_i( x, y )
     TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
     TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: y(:)
   where
     x and y have shape (n)
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xINTERV_I( N, X, INCX, Y, INCY )
    INTEGER            N, INCX, INCY
    <type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xinterv_i( int n, const <interval_array> x, int incx,
                    <interval_array> y, int incy );
```

WINTERV_I (Intersection of two interval vectors) $\mathbf{z} \leftarrow \mathbf{x} \cap \mathbf{y}$.

This routine finds the intersection of two interval vectors $\mathbf{x}$ and $\mathbf{y}$, and stores the result in another interval vector $\mathbf{z}$. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx or incy or incz less than zero is permitted. However, if incx, incy, or incz is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
    SUBROUTINE winterv_i(x, y, z )
    TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:), y(:)
    TYPE(INTERVAL) (<wp>), INTENT(OUT) :: z(:)
  where
    x, y and z have shape (n)
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xWINTERV_I( N, X, INCX, Y, INCY, Z, INCZ )
    INTEGER            SIDE, LDA, M, N
    INTEGER            N, INCX, INCY, INCZ
    <type>             X( 2, * ), Y( 2,* ), Z( 2, * )
```

- C binding:

```
void BLAS_xwinterv_i( int n, const <interval_array> x, int incx,
                      const <interval_array> y, int incy,
                      <interval_array> z, int incz );
```

HULLV_I (Convex hull of an interval vector with another) $\mathbf{y} \leftarrow$ a convex set which contains $\mathbf{x} \cup \mathbf{y}$

This routine computes a convex set which contains both interval vectors $\mathbf{x}$ and $\mathbf{y}$, and overwrites the input interval vector $\mathbf{y}$ with the result. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
      SUBROUTINE hullv_i( x, y )
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
      TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: y(:)
    where
      x and y have shape (n)
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xHULLV_I( N, X, INCX, Y, INCY )
      INTEGER            N, INCX, INCY
      <type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xhullv_i( int n, const <interval_array> x, int incx,
                    <interval_array> y, int incy );
```

WHULLV_I (Convex hull of two interval vectors) $\mathbf{z} \leftarrow$ a convex set which contains $\mathbf{x} \cup \mathbf{y}$.

This routine finds a convex hull of two interval vectors $\mathbf{x}$ and $\mathbf{y}$, and stores the result in another interval vector $\mathbf{z}$. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx incy, or incz less than zero is permitted. However, if incx or incy or incz is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
      SUBROUTINE whullv_i( x, y, z )
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:), y(:)
      TYPE(INTERVAL) (<wp>), INTENT(OUT) :: z(:)
    where
      x, y and z have shape (n)
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xWHULLV_I( N, X, INCX, Y, INCY, Z, INCZ )
      INTEGER            N, INCX, INCY, INCZ
      <type>             X( 2, * ), Y( 2, * ), Z( 2, * )
```

- C binding:

```
void BLAS_xwhullv_i( int n, const <interval_array> x, int incx,
                     const <interval_array> y, int incy, <interval_array> z,
                     int incz );
```

Set Operations Involving Interval Matrices

{GE,GB,SY,SB,SP,TR,TB,TP}_ENCM_I (If an interval matrix is enclosed in another) True if $\mathbf{A} \subseteq \mathbf{B}$

This routine checks if an interval matrix $\mathbf{A}$ is enclosed in another interval matrix $\mathbf{B}$. We say that an interval matrix $\mathbf{A}$ is enclosed in another interval matrix $\mathbf{B}$, denoted as $\mathbf{A} \subseteq \mathbf{B}$, if and only if $\mathbf{a}_{i,j} \subseteq \mathbf{b}_{i,j}$ $\forall i$ and $\forall j$. Matrices $\mathbf{A}$ and $\mathbf{B}$ have the same storage format.

- Fortran 95 binding:

```
General:
      LOGICAL FUNCTION ge_encm_i( a, b )
General Band:
      LOGICAL FUNCTION gb_encm_i( a, m, kl, b )
Symmetric:
      LOGICAL FUNCTION sy_encm_i( a, b [, uplo] )
Symmetric Band:
      LOGICAL FUNCTION sb_encm_i( a, b [, uplo] )
Symmetric Packed:
      LOGICAL FUNCTION sp_encm_i( ap, bp [, uplo] )
Triangular:
      LOGICAL FUNCTION tr_encm_i( a, b [, uplo] [, diag] )
Triangular Band:
      LOGICAL FUNCTION tb_encm_i( a, b [, uplo] [, diag] )
Triangular Packed:
      LOGICAL FUNCTION tp_encm_i( ap, bp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:),  b(:,:) | bp(:)
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
      a and b have shape (m, n) for general matrix
```

```
                              (l, n) for general banded matrix (l > kl)
                              (n, n) for symmetric or triangular
                              (p+1, n) for symmetric banded or triangular
                                     banded (p = band width)
             ap and bp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      LOGICAL FUNCTION BLAS_xGE_ENCM_I( M, N, A, LDA, B, LDB)
General Band:
      LOGICAL FUNCTION BLAS_xGB_ENCM_I( M, N, KL, KU, A, LDA, B, LDB )
Symmetric:
      LOGICAL FUNCTION BLAS_xSY_ENCM_I( N, A, LDA, B, LDB )
Symmetric Band:
      LOGICAL FUNCTION BLAS_xSB_ENCM_I( N, K, A, LDA, B, LDB )
Symmetric Packed:
      LOGICAL FUNCTION BLAS_xSP_ENCM_I( N, AP, BP )
Triangular:
      LOGICAL FUNCTION BLAS_xTR_ENCM_I( N, A, LDA, B, LDB )
Triangular Band:
      LOGICAL FUNCTION BLAS_xTB_ENCM_I( N, K, A, LDA, B, LDB )
Triangular Packed:
      LOGICAL FUNCTION BLAS_xTP_ENCM_I(  N, AP, BP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, B, LDB
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDA, * )
     $                   or BP( 2, * )
```

- C binding:

```
General:
int BLAS_xge_encm_i( enum blas_order_type order, int m, int n,
                     const <interval_array> a, int lda,
                     const <interval_array> b, int ldb );
General Band:
int BLAS_xgb_encm_i( enum blas_order_type order, int m, int n, int kl,
                     int ku, const <interval_array> a, int lda,
                     const <interval_array> b, int ldb );
Symmetric:
int BLAS_xsy_encm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, const <interval_array> a, int lda,
                     const <interval_array> b, int ldb );
Symmetric Band:
int BLAS_xsb_encm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, int k, const <interval_array> a, int lda,
                     const <interval_array> b, int ldb );
```

```
Symmetric Packed:
int BLAS_xsp_encm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, const <interval_array> ap,
                     const <interval_array> bp );
Triangular:
int BLAS_xtr_encm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_diag_type diag, int n,
                     const <interval_array> a, int lda,
                     const <interval_array> b, int ldb );
Triangular Band:
int BLAS_xtb_encm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_diag_type diag, int n, int k,
                     const <interval_array> a, int lda,
                     const <interval_array> b, int ldb );
Triangular Packed:
int BLAS_xtp_encm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_diag_type diag, int n,
                     const <interval_array> ap, <interval_array> bp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_INTERIORM_I (If an interval matrix is in the interior of another interval matrix) True if $\mathbf{A} \subset \mathbf{B}$

This routine checks if an interval matrix $\mathbf{A}$ is enclosed in the interior of another interval matrix $\mathbf{B}$. We say that an interval matrix $\mathbf{A}$ is enclosed in the interior of an interval vatrix $\mathbf{B}$, if and only if $\mathbf{a}_{i,j} \subset \mathbf{b}_{i,j}$ $\forall i$ and $\forall j$. Matrices $\mathbf{A}$ and $\mathbf{B}$ have the same storage format.

- Fortran 95 binding:

```
General:
      LOGICAL FUNCTION ge_interiorm_i( a, b )
General Band:
      LOGICAL FUNCTION gb_interiorm_i( a, m, kl, b )
Symmetric:
      LOGICAL FUNCTION sy_interiorm_i( a, b [, uplo] )
Symmetric Band:
      LOGICAL FUNCTION sb_interiorm_i( a, b [, uplo] )
Symmetric Packed:
      LOGICAL FUNCTION sp_interiorm_i( ap, bp [, uplo] )
Triangular:
      LOGICAL FUNCTION tr_interiorm_i( a, b [, uplo] [, diag] )
Triangular Band:
      LOGICAL FUNCTION tb_interiorm_i( a, b [, uplo] [, diag] )
Triangular Packed:
      LOGICAL FUNCTION tp_interiorm_i( ap, bp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:), b(:,:) | bp(:)
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
```

```
        TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
      where
         a and b have shape (m, n) for general matrix
                            (l, n) for general banded matrix (l > kl)
                            (n, n) for symmetric or triangular
                            (p+1, n) for symmetric banded or triangular
                                   banded (p = band width)
         ap and bp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      LOGICAL FUNCTION BLAS_xGE_INTERIORM_I( M, N, A, LDA, B, LDB )
General Band:
      LOGICAL FUNCTION BLAS_xGB_INTERIORM_I( M, N, KL, KU, A, LDA, B, LDB )
Symmetric:
      LOGICAL FUNCTION BLAS_xSY_INTERIORM_I( N, A, LDA, B, LDB )
Symmetric Band:
      LOGICAL FUNCTION BLAS_xSB_INTERIORM_I( N, K, A, LDA, B, LDB )
Symmetric Packed:
      LOGICAL FUNCTION BLAS_xSP_INTERIORM_I( N, AP, BP )
Triangular:
      LOGICAL FUNCTION BLAS_xTR_INTERIORM_I( N, A, LDA, B, LDB )
Triangular Band:
      LOGICAL FUNCTION BLAS_xTB_INTERIORM_I( N, K, A, LDA, B, LDB )
Triangular Packed:
      LOGICAL FUNCTION BLAS_xTP_INTERIORM_I(  N, AP, BP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, B, LDB
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDA, * )
     $                   or BP( 2, * )
```

- C binding:

```
General:
int BLAS_xge_interiorm_i( enum blas_order_type order, int m, int n,
                          const <interval_array> a, int lda,
                          const <interval_array> b, int ldb );
General Band:
int BLAS_xgb_interiorm_i( enum blas_order_type order, int m, int n, int kl,
                          int ku, const <interval_array> a, int lda,
                          const <interval_array> b, int ldb );
Symmetric:
int BLAS_xsy_interiorm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          int n, const <interval_array> a, int lda,
                          const <interval_array> b, int ldb );
Symmetric Band:
```

```
int BLAS_xsb_interiorm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          int n, int k, const <interval_array> a, int lda,
                          const <interval_array> b, int ldb );
Symmetric Packed:
int BLAS_xsp_interiorm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          int n, const <interval_array> ap,
                          const <interval_array> bp );
Triangular:
int BLAS_xtr_interiorm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          enum blas_diag_type diag, int n,
                          const <interval_array> a, int lda,
                          const <interval_array> b, int ldb );
Triangular Band:
int BLAS_xtb_interiorm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          enum blas_diag_type diag, int n, int k,
                          const <interval_array> a, int lda,
                          const <interval_array> b, int ldb );
Triangular Packed:
int BLAS_xtp_interiorm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          enum blas_diag_type diag, int n,
                          const <interval_array> ap, <interval_array> bp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_DISJM_I (If two interval matrices disjoint) True if $\mathbf{A} \cap \mathbf{B} = \emptyset$

This routine checks if two interval matrices $\mathbf{A}$ and $\mathbf{B}$ disjoint, which means that if for some i, j, $\mathbf{a}_{i,j} \cap \mathbf{b}_{i,j} = \emptyset$. Matrices $\mathbf{A}$ and $\mathbf{B}$ have the same storage format.

- Fortran 95 binding:

```
General:
      LOGICAL FUNCTION ge_disjm_i( a, b )
General Band:
      LOGICAL FUNCTION gb_disjm_i( a, m, kl, b )
Symmetric:
      LOGICAL FUNCTION sy_disjm_i( a, b [, uplo] )
Symmetric Band:
      LOGICAL FUNCTION sb_disjm_i( a, b [, uplo] )
Symmetric Packed:
      LOGICAL FUNCTION sp_disjm_i( ap, bp [, uplo] )
Triangular:
      LOGICAL FUNCTION tr_disjm_i( a, b [, uplo] [, diag] )
Triangular Band:
      LOGICAL FUNCTION tb_disjm_i( a, b [, uplo] [, diag] )
Triangular Packed:
      LOGICAL FUNCTION tp_disjm_i( ap, bp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:), b(:,:) | bp(:)
```

```
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a and b have shape (m, n) for general matrix
                          (l, n) for general banded matrix (l > kl)
                          (n, n) for symmetric or triangular
                          (p+1, n) for symmetric banded or triangular
                                 banded (p = band width)
       ap and bp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      LOGICAL FUNCTION BLAS_xGE_DISJM_I( M, N, A, LDA, B, LDB )
General Band:
      LOGICAL FUNCTION BLAS_xGB_DISJM_I( M, N, KL, KU, A, LDA, B, LDB )
Symmetric:
      LOGICAL FUNCTION BLAS_xSY_DISJM_I( N, A, LDA, B, LDB )
Symmetric Band:
      LOGICAL FUNCTION BLAS_xSB_DISJM_I( N, K, A, LDA, B, LDB )
Symmetric Packed:
      LOGICAL FUNCTION BLAS_xSP_DISJM_I( N, AP, BP )
Triangular:
      LOGICAL FUNCTION BLAS_xTR_DISJM_I( N, A, LDA, B, LDB )
Triangular Band:
      LOGICAL FUNCTION BLAS_xTB_DISJM_I( N, K, A, LDA, B, LDB )
Triangular Packed:
      LOGICAL FUNCTION BLAS_xTP_DISJM_I(  N, AP, BP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, B, LDB
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDA, * )
     $                   or BP( 2, * )
```

- C binding:

```
General:
int BLAS_xge_disjm_i( enum blas_order_type order, int m, int n,
                      const <interval_array> a, int lda,
                      const <interval_array> b, int ldb );
General Band:
int BLAS_xgb_disjm_i( enum blas_order_type order, int m, int n, int kl,
                      int ku, const <interval_array> a, int lda,
                      const <interval_array> b, int ldb );
Symmetric:
int BLAS_xsy_disjm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, const <interval_array> a, int lda,
                      const <interval_array> b, int ldb );
```

```
Symmetric Band:
int BLAS_xsb_disjm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, int k, const <interval_array> a, int lda,
                      const <interval_array> b, int ldb );
Symmetric Packed:
int BLAS_xsp_disjm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, const <interval_array> ap,
                      const <interval_array> bp );
Triangular:
int BLAS_xtr_disjm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_diag_type diag, int n,
                      const <interval_array> a, int lda,
                      const <interval_array> b, int ldb );
Triangular Band:
int BLAS_xtb_disjm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_diag_type diag, int n, int k,
                      const <interval_array> a, int lda,
                      const <interval_array> b, int ldb );
Triangular Packed:
int BLAS_xtp_disjm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_diag_type diag, int n,
                      const <interval_array> ap, <interval_array> bp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_INTERM_I (Elementwise intersection of two interval matrices) $\mathbf{B} \leftarrow \mathbf{A} \cap \mathbf{B}$

This routine finds the elementwise intersection of two interval matrices **A** and **B**, and stores the result in **B**. Matrices **A** and **B** have the same storage format.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_interm_i( a, b )
General Band:
      SUBROUTINE gb_interm_i( a, m, kl, b )
Symmetric:
      SUBROUTINE sy_interm_i( a, b [, uplo] )
Symmetric Band:
      SUBROUTINE sb_interm_i( a, b [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_interm_i( ap, bp [, uplo] )
Triangular:
      SUBROUTINE tr_interm_i( a, b [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_interm_i( a, b [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_interm_i( ap, bp [, uplo], [, diag] )
```

```
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:), b(:,:) | bp(:)
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a and b have shape (m, n) for general matrix
                          (l, n) for general banded matrix (l > kl)
                          (n, n) for symmetric or triangular
                          (p+1, n) for symmetric banded or triangular
                                 banded (p = band width)
       ap and bp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_INTERM_I( M, N, A, LDA, B, LDB )
General Band:
      SUBROUTINE BLAS_xGB_INTERM_I( M, N, KL, KU, A, LDA, B, LDB )
Symmetric:
      SUBROUTINE BLAS_xSY_INTERM_I( N, A, LDA, B, LDB )
Symmetric Band:
      SUBROUTINE BLAS_xSB_INTERM_I( N, K, A, LDA, B, LDB )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_INTERM_I( N, AP, BP )
Triangular:
      SUBROUTINE BLAS_xTR_INTERM_I( N, A, LDA, B, LDB )
Triangular Band:
      SUBROUTINE BLAS_xTB_INTERM_I( N, K, A, LDA, B, LDB )
Triangular Packed:
      SUBROUTINE BLAS_xTP_INTERM_I(  N, AP, BP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, B, LDB
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDA, * )
     $                   or BP( 2, * )
```

- C binding:

```
General:
void BLAS_xge_interm_i( enum blas_order_type order, int m, int n,
                        const <interval_array> a, int lda,
                        <interval_array> b, int ldb );
General Band:
void BLAS_xgb_interm_i( enum blas_order_type order, int m, int n, int kl,
                        int ku, const <interval_array> a, int lda,
                        <interval_array> b, int ldb );
Symmetric:
void BLAS_xsy_interm_i( enum blas_order_type order, enum blas_uplo_type uplo,
```

```
                         int n, const <interval_array> a, int lda,
                         <interval_array> b, int ldb );
Symmetric Band:
void BLAS_xsb_interm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         int n, int k, const <interval_array> a, int lda,
                         <interval_array> b, int ldb );
Symmetric Packed:
void BLAS_xsp_interm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         int n, const <interval_array> ap,
                         <interval_array> bp );
Triangular:
void BLAS_xtr_interm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_diag_type diag, int n,
                         const <interval_array> a, int lda,
                         <interval_array> b, int ldb );
Triangular Band:
void BLAS_xtb_interm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_diag_type diag, int n, int k,
                         const <interval_array> a, int lda,
                         <interval_array> b, int ldb );
Triangular Packed:
void BLAS_xtp_interm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_diag_type diag, int n,
                         const <interval_array> ap, <interval_array> bp );
```

GE_WINTERM_I (Intersection of two interval matrices) $\mathbf{C} \leftarrow \mathbf{A} \cap \mathbf{B}$

This routine finds the intersection of two interval matrices $\mathbf{A}$ and $\mathbf{B}$, and stores the result in another interval matrix $\mathbf{C}$. Matrices $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ have the same storage format.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_winterm_i( a, b, c )
General Band:
      SUBROUTINE gb_winterm_i( a, m, kl, b, c )
Symmetric:
      SUBROUTINE sy_winterm_i( a, b, c [, uplo] )
Symmetric Band:
      SUBROUTINE sb_winterm_i( a, b, c [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_winterm_i( ap, bp, cp [, uplo] )
Triangular:
      SUBROUTINE tr_winterm_i( a, b, c [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_winterm_i( a, b, c [, uplo] [, diag] )
Triangular Packed:
```

```
      SUBROUTINE tp_winterm_i( ap, bp, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), b(:,:),
                                         c(:,:), ap(:), bp(:), cp(:)
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a, b and c have shape (m, n) for general matrix
                             (l, n) for general banded matrix (l > kl)
                             (n, n) for symmetric or triangular
                             (p+1, n) for symmetric banded or triangular
                                    banded (p = band width)
       ap, bp and cp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_WINTERM_I( M, N, A, LDA, B, LDB, C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_WINTERM_I( M, N, KL, KU, A, LDA, B, LDB,
     $                               C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_WINTERM_I( N, A, LDA, B, LDB, C, LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_WINTERM_I( N, K, A, LDA, B, LDB, C, LDC )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_WINTERM_I( N, AP, BP, CP )
Triangular:
      SUBROUTINE BLAS_xTR_WINTERM_I( N, A, LDA, B, LDB, C, LDC )
Triangular Band:
      SUBROUTINE BLAS_xTB_WINTERM_I( N, K, A, LDA, B, LDB, C, LDC )
Triangular Packed:
      SUBROUTINE BLAS_xTP_WINTERM_I(  N, AP, BP, CP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, B, LDB,
     $                   C, LDC
       <type>            A( 2, LDA, * ) or AP( 2, * ), B( 2, LDA, *)
     $                   or BP(2,*), C(2, LDC, *) or CP(2,*)
```

- C binding:

```
General:
void BLAS_xge_winterm_i( enum blas_order_type order, int m, int n,
                         const <interval_array> a, int lda,
                         const <interval_array> b, int ldb,
                         <interval_array> c, int ldc );
General Band:
```

```
void BLAS_xgb_winterm_i( enum blas_order_type order, int m, int n, int kl,
                         int ku, const <interval_array> a, int lda,
                         const <interval_array> b, int ldb,
                         <interval_array> c, int ldc );
Symmetric:
void BLAS_xsy_winterm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         int n, const <interval_array> a, int lda,
                         const <interval_array> b, int ldb,
                         <interval_array> c, int ldc );
Symmetric Band:
void BLAS_xsb_winterm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         int n, int k, const <interval_array> a, int lda,
                         const <interval_array> b, int ldb,
                        <interval_array> c, int ldc );
Symmetric Packed:
void BLAS_xsp_winterm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         int n, const <interval_array> ap,
                         const <interval_array> bp, <interval_array> cp );
Triangular:
void BLAS_xtr_winterm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_diag_type diag, int n,
                         const <interval_array> a, int lda,
                         const <interval_array> b, int ldb,
                         <interval_array> c, int ldc );
Triangular Band:
void BLAS_xtb_winterm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_diag_type diag, int n, int k,
                         const <interval_array> a, int lda,
                         const <interval_array> b, int ldb,
                         <interval_array> c, int ldc );
Triangular Packed:
void BLAS_xtp_winterm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_diag_type diag, int n,
                         const <interval_array> ap, <interval_array> bp,
                         <interval_array> cp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_HULLM_I (Convex hull of an interval matrix with another) $\mathbf{B} \leftarrow$ the convex hull contains $\mathbf{A} \cup \mathbf{B}$

This routine finds an interval matrix which contains both interval matrices **A** and **B**, and stores the result in **B**. Matrices **A** and **B** have the same storage format.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_hullm_i( a, b )
General Band:
```

```
      SUBROUTINE gb_hullm_i( a, m, kl, b )
Symmetric:
      SUBROUTINE sy_hullm_i( a, b [, uplo] )
Symmetric Band:
      SUBROUTINE sb_hullm_i( a, b [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_hullm_i( ap, bp [, uplo] )
Triangular:
      SUBROUTINE tr_hullm_i( a, b [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_hullm_i( a, b [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_hullm_i( ap, bp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), b(:,:), ap(:), bp(:)
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a and b have shape (m, n) for general matrix
                          (l, n) for general banded matrix (l > kl)
                          (n, n) for symmetric or triangular
                          (p+1, n) for symmetric banded or triangular
                                  banded (p = band width)
       ap and bp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_HULLM_I( M, N, A, LDA, B, LDB )
General Band:
      SUBROUTINE BLAS_xGB_HULLM_I( M, N, KL, KU, A, LDA, B, LDB )
Symmetric:
      SUBROUTINE BLAS_xSY_HULLM_I( N, A, LDA, B, LDB )
Symmetric Band:
      SUBROUTINE BLAS_xSB_HULLM_I( N, K, A, LDA, B, LDB )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_HULLM_I( N, AP, BP )
Triangular:
      SUBROUTINE BLAS_xTR_HULLM_I( N, A, LDA, B, LDB )
Triangular Band:
      SUBROUTINE BLAS_xTB_HULLM_I( N, K, A, LDA, B, LDB )
Triangular Packed:
      SUBROUTINE BLAS_xTP_HULLM_I(  N, AP, BP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, B, LDB
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDA, * )
     $                   or BP( 2, * )
```

- C binding:

```
General:
void BLAS_xge_hullm_i( enum blas_order_type order, int m, int n,
                       const <interval_array> a, int lda,
                       <interval_array> b, int ldb );
General Band:
void BLAS_xgb_hullm_i( enum blas_order_type order, int m, int n, int kl,
                       int ku, const <interval_array> a, int lda,
                       <interval_array> b, int ldb );
Symmetric:
void BLAS_xsy_hullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                       int n, const <interval_array> a, int lda,
                       <interval_array> b, int ldb );
Symmetric Band:
void BLAS_xsb_hullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                       int n, int k, const <interval_array> a, int lda,
                       <interval_array> b, int ldb );
Symmetric Packed:
void BLAS_xsp_hullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                       int n, const <interval_array> ap,
                       <interval_array> bp );
Triangular:
void BLAS_xtr_hullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                       enum blas_diag_type diag, int n,
                       const <interval_array> a, int lda,
                       <interval_array> b, int ldb );
Triangular Band:
void BLAS_xtb_hullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                       enum blas_diag_type diag, int n, int k,
                       const <interval_array> a, int lda,
                       <interval_array> b, int ldb );
Triangular Packed:
void BLAS_xtp_hullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                       enum blas_diag_type diag, int n,
                       const <interval_array> ap, <interval_array> bp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_WHULLM_I (Convex hull of two interval matrices)

$$\mathbf{C} \leftarrow \text{the convex hull contains } \mathbf{A} \cup \mathbf{B}$$

This routine finds the convex set which contains both interval matrices $\mathbf{A}$ and $\mathbf{B}$, and stores the result in an interval matrix $\mathbf{C}$. Matrices $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ have the same storage format.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_whullm_i( a, b, c )
```

```
General Band:
      SUBROUTINE gb_whullm_i( a, m, kl, b, c )
Symmetric:
      SUBROUTINE sy_whullm_i( a, b, c [, uplo] )
Symmetric Band:
      SUBROUTINE sb_whullm_i( a, b, c [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_whullm_i( ap, bp, cp [, uplo] )
Triangular:
      SUBROUTINE tr_whullm_i( a, b, c [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_whullm_i( a, b, c [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_whullm_i( ap, bp, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), b(:,:),
                                           c(:,:), ap(:), bp(:), cp(:)
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a, b and c have shape (m, n) for general matrix
                             (l, n) for general banded matrix (l > kl)
                             (n, n) for symmetric or triangular
                             (p+1, n) for symmetric banded or triangular
                                    banded (p = band width)
       ap, bp and cp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_WHULLM_I( M, N, A, LDA, B, LDB, C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_WHULLM_I( M, N, KL, KU, A, LDA, B, LDB,
     $                              C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_WHULLM_I( N, A, LDA, B, LDB, C, LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_WHULLM_I( N, K, A, LDA, B, LDB, C, LDC )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_WHULLM_I( N, AP, BP, CP )
Triangular:
      SUBROUTINE BLAS_xTR_WHULLM_I( N, A, LDA, B, LDB, C, LDC )
Triangular Band:
      SUBROUTINE BLAS_xTB_WHULLM_I( N, K, A, LDA, B, LDB, C, LDC )
Triangular Packed:
      SUBROUTINE BLAS_xTP_WHULLM_I(  N, AP, BP, CP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, B, LDB
```

```
     $                   C, LDC
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDA, *)
     $                   or BP(2,*), C(2, LDC, *) or CP(2,*)
```

- C binding:

```
General:
void BLAS_xge_whullm_i( enum blas_order_type order, int m, int n,
                        const <interval_array> a, int lda,
                        const <interval_array> b, int ldb,
                        <interval_array> c, int ldc );
General Band:
void BLAS_xgb_whullm_i( enum blas_order_type order, int m, int n, int kl,
                        int ku, const <interval_array> a, int lda,
                        const <interval_array> b, int ldb,
                        <interval_array> c, int ldc );
Symmetric:
void BLAS_xsy_whullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        int n, const <interval_array> a, int lda,
                        const <interval_array> b, int ldb,
                        <interval_array> c, int ldc );
Symmetric Band:
void BLAS_xsb_whullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        int n, int k, const <interval_array> a, int lda,
                        const <interval_array> b, int ldb,
                        <interval_array> c, int ldc );
Symmetric Packed:
void BLAS_xsp_whullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        int n, const <interval_array> ap,
                        const <interval_array> bp, <interval_array> cp );
Triangular:
void BLAS_xtr_whullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        enum blas_diag_type diag, int n,
                        const <interval_array> a, int lda,
                        const <interval_array> b, int ldb,
                        <interval_array> c, int ldc );
Triangular Band:
void BLAS_xtb_whullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        enum blas_diag_type diag, int n, int k,
                        const <interval_array> a, int lda,
                        const <interval_array> b, int ldb,
                        <interval_array> c, int ldc );
Triangular Packed:
void BLAS_xtp_whullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        enum blas_diag_type diag, int n,
                        const <interval_array> ap, <interval_array> bp,
                        <interval_array> cp );
```

Utility Functions Involving Interval Vectors

EMPTYELEV_I (Empty entry & location) $k \leftarrow \mathbf{x}_k = \emptyset$; or -1

This routine checks if an interval vector, $\mathbf{x}$, contains an empty interval entry. If $\mathbf{x}$ contains empty interval entries, then the routine returns the smallest offset or index k such that $\mathbf{x}_k =$ [NaN_empty, NaN_empty]. Otherwise, the routine returns -1.

- Fortran 95 binding:

```
    INTEGER FUNCTION emptyelev_i( x )
    TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
  where
    x has shape (n)
```

- Fortran 77 binding:

```
    INTEGER FUNCTION BLAS_xEMPTYELEV_I( N, X, INCX )
    INTEGER            N, INCX
    <type>             X( 2, * )
```

- C binding:

```
int BLAS_xemptyelev_i( int n, const <interval_array> x, int incx);
```

INFV_I (The left endpoint of an interval vector) $v \leftarrow \underline{x}$

This routine finds the real vector v such that $v_i = \underline{x}_i \ \forall i$.

- Fortran 95 binding:

```
    SUBROUTINE infv_i( x, v )
    REAL (<wp>), INTENT(OUT) :: v(:)
    TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
  where
    v and x have shape (n)
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xINFV_I( N, X, INCX, V )
    INTEGER            N, INCX
    <type>             X( 2, * ), V( * )
```

- C binding:

```
void BLAS_xinfv_i( int n, const <interval_array> x, int incx, RARRAY v );
```

SUPV_I (The right endpoint of an interval vector) $v \leftarrow \overline{x}$

This routine finds the real vector v such that $v_i = \overline{x}_i \ \forall i$.

- Fortran 95 binding:

```
    SUBROUTINE supv_i( x, v )
    REAL (<wp>), INTENT(OUT) :: v(:)
    TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
  where
    v and x have shape (n)
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xSUPV_I( N, X, INCX, V )
    INTEGER            N, INCX
    <type>             X( 2, * ), V( * )
```

- C binding:

```
void BLAS_xsupv_i( int n, const <interval_array> x, int incx, RARRAY v );
```

MIDV_I (The approximate midpoint of an interval vector) $v \leftarrow (\overline{x} + \underline{x})/2$

This routine finds the real vector v such that $v_i = \dfrac{\overline{x}_i + \underline{x}_i}{2} \ \forall i$.

- Fortran 95 binding:

```
    SUBROUTINE midv_i( x, v )
    REAL (<wp>), INTENT(OUT) :: v(:)
    TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
  where
    v and x have shape (n)
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xMIDV_I( N, X, INCX, V )
    INTEGER            N, INCX
    <type>             X( 2, * ), V( * )
```

- C binding:

```
void BLAS_xmidv_i( int n, const <interval_array> x, int incx, RARRAY v );
```

WIDTHV_I (The elementwise width of an interval vector) $v \leftarrow \overline{x} - \underline{x}$

This routine finds the real vector v such that $v_i = \overline{x}_i - \underline{x}_i \ \forall i$.

- Fortran 95 binding:

```
SUBROUTINE widthv_i( x, v )
REAL (<wp>), INTENT(OUT) :: v(:)
TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
where
  v and x have shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xWIDTHV_I( N, X, INCX, V )
INTEGER            N, INCX
<type>             X( 2, * ), V( * )
```

- C binding:

```
void BLAS_xwidthv_i( int n, const <interval_array> x, int incx, RARRAY v );
```

CONSTRUCTV_I (Constructs an interval vector from two floating point vectors)

$\mathbf{x} \leftarrow [\min\{u, v\}, \max\{u, v\}]$

This routine constructs an interval vector $\mathbf{x}$ from two floating point vectors u and v such that $\mathbf{x}_i$ contains the interval $[\min\{u_i, v_i\}, \max\{u_i, v_i\}] \ \forall i$. By letting $u = v$, the routine constructs an interval vector from a single floating point vector.

- Fortran 95 binding:

```
SUBROUTINE constructv_i( x, u, v )
REAL (<wp>), INTENT(IN) :: u(:), v(:)
TYPE(INTERVAL) (<wp>), INTENT(OUT) :: x(:)
where
  u, v and x have shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xCONSTRUCTV_I( N, U, INCU, V, INCV, X, INCX )
INTEGER            N, INCU, INCV, INCX
<type>             X( 2, * ), U( * ), V( * )
```

- C binding:

```
void BLAS_xconstructv_i( int n, RARRAY u, int incu, RARRAY v, int incv,
                         <interval_array> x, int incx );
```

Utility Functions Involving Interval Matrices

{GE,GB,SY,SB,SP,TR,TB,TP}_EMPTYELEM_I (Empty entry & location) $l \leftarrow \mathbf{a}_{l,j} = \emptyset$; or -1

This routine checks if an interval matrix, $\mathbf{A}$, contains an empty interval entry. If $\mathbf{A}$ contains empty interval entries, then the routine returns the smallest offset or index l (according to the first index) such that $\mathbf{a}_{l,j} = [\text{NaN\_empty}, \text{NaN\_empty}]$. Otherwise, it returns -1.

- Fortran 95 binding:

```
General:
      INTEGER FUNCTION  ge_emptyelem_i( a )
General Band:
      INTEGER FUNCTION  gb_emptyelem_i( a, m, kl )
Symmetric:
      INTEGER FUNCTION  sy_emptyelem_i( a [, uplo] )
Symmetric Band:
      INTEGER FUNCTION  sb_emptyelem_i( a, kl [, uplo] )
Symmetric Packed:
      INTEGER FUNCTION  sp_emptyelem_i( ap [, uplo] )
Triangular:
      INTEGER FUNCTION  tr_emptyelem_i( a [, uplo] [, diag] )
Triangular Band:
      INTEGER FUNCTION  tb_emptyelem_i( a, kl [, uplo] [, diag] )
Triangular Packed:
      INTEGER FUNCTION  tp_emptyelem_i( ap, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), ap(:)
      INTEGER, INTENT(OUT) :: i, j
      INTEGER, INTENT(IN) :: kl
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a has shape (m, n) for general matrix
                   (l, n) for general banded matrix (l > kl)
                   (n, n) for symmetric or triangular
                   (p+1, n) for symmetric banded or triangular
                            banded (p = band width)
       ap has shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      INTEGER FUNCTION  BLAS_xGE_EMPTYELEM_I( M, N, A, LDA, B, LDB )
General Band:
      INTEGER FUNCTION  BLAS_xGB_EMPTYELEM_I( M, N, KL, KU, A, LDA )
Symmetric:
      INTEGER FUNCTION  BLAS_xSY_EMPTYELEM_I( UPLO, N, A, LDA )
```

```
Symmetric Band:
      INTEGER FUNCTION  BLAS_xSB_EMPTYELEM_I( UPLO, N, K, A, LDA )
Symmetric Packed:
      INTEGER FUNCTION  BLAS_xSP_EMPTYELEM_I( UPLO, N, AP )
Triangular:
      INTEGER FUNCTION  BLAS_xTR_EMPTYELEM_I( UPLO, TRANS, DIAG, N, A, LDA )
Triangular Band:
      INTEGER FUNCTION  BLAS_xTB_EMPTYELEM_I( UPLO, TRANS, DIAG, N, K, A, LDA )
Triangular Packed:
      INTEGER FUNCTION  BLAS_xTP_EMPTYELEM_I( UPLO, TRANS, DIAG, N, AP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, I, J
      <type>             A( 2, LDA, * ) or AP( 2, * )
```

- C binding:

```
General:
int BLAS_xge_emptyelem_i( enum blas_order_type order, int m, int n,
                          const <interval_array> a, int lda );
General Band:
int BLAS_xgb_emptyelem_i( enum blas_order_type order, int m, int n, int kl,
                          int ku, const <interval_array> a, int lda, int i,
                          int j);
Symmetric:
int BLAS_xsy_emptyelem_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          int n, const <interval_array> a, int lda );
Symmetric Band:
int BLAS_xsb_emptyelem_i( enum blas_order_type order, int n, int k,
                          const <interval_array> a, int lda );
Symmetric Packed:
int BLAS_xsp_emptyelem_i( enum blas_order_type order, int n,
                          const <interval_array> ap );
Triangular:
int BLAS_xtr_emptyelem_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          enum blas_trans_type trans, enum blas_diag_type diag,
                          int n, const <interval_array> a, int lda );
Triangular Band:
int BLAS_xtb_emptyelem_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          enum blas_trans_type trans, enum blas_diag_type diag,
                          int n, int k, const <interval_array> a, int lda );
Triangular Packed:
int BLAS_xtp_emptyelem_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          enum blas_trans_type trans, enum blas_diag_type diag,
                          int n, const <interval_array> ap, int i, int j);
```

{GE,GB,SY,SB,SP,TR,TB,TP}_INFM_I (Left endpoint of an interval matrix) $C \leftarrow \underline{A}$

This routine finds the real matrix C such that $c_{i,j} = \underline{a}_{i,j}$ $\forall i$ and $\forall j$, where $\mathbf{A} = \{\mathbf{a}_{i,j}\}$ is a general (or general banded, or symmetric, or symmetric banded, symmetric packed, or triangular, triangular banded, triangular packed) interval matrix.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_infm_i( a, c )
General Band:
      SUBROUTINE gb_infm_i( a, m, kl, c )
Symmetric:
      SUBROUTINE sy_infm_i( a, c [, uplo] )
Symmetric Band:
      SUBROUTINE sb_infm_i( a, kl, c [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_infm_i( ap, cp [, uplo] )
Triangular:
      SUBROUTINE tr_infm_i( a, c [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_infm_i( a, kl, c [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_infm_i( ap, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), ap(:)
      REAL (<wp>), INTENT(OUT) :: c(:,:), cp(:)
      INTEGER, INTENT(IN) :: kl
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a and c have shape
           (m, n) for general matrix
           (l, n) for general banded matrix (l > kl)
           (n, n) for symmetric or triangular
           (p+1, n) for symmetric banded or triangular
                    banded (p = band width)
       ap and cp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_INFM_I( M, N, A, LDA C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_INFM_I( M, N, KL, KU, A, LDA, C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_INFM_I( UPLO, N, A, LDA, C, LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_INFM_I( UPLO, N, K, A, LDA, C, LDC )
Symmetric Packed:
```

```
          SUBROUTINE BLAS_xSP_INFM_I( UPLO, N, AP, CP )
     Triangular:
          SUBROUTINE BLAS_xTR_INFM_I( UPLO, TRANS, DIAG, N, A, LDA, C, LDC )
     Triangular Band:
          SUBROUTINE BLAS_xTB_INFM_I( UPLO, TRANS, DIAG, N, K, A, LDA, C, LDC )
     Triangular Packed:
          SUBROUTINE BLAS_xTP_INFM_I( UPLO, TRANS, DIAG, N, AP, CP )
     all:
          INTEGER           UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, LDC
          <type>            A( 2, LDA, * ) or AP( 2, * ), C( LDC, * ) or CP( * )
```

- C binding:

```
General:
void BLAS_xge_infm_i( enum blas_order_type order, int m, int n,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
General Band:
void BLAS_xgb_infm_i( enum blas_order_type order, int m, int n, int kl, int ku,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
Symmetric:
void BLAS_xsy_infm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Symmetric Band:
void BLAS_xsb_infm_i( enum blas_order_type order, int n, int k,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
Symmetric Packed:
void BLAS_xsp_infm_i( enum blas_order_type order, int n,
                      const <interval_array> ap, RARRAY cp );
Triangular:
void BLAS_xtr_infm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Triangular Band:
void BLAS_xtb_infm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, int k, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Triangular Packed:
void BLAS_xtp_infm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> ap, RARRAY cp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_SUPM_I (Right endpoint of an interval matrix) $C \leftarrow \overline{A}$

This routine finds the real matrix C such that $c_{i,j} = \overline{a}_{i,j}\ \forall i$ and $\forall j$, where $\mathbf{A} = \{\mathbf{a}_{i,j}\}$ is a general (or general banded, or symmetric, or symmetric banded, symmetric packed, or triangular, triangular banded, triangular packed) interval matrix.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_supm_i( a, c )
General Band:
      SUBROUTINE gb_supm_i( a, m, kl, c )
Symmetric:
      SUBROUTINE sy_supm_i( a, c [, uplo] )
Symmetric Band:
      SUBROUTINE sb_supm_i( a, kl, c [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_supm_i( ap, cp [, uplo] )
Triangular:
      SUBROUTINE tr_supm_i( a, c [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_supm_i( a, kl, c [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_supm_i( ap, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), ap(:)
      REAL (<wp>), INTENT(OUT) :: c(:,:), cp(:)
      INTEGER, INTENT(IN) :: kl
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a and c have shape
           (m, n) for general matrix
           (l, n) for general banded matrix (l > kl)
           (n, n) for symmetric or triangular
           (p+1, n) for symmetric banded or triangular
                    banded (p = band width)
       ap and cp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_SUPM_I( M, N, A, LDA, C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_SUPM_I( M, N, KL, KU, A, LDA, C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_SUPM_I( UPLO, N, A, LDA, C, LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_SUPM_I( UPLO, N, K, A, LDA,  C, LDC )
Symmetric Packed:
```

```
        SUBROUTINE BLAS_xSP_SUPM_I( UPLO, N, AP, CP )
  Triangular:
        SUBROUTINE BLAS_xTR_SUPM_I( UPLO, TRANS, DIAG, N, A, LDA, C, LDC )
  Triangular Band:
        SUBROUTINE BLAS_xTB_SUPM_I( UPLO, TRANS, DIAG, N, K, A, LDA, C, LDC )
  Triangular Packed:
        SUBROUTINE BLAS_xTP_SUPM_I( UPLO, TRANS, DIAG, N, AP, CP )
  all:
        INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, LDC
        <type>             A( 2, LDA, * ) or AP( 2, * ), C( LDC, * ) or CP( * )
```

- C binding:

```
General:
void BLAS_xge_supm_i( enum blas_order_type order, int m, int n,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
General Band:
void BLAS_xgb_supm_i( enum blas_order_type order, int m, int n, int kl, int ku,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
Symmetric:
void BLAS_xsy_supm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Symmetric Band:
void BLAS_xsb_supm_i( enum blas_order_type order, int n, int k,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
Symmetric Packed:
void BLAS_xsp_supm_i( enum blas_order_type order, int n,
                      const <interval_array> ap, RARRAY cp );
Triangular:
void BLAS_xtr_supm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Triangular Band:
void BLAS_xtb_supm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, int k, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Triangular Packed:
void BLAS_xtp_supm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> ap, RARRAY cp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_MIDM_I (Midpoint matrix of an interval matrix) $C \leftarrow (\mathbf{A}+\mathbf{B})/2$

This routine finds the real matrix C such that $BLAS_{i,j} = \dfrac{\underline{a}_{i,j} + \overline{a}_{i,j}}{2}$ $\forall i$ and $\forall j$, where $\mathbf{A} = \{\mathbf{a}_{i,j}\}$ is a general (or general banded, or symmetric, or symmetric banded, symmetric packed, or triangular, triangular banded, triangular packed) interval matrix.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_midm_i( a, c)
General Band:
      SUBROUTINE gb_midm_i( a, m, kl, c )
Symmetric:
      SUBROUTINE sy_midm_i( a, c [, uplo] )
Symmetric Band:
      SUBROUTINE sb_midm_i( a, kl, c [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_midm_i( ap, cp [, uplo] )
Triangular:
      SUBROUTINE tr_midm_i( a, c [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_midm_i( a, kl, c [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_midm_i( ap, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), ap(:)
      REAL (<wp>), INTENT(OUT) :: c(:,:), cp(:)
      INTEGER, INTENT(IN) :: kl
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a and c have shape
           (m, n) for general matrix
           (l, n) for general banded matrix (l > kl)
           (n, n) for symmetric or triangular
           (p+1, n) for symmetric banded or triangular
                    banded (p = band width)
       ap and cp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_MIDM_I( M, N, A, LDA, C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_MIDM_I( M, N, KL, KU, A, LDA, C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_MIDM_I( UPLO, N, A, LDA, C, LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_MIDM_I( UPLO, N, K, A, LDA, C, LDC )
```

```
Symmetric Packed:
      SUBROUTINE BLAS_xSP_MIDM_I( UPLO, N, AP, CP )
Triangular:
      SUBROUTINE BLAS_xTR_MIDM_I( UPLO, TRANS, DIAG, N, A, LDA, C, LDC )
Triangular Band:
      SUBROUTINE BLAS_xTB_MIDM_I( UPLO, TRANS, DIAG, N, K, A, LDA, C, LDC )
Triangular Packed:
      SUBROUTINE BLAS_xTP_MIDM_I( UPLO, TRANS, DIAG, N, AP, CP )
all:
      INTEGER             UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, LDC
      <type>              A( 2, LDA, * ) or AP( 2, * ), C( 2, LDA, * ) or
     $                    CP( 2, * )
```

- C binding:

```
General:
void BLAS_xge_midm_i( enum blas_order_type order, int m, int n,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
General Band:
void BLAS_xgb_midm_i( enum blas_order_type order, int m, int n, int kl, int ku,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
Symmetric:
void BLAS_xsy_midm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, const <interval_array> a, int lda, RARRAY c,
                      int ldc );
Symmetric Band:
void BLAS_xsb_midm_i( enum blas_order_type order, int n, int k,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
Symmetric Packed:
void BLAS_xsp_midm_i( enum blas_order_type order, int n,
                      const <interval_array> ap, RARRAY cp );
Triangular:
void BLAS_xtr_midm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Triangular Band:
void BLAS_xtb_midm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, int k, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Triangular Packed:
void BLAS_xtp_midm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> ap, RARRAY cp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_WIDTHM_I (Elementwise width of an interval matrix) $C \leftarrow \overline{A} - \underline{A}$

This routine finds the real matrix C such that $c_{i,j} = \overline{a}_{i,j} - \underline{a}_{i,j}$ $\forall i$ and $\forall j$, where $\mathbf{A} = \{\mathbf{a}_{i,j}\}$ is a general (or general banded, or symmetric, or symmetric banded, symmetric packed, or triangular, triangular banded, triangular packed) interval matrix.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_widthm_i( a, c )
General Band:
      SUBROUTINE gb_widthm_i( a, m, kl, c )
Symmetric:
      SUBROUTINE sy_widthm_i( a, c [, uplo] )
Symmetric Band:
      SUBROUTINE sb_widthm_i( a, kl, c [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_widthm_i( ap, cp [, uplo] )
Triangular:
      SUBROUTINE tr_widthm_i( a, c [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_widthm_i( a, kl, c [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_widthm_i( ap, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), ap(:)
      REAL (<wp>), INTENT(OUT) :: c(:,:), cp(:)
      INTEGER, INTENT(IN) :: kl
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a and c have shape
           (m, n) for general matrix
           (l, n) for general banded matrix (l > kl)
           (n, n) for symmetric or triangular
           (p+1, n) for symmetric banded or triangular
                    banded (p = band width)
       ap and cp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_WIDTHM_I( M, N, A, LDA, C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_WIDTHM_I( M, N, KL, KU, A, LDA, C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_WIDTHM_I( UPLO, N, A, LDA, C, LDC )
Symmetric Band:
```

```
            SUBROUTINE BLAS_xSB_WIDTHM_I( UPLO, N, K, A, LDA, C, LDC )
      Symmetric Packed:
            SUBROUTINE BLAS_xSP_WIDTHM_I( UPLO, N, AP, CP )
      Triangular:
            SUBROUTINE BLAS_xTR_WIDTHM_I( UPLO, TRANS, DIAG, N, A, LDA, C, LDC )
      Triangular Band:
            SUBROUTINE BLAS_xTB_WIDTHM_I( UPLO, TRANS, DIAG, N, K, A, LDA, C, LDC )
      Triangular Packed:
            SUBROUTINE BLAS_xTP_WIDTHM_I( UPLO, TRANS, DIAG, N, AP, CP )
      all:
            INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, LDC
            <type>             A( 2, LDA, * ) or AP( 2, * ), C( LDC, * ) or
           $                   CP( * )
```

- C binding:

```
General:
void BLAS_xge_widthm_i( enum blas_order_type order, int m, int n,
                        const <interval_array> a, int lda, RARRAY c,
                        int ldc );
General Band:
void BLAS_xgb_widthm_i( enum blas_order_type order, int m, int n, int kl,
                        int ku, const <interval_array> a, int lda, RARRAY c,
                        int ldc );
Symmetric:
void BLAS_xsy_widthm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        int n, const <interval_array> a, int lda,
                        RARRAY c, int ldc );
Symmetric Band:
void BLAS_xsb_widthm_i( enum blas_order_type order, int n, int k,
                        const <interval_array> a, int lda, RARRAY c, int ldc );
Symmetric Packed:
void BLAS_xsp_widthm_i( enum blas_order_type order, int n,
                        const <interval_array> ap, RARRAY cp );
Triangular:
void BLAS_xtr_widthm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        enum blas_trans_type trans, enum blas_diag_type diag,
                        int n, const <interval_array> a, int lda,
                        RARRAY c, int ldc );
Triangular Band:
void BLAS_xtb_widthm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        enum blas_trans_type trans, enum blas_diag_type diag,
                        int n, int k, const <interval_array> a, int lda,
                        RARRAY c, int ldc );
Triangular Packed:
void BLAS_xtp_widthm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        enum blas_trans_type trans, enum blas_diag_type diag,
                        int n, const <interval_array> ap, RARRAY cp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_CONSTRUCTM_I (Constructs an interval matrix from two floating point matrices) $\mathbf{A} \supseteq B, C$

This routine constructs an interval matrix from two floating point matrices B and C such that $\mathbf{a}_{i,j} = [\min\{b_{i,j}, BLAS_{i,j}\}, \max\{b_{i,j}, BLAS_{i,j}\}]$ $\forall i \in \{0, 1, \cdots, m-1\}$ and $\forall j \in \{0, 1, \cdots, n-1\}$. Both floating point matrices B and C have the same storage format.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_constructm_i( a, b, c )
General Band:
      SUBROUTINE gb_constructm_i( a, b, m, kl, c )
Symmetric:
      SUBROUTINE sy_constructm_i( a, b, c [, uplo] )
Symmetric Band:
      SUBROUTINE sb_constructm_i( a, b, kl, c [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_constructm_i( ap, bp, cp [, uplo] )
Triangular:
      SUBROUTINE tr_constructm_i( a, b, c [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_constructm_i( a, b, kl, c [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_constructm_i( ap, bp, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(OUT) :: a(:,:), ap(:)
      REAL (<wp>), INTENT(IN) :: b(:,:), c(:,:), cp(:)
      INTEGER, INTENT(IN) :: kl
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a, b and c have shape
           (m, n) for general matrix
           (l, n) for general banded matrix (l > kl)
           (n, n) for symmetric or triangular
           (p+1, n) for symmetric banded or triangular
                    banded (p = band width)
       ap and cp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_CONSTRUCTM_I( M, N, A, LDA, B, LDB, C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_CONSTRUCTM_I( M, N, KL, KU, A, LDA, B, LDB, C, LDC )
```

```
Symmetric:
      SUBROUTINE BLAS_xSY_CONSTRUCTM_I( UPLO, N, A, LDA, B, LDB, C, LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_CONSTRUCTM_I( UPLO, N, K, A, LDA, B. LDB, C, LDC )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_CONSTRUCTM_I( UPLO, N, AP, BP, CP )
Triangular:
      SUBROUTINE BLAS_xTR_CONSTRUCTM_I( UPLO, TRANS, DIAG, N, A, LDA,
     $                                  B, LDB, C, LDC )
Triangular Band:
      SUBROUTINE BLAS_xTB_CONSTRUCTM_I( UPLO, TRANS, DIAG, N, K, A, LDA,
     $                                  B, LDB, C, LDC )
Triangular Packed:
      SUBROUTINE BLAS_xTP_CONSTRUCTM_I( UPLO, TRANS, DIAG, N, AP, BP, CP )
all:
      INTEGER           UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, LDC
      <type>            A( 2, LDA, * ) or AP( 2, * ),  B( LDC, * )
     $                  or BP( * ), C( LDC, * ) or CP( * ),
```

- C binding:

```
General:
void BLAS_xge_constructm_i( enum blas_order_type order, int m, int n,
                            <interval_array> a, int lda, RARRAY b, int ldb
                            RARRAY c, int ldc );
General Band:
void BLAS_xgb_constructm_i( enum blas_order_type order, int m, int n, int kl,
                            int ku, <interval_array> a, int lda, RARRAY c,
                            int ldc );
Symmetric:
void BLAS_xsy_constructm_i( enum blas_order_type order,
                            enum blas_uplo_type uplo, int n,
                            <interval_array> a, int lda, RARRAY b,
                            int ldb, RARRAY c, int ldc );
Symmetric Band:
void BLAS_xsb_constructm_i( enum blas_order_type order, int n, int k,
                            <interval_array> a, int lda, RARRAY b, int ldb,
                            RARRAY c, int ldc );
Symmetric Packed:
void BLAS_xsp_constructm_i( enum blas_order_type order, int n,
                            <interval_array> ap, RARRAY bp, RARRAY cp );
Triangular:
void BLAS_xtr_constructm_i( enum blas_order_type order,
                            enum blas_uplo_type uplo,
                            enum blas_trans_type trans,
                            enum blas_diag_type diag, int n,
                            <interval_array> a, int lda, RARRAY b,
                            int ldb, RARRAY c, int ldc );
```

```
Triangular Band:
void BLAS_xtb_constructm_i( enum blas_order_type order,
                            enum blas_uplo_type uplo,
                            enum blas_trans_type trans,
                            enum blas_diag_type diag, int n, int k,
                            <interval_array> a, int lda, RARRAY b,
                            int ldb, RARRAY c, int ldc );
Triangular Packed:
void BLAS_xtp_constructm_i( enum blas_order_type order,
                            enum blas_uplo_type uplo,
                            enum blas_trans_type trans,
                            enum blas_diag_type diag, int n,
                            <interval_array> ap, RARRAY bp, RARRAY cp );
```

Environmental Enquiry

FPINFO_I (Environmental enquiry)

This routine queries for machine-specific floating point characteristics. Refer to section 1.6 for a list of all possible return values of this routine, and sections A.4, A.5, and A.6, for their respective language dependent representations in Fortran 95, Fortran 77, and C.

- Fortran 95 binding:

```
REAL(<wp>) FUNCTION fpinfo_i( cmach, prec )
  TYPE (blas_cmach_type), INTENT(IN) :: cmach
  <type>(<wp>), INTENT(IN) :: prec
```

- Fortran 77 binding:

```
<rtype> FUNCTION BLAS_xFPINFO_I( CMACH )
INTEGER           CMACH
```

- C binding:

```
<rtype> BLAS_xfpinfo_i( enum blas_cmach_type cmach );
```

Bibliography

[1] Global solutions. `http://www.mscs.mu.edu/~globsol/`.

[2] Interval computations. `http://cs.utep.edu/interval-comp/icompwww.html`.

[3] M. Aboelaze, N. Chrisochoides, and E. Houstis. The Parallelization of Level 2 and 3 BLAS Operations on Distributed Memory Machines. Technical Report CSD-TR-91-007, Purdue University, West Lafayette, IN, 1991.

[4] R. Agarwal, S. Balle, F. Gustavson, M. Joshi, and P. Palkar. A Three-Dimensional Approach to Parallel Matrix Multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[5] R. Agarwal, F. Gustavson, and M. Zubair. A High Performance Matrix Multiplication Algorithm on a Distributed-Memory Parallel Computer, Using Overlapped Communication. *IBM Journal of Research and Development*, 38(6):673–681, 1994.

[6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, third edition, 1999. (Also available in Japanese, published by Maruzen, Tokyo, translated by Dr Oguni).

[7] ANSI/IEEE, New York. *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, 1985.

[8] P. Bangalore. The Data-Distribution-Independent Approach to Scalable Parallel Libraries. Master's thesis, Mississippi State University, 1995.

[9] D. Bindel, J. Demmel, W. Kahan, and O. Marques. On computing givens rotations reliably and efficiently. LAPACK Working Note No.148. Technical Report CS-00-449, Department of Computer Science, University of Tennessee, 1122 Volunteer Boulevard, Knoxville, TN 37996-3450, USA, 2000. URL: http://www.netlib.org/lapack/lawns/.

[10] R. Bisseling and J. van der Vorst. Parallel Triangular System Solving on a mesh network of Transputers. *SIAM Journal on Scientific and Statistical Computing*, 12:787–799, 1991.

[11] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.

[12] G. Bohlender. Literature list on enclosure methods and related topics. `http://ma70.rz.uni-karlsruhe.de/~ae15/litlist.html`.

[13] R. Brent and P. Strazdins. Implementation of BLAS Level 3 and LINPACK Benchmark on the AP1000. *Fujitsu Scientific and Technical Journal*, 5(1):61–70, 1993.

[14] D. Chiriaev and G. W. Walster. Interval arithmetic specification. `http://www.mscs.mu.edu/~globsol/walster-papers.html`.

[15] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. In J. Dongarra, K. Masden, and J. Waśniewski, editors, *Applied Parallel Computing*, pages 107–114. Springer Verlag, 1995. (also LAPACK Working Note No.100).

[16] J. Choi, J. Dongarra, and D. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994. (also LAPACK Working Note No.57).

[17] J. Choi, J. Dongarra, and D. Walker. PB-BLAS: A Set of Parallel Block Basic Linear Algebra Subroutines. *Concurrency: Practice and Experience*, 8(7):517–535, 1996.

[18] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, and R. van de Geijn. Parallel Implementation of BLAS: General Techniques for Level 3 BLAS. Technical Report TR95-49, Department of Computer Sciences, UT-Austin, 1995. Submitted to Concurrency: Practice and Experience.

[19] J. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[20] D. S. Dodson. Corrigendum: Remark on "Algorithm 539: Basic Linear Algebra Subroutines for FORTRAN usage". *ACM Trans. Math. Software*, 9:140, 1983. (See also [39] and [21]).

[21] D. S. Dodson and R. G. Grimes. Remark on algorithm 539: Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Software*, 8:403–404, 1982. (See also [39] and [20]).

[22] D. S. Dodson, R. G. Grimes, and J. G. Lewis. Sparse extensions to the FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 17:253–272, 1991. (Algorithm 692).

[23] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1979.

[24] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A proposal for a set of Level 3 Basic Linear Algebra Subprograms. In G. Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 40–44. SIAM, Philadelphia, PA, USA, 1989. (Proceedings of the Third SIAM Conference).

[25] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–28, 1990. (Algorithm 679).

[26] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 14:1–32, 399, 1988. (Algorithm 656).

[27] I. S. Duff, M. Marrone, G. Radicati, and C. Vittoli. Level 3 Basic Linear Algebra Subprograms for sparse matrics: A user-level interface. *ACM Trans. Math. Software*, 23:379–401, 1997.

[28] A. Elster. Basic Matrix Subprograms for Distributed Memory Systems. In D. Walker and Q. Stout, editors, *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 311–316. IEEE Press, 1990.

[29] R. Falgout, A. Skjellum, S. Smith, and C. Still. The Multicomputer Toolbox Approach to Concurrent BLAS and LACS. In *Proceedings of the Scalable High Performance Computing Conference SHPCC-92*. IEEE Computer Society Press, 1992.

[30] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3–4), 1994.

[31] G. Fox, S. Otto, and A. Hey. Matrix Algorithms on a Hypercube I: Matrix Multiplication. *Parallel Computing*, 3:17–31, 1987.

[32] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide Extension*, volume 51 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1977.

[33] G. Golub and C. van Loan. *Matrix Computations*. Johns-Hopkins, Baltimore, third edition, 1996.

[34] M. Heath and C. Romine. Parallel Solution Triangular Systems on Distributed Memory Multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 9:558–588, 1988.

[35] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, 1996.

[36] S. Huss-Lederman, E. Jacobson, A. Tsao, and G. Zhang. Matrix Multiplication on the Intel Touchstone DELTA. *Concurrency: Practice and Experience*, 6(7):571–594, 1994.

[37] IEEE. *ANSI/IEEE Standard for Binary Floating Point Arithmetic: Std 754-1985*. IEEE Press, New York, NY, USA, 1985.

[38] ISO/IEC. *C9X FCD Standard (Draft)*, x3j11/98-049, wg14/n843 edition, 1998. http://http://wwwold.dkuug.dk/JTC1/SC22/WG14/.

[39] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Trans. Math. Software*, 5:308–323, 1979. (Algorithm 539. See also [21] and [20]).

[40] G. Li and T. Coleman. A Parallel Triangular Solver for a Distributed-Memory Multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9(3):485–502, 1988.

[41] G. Li and T. Coleman. A New Method for Solving Triangular Systems on Distributed-Memory Message-Passing Multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 10(2):382–396, 1989.

[42] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, A. Kapur, M. Martin, T. Tung, and D. Yoo. Design, implementation and testing of extended and mixed precision blas. Technical Report CS-00-451, Department of Computer Science, University of Tennessee, 1122 Volunteer Boulevard, Knoxville, TN 37996-3450, USA, 2000. URL: http://www.netlib.org/lapack/lawns/.

[43] W. Lichtenstein and S. L. Johnsson. Block-Cyclic Dense Linear Algebra. *SIAM Journal on Scientific and Statistical Computing*, 14(6):1259–1288, 1993.

[44] K. Mathur and S. L. Johnsson. Multiplication of Matrices of Arbitrary Shapes on a Data Parallel Computer. *Parallel Computing*, 20:919–951, 1994.

[45] O. McBryan and E. van de Velde. Matrix and Vector Operations on Hypercube Parallel Processors. *Parallel Computing*, 5:117–126, 1987.

[46] D. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–145, Grenoble, France, June 26-28 1991. IEEE Computer Society Press.

[47] Jonathan Richard Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. Technical Report CMU-CS-96-140, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996. to appear in Discrete & Computational Geometry.

[48] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1976.

[49] P. Strazdins. A High Performance, Portable Distributed BLAS Implementation. In *Proceedings of the Sixth Parallel Computing Workshop*, Fujitsu Parallel Computing Center, 1996.

[50] Thinking Machines Corporation. *CMSSL for Fortran*, 1990.

[51] R. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, Cambridge, Massachusetts, 1997.

[52] R. van de Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. Technical Report UT CS-95-286, LAPACK Working Note No.96, University of Tennessee, 1995.

Index