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	$A \leftarrow \alpha xy^T + \beta A$	GE	R
and symmetric ($A = A^T$)	$A \leftarrow \alpha xx^T + \beta A$	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 $\max(0,j-ku) \leq i \leq \min(m-1,j+kl)$. For row-major storage, a_{ij} is stored in $\mathrm{AB}(i,kl+j-i)$ for $\max(0,j-ku) \leq i \leq \min(n-1,j+kl)$. 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 $\max(0,j-kd) \leq i \leq j$;
- if `uplo = blas_lower`, a_{ij} is stored in $\mathrm{AB}(i-j,j)$ for $j \leq i \leq \min(n-1,j+kd)$.

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, $1 \le \tau \le 2$, except that $\tau = 0$ implies $H = I$.

In complex arithmetic, τ may be complex, and satisfies $1 \le \mathrm{Re}(\tau) \le 2$ and $|\tau - 1| \le 1$. 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 $\mathsf{n} - 1$
  $\mathsf{x(i))} \leftrightarrow \mathsf{x(\ p(i))\ ))}$
End do
	or	Do $\mathsf{i} = \mathsf{n} - 1$ to 0
		  $\mathsf{x(i))} \leftrightarrow \mathsf{x(\ p(i))\ ))}$
		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 $\arg\min_{0 \le i < \mathtt{n}} x_i$ 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 $k = \arg\max_{0 \le i < n} (|Re(x_i)| + |Im(x_i)|)$

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 $1.0 \leq ssq \leq (sumsq + n)$ when x is a real vector and $1.0 \leq ssq \leq (sumsq + 2n)$ when x is a complex vector. *scale* is assumed to be non-negative and *scl* returns the value

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

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) $x \leftarrow \mathrm{sort}(x)$

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) $(p, x) \leftarrow \mathrm{sort}(x)$

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) $A \in I\!R^{n^2}, A \leftarrow \alpha x y^T + \beta A$ $A \in \mathbb{C}^{n^2}, A \leftarrow \alpha x y^T + \beta A$ 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
                  (1,n) for general banded matrix (1 > 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) $B \leftarrow \alpha op(T)B$ 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 $B \leftarrow \alpha B op(T^{-1})$

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.

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	$B \leftarrow \alpha BT^{-1}$
2	T	R	$B \leftarrow \alpha BT^{-T}$
2	H	R	$B \leftarrow \alpha BT^{-H}$
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 );
```