C.4 Interval BLAS

C.4.1 Introduction

Interval computation refers to performing computations with intervals. Computing with intervals guarantees that interval results contain the set of all possible correct answers. Valid implementations of interval arithmetic produce correct bounds on the set of all possible correct answers, including the effìcts of accumulated roundoff errors. Recent advances in interval algorithms have generated interest in using these methods in increasing numbers of applications. This motivates us to establish the standard for interval BLAS described in this chapter.

Intervals

A nonempty *mathematical interval* $[a, b]$ is the set $\{x \in \Re | a \leq x \leq b\}$ where $a \leq b$. A *machine interval* $[a^*, b^*]$ is a mathematical interval whose endpoints are machine representable numbers. We say that $[a^*, b^*]$ is a machine representation of $[a, b]$ if $[a^*, b^*]$ *contains* $[a, b]$ i.e. $a^* \leq a$ and $b \leq b^*$. We say that the machine interval $[a^*, b^*]$ is a *tight representation* of a mathematical interval $[a, b]$ if and only if a^* is the greatest machine representable number which is less than or equal to a, and b^* is the least machine representable number which is greater than or equal to b.
The *empty interval* $\emptyset$, which does not contain any real number, is required in the interval BLAS. For machines in compliance with the IEEE-standard, we recommend the use of [NaN_empty, NaN_empty] to represent the empty interval, where NaN_empty is a unique non-default quiet not-a-number that is used to represent the empty interval *only.*
interval.

Notation

Both scalar (floating point number) and interval arguments are used for the specification of routines in this chapter. *Interval vectors* and *interval matrices* are vectors and matrices whose entries are intervals. The notation used in this chapter is consistent with other chapters, but we use **boldface letters** to specify interval arguments. We also use overline and underline to specify the greatest lower bound and the least upper bound of an interval variable, respectively. For example, if $\mathbf{x}$ is an interval vector, then $\mathbf{x} = [\underline{x}, \overline{x}]$.

Interval arithmetic

Interval arithmetic on mathematical intervals is defined as follows.

> *Let* $\mathbf{a}$ *and* $\mathbf{b}$ *be two mathematical intervals. Let* op *be one of the arithmetic operations* $+, -, \times, \div$. *Then* $\mathbf{a}$ op $\mathbf{b} \equiv \{a \text{ op } b : a \in \mathbf{a}, b \in \mathbf{b}\}$, *provided that* $0 \notin \mathbf{b}$ *if* op *represents* $\div$.

> *Advice to users:* The above definition of division implies that the user is responsible to trapping and dealing with any division by an interval containing zero.

Table C.1 gives explicit implementations of these four basic interval arithmetic operations and other operations on mathematical intervals used in this chapter. We use the notation $\mathbf{a} = [\underline{a}, \overline{a}]$ and $\mathbf{b} = [\underline{b}, \overline{b}]$.
All operations in the interval BLAS are necessarily performed on machine intervals. Arithmetic on machine intervals must satisfy the following property:

Operation	$\mathbf{a} \neq \emptyset$ and $\mathbf{b} \neq \emptyset$	$\mathbf{a} = \emptyset$ or/and $\mathbf{b} = \emptyset$
Addition $\mathbf{a} + \mathbf{b}$	$[\underline{a} + \underline{b}, \overline{a} + \overline{b}]$	$\emptyset$
Subtraction $\mathbf{a} - \mathbf{b}$	$[\underline{a} - \overline{b}, \overline{a} - \underline{b}]$	$\emptyset$
Multiplication $\mathbf{a} * \mathbf{b}$	$[\min\{\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}\}, \max\{\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}\}]$	$\emptyset$
Cancellation $\mathbf{a} \ominus \mathbf{b}$	$[\underline{a} - \underline{b}, \overline{a} - \overline{b}]$ if $(\underline{a} - \underline{b}) \leq (\overline{a} - \overline{b})$; Otherwise, $\emptyset$	$\emptyset$
Division $\dfrac{\mathbf{a}}{\mathbf{b}}, (0 \notin \mathbf{b})$	$[\min\{\frac{\underline{a}}{\underline{b}}, \frac{\underline{a}}{\overline{b}}, \frac{\overline{a}}{\underline{b}}, \frac{\overline{a}}{\overline{b}}\}, \max\{\frac{\underline{a}}{\underline{b}}, \frac{\underline{a}}{\overline{b}}, \frac{\overline{a}}{\underline{b}}, \frac{\overline{a}}{\overline{b}}\}]$	$\emptyset$
Convex Hull $\mathbf{a}, \mathbf{b}$	$[\min\{\underline{a}, \underline{b}\}, \max\{\overline{a}, \overline{b}\}]$	$\mathbf{b}$ if $\mathbf{a} = \emptyset$; or $\mathbf{a}$ if $\mathbf{b} = \emptyset$
Intersection $\mathbf{a} \cap \mathbf{b}$	$[\max\{\underline{a}, \underline{b}\}, \min\{\overline{a}, \overline{b}\}]$ if $\max\{\underline{a}, \underline{b}\} \leq \min\{\overline{a}, \overline{b}\}$; Otherwise, $\emptyset$	$\emptyset$
Disjoint	*True* if $\mathbf{a} \cap \mathbf{b} = \emptyset$; *False*, otherwise	*True*
Absolute value $\lvert\mathbf{a}\rvert$	$\max\{\lvert\underline{a}\rvert, \lvert\overline{a}\rvert\}$	NaN_empty
Midpoint $\mathbf{a}$	$(\underline{a} + \overline{a})/2$	NaN_empty
Width $\mathbf{a}$	$\overline{a} - \underline{a}$	NaN_empty

Table C.1: Elementary interval operations

> *Containment Condition:* Let $\mathbf{a} = [\underline{a}, \overline{a}]$ and $\mathbf{b} = [\underline{b}, \overline{b}]$ be intervals. Let $\mathbf{c} = [\underline{c}, \overline{c}]$ be the interval result of computing **a op b** where op is defined in Table C.1. If **c** is nonempty, then **c** must contain the exact mathematical interval **a op b**.

In other words, interval arithmetic on nonempty machine intervals requires that we round down the lower bound and round up the upper bound to guarantee that the machine interval result contains the true mathematical interval result. This is needed to propagate guaranteed error bounds. A good implementation will round down and round up to the nearest possible floating point numbers, in order to get the narrowest possible machine intervals. But coarser rounding is enough to get a correct implementation. For more information on interval arithmetic specifications, one may refer to [14].

> *Advice to implementors:* In implementations of interval BLAS, a warning message should be provided to users whenever there is no finite machine interval that satisfies the containment condition during computations.

With interval arithmetic, one may automatically bound truncation error, round-off error, and even error in the original data to obtain machine intervals that are guaranteed to contain the true mathematical result of a computation. However, simply changing floating point numbers in an algorithm into intervals and all floating point operations into interval operations may result in such wide intervals that the output is useless in practice. For example, $[-100, 200]$ is a correct but probably useless bound for a true result of 3.1416. To apply the interval BLAS routines effectively, appropriate algorithms should be used that attempt to keep interval widths narrow. Many such algorithms are available in the literature. Readers may find a list of reference books, websites, software packages, and applications in [2, 12, 1].

C.4.2 Functionality

This chapter defines the functionality and language bindings for both the interval BLAS routines, and for selected mathematical operations on: intervals; interval vectors; and, dense, banded, and triangular interval matricies. Neither sparse data structures, nor complex intervals are treated.

Sections C.4.2 – C.4.2 outline the functionality of the proposed routines in tabular form. Sections C.2–C.4.5 present the language bindings for the proposed routines in the functionality tables.

Interval Vector Operations

Table C.2 lists interval vector reduction operations. Table C.3 lists interval vector operations. Table C.4 lists interval vector operations which involve only data movement.

Dot product	$\mathbf{r} \leftarrow \beta\mathbf{r} + \alpha\mathbf{x}^T\mathbf{y}$	DOT_I
Vector norms	$r \leftarrow \lVert\mathbf{x}\rVert_1, r \leftarrow \lVert\mathbf{x}\rVert_2$	
	$r \leftarrow \lVert\mathbf{x}\rVert_\infty$	NORM_I
Sum	$\mathbf{r} \leftarrow \sum_i \mathbf{x}_i$	SUM_I
Max magnitude & location	$k, \mathbf{x}_k;\ k = \arg\ \max_i\{\lvert\underline{x}_i\rvert, \lvert\overline{x}_i\rvert\}$	AMAX_VAL_I
Min absolute value & location	$k, \mathbf{x}_k;\ k = \arg\ \min_i\{\lvert\underline{x}_i\rvert, \lvert\overline{x}_i\rvert\}$	AMIN_VAL_I
Sum of squares	$(\mathbf{a}, \mathbf{b}) \leftarrow \sum_i \mathbf{x}_i^2,\ \mathbf{a} \cdot \mathbf{b}^2 = \sum_i \mathbf{x}_i^2$	SUMSQ_I

Table C.2: Reduction Operations

Reciprocal scale	$\mathbf{x} \leftarrow \mathbf{x}/\alpha$	RSCALE_I
Scaled interval vector accumulation	$\mathbf{y} \leftarrow \alpha\mathbf{x} + \beta\mathbf{y}$	AXPBY_I
Scaled interval vector accumulation	$\mathbf{w} \leftarrow \alpha\mathbf{x} + \beta\mathbf{y}$	WAXPBY_I
Scaled interval vector cancellation	$\mathbf{y} \leftarrow \alpha\mathbf{x} \ominus \beta\mathbf{y}$	CANCEL_I
Scaled interval vector cancellation	$\mathbf{w} \leftarrow \alpha\mathbf{x} \ominus \beta\mathbf{y}$	WCANCEL_I

Table C.3: Interval Vector Operations

Copy	$\mathbf{y} \leftarrow \mathbf{x}$	COPY_I
Swap	$\mathbf{y} \leftrightarrow \mathbf{x}$	SWAP_I
Permute vector	$\mathbf{x} \leftarrow P\mathbf{x}$	PERMUTE_I

Table C.4: Data Movement with Interval Vector Operations

Interval Matrix-Vector Operations

Table C.5 lists interval matrix-vector operations.

Matrix vector product	$\mathbf{y} \leftarrow \alpha \mathbf{A}\mathbf{x} + \beta \mathbf{y}$	GE,GB,SY,SB,SP	MV_I
	$\mathbf{y} \leftarrow \alpha \mathbf{A}^T\mathbf{x} + \beta \mathbf{y}$	GE,GB	MV_I
	$\mathbf{x} \leftarrow \mathbf{T}\mathbf{x}, \mathbf{x} \leftarrow \mathbf{T}^T\mathbf{x}$	TR, TB, TP	MV_I
Triangular solve	$\mathbf{x} \leftarrow \alpha \mathbf{T}^{-1}\mathbf{x}, \mathbf{x} \leftarrow \alpha \mathbf{T}^{-T}\mathbf{x}$	TR, TB, TP	SV_I
Rank one updates	$\mathbf{A} \leftarrow \alpha \mathbf{x}\mathbf{y}^T + \beta \mathbf{A}$	GE,SY,SP	R_I

Table C.5: Interval Matrix-vector Operations

Interval Matrix Operations

Table C.6 lists single interval matrix operations and interval matrix operations that involve $O(n^2)$ floating point operations. The matrix $\mathbf{T}$ represents an upper or lower triangular interval matrix. $\mathbf{D}$ represents a diagonal interval matrix. Table C.7 lists the interval matrix-matrix operations that involve $O(n^3)$ floating point operations and Table C.8 lists those operations that involve only data movement.

Matrix norms	$r \leftarrow \|\|\mathbf{A}\|\|_1, r \leftarrow \|\|\mathbf{A}\|\|_F$,	GE,GB,SY,SB,	_NORM_I
	$r \leftarrow \|\|\mathbf{A}\|\|_\infty, r \leftarrow \|\|\mathbf{A}\|\|_{\max}$	SP,TR,TB,TP	
Diagonal scaling	$\mathbf{A} \leftarrow \mathbf{D}\mathbf{A}, \mathbf{A} \leftarrow \mathbf{A}\mathbf{D}$	GE, GB	_DIAG_SCALE_I
Two sided diagonal scaling	$\mathbf{A} \leftarrow \mathbf{D}_1\mathbf{A}\mathbf{D}_2$	GE, GB	_LRSCALE_I
Two sided diagonal scaling	$\mathbf{A} \leftarrow \mathbf{D}\mathbf{A}\mathbf{D}$	SY, SB, SP	_LRSCALE_I
	$\mathbf{A} \leftarrow \mathbf{A} + \mathbf{B}\mathbf{D}$	GE, GB	
Matrix acc and scale	$\mathbf{B} \leftarrow \alpha \mathbf{A} + \beta \mathbf{B}$,	GE,GB,SY,SB,	_ACC_I
	$\mathbf{B} \leftarrow \alpha \mathbf{A}^T + \beta \mathbf{B}$	SP,TR,TB,TP	
Matrix add and scale	$\mathbf{C} \leftarrow \alpha \mathbf{A} + \beta \mathbf{B}$	GE,GB,SY,SB,	_ADD_I
		SP,TR,TB,TP	

Table C.6: Matrix Operations – $O(n^2)$ floating point operations

Matrix matrix product	$\mathbf{C} \leftarrow \alpha \mathbf{A}\mathbf{B} + \beta \mathbf{C}$, $\mathbf{C} \leftarrow \alpha \mathbf{A}^T\mathbf{B} + \beta \mathbf{C}$,	GE,GB,SY,SB	MM_I
	$\mathbf{C} \leftarrow \alpha \mathbf{A}\mathbf{B}^T + \beta \mathbf{C}$, $\mathbf{C} \leftarrow \alpha \mathbf{A}^T\mathbf{B}^T + \beta \mathbf{C}$		
	$\mathbf{C} \leftarrow \alpha \mathbf{B}\mathbf{A} + \beta \mathbf{C}$, $\mathbf{C} \leftarrow \alpha \mathbf{B}^T\mathbf{A} + \beta \mathbf{C}$,	GB	MM_I
	$\mathbf{C} \leftarrow \alpha \mathbf{B}\mathbf{A}^T + \beta \mathbf{C}$, $\mathbf{C} \leftarrow \alpha \mathbf{B}^T\mathbf{A}^T + \beta \mathbf{C}$		
Triangular multiply	$\mathbf{B} \leftarrow \alpha \mathbf{T}\mathbf{B}$, $\mathbf{B} \leftarrow \alpha \mathbf{B}\mathbf{T}$	TR, TB	MM_I
	$\mathbf{B} \leftarrow \alpha \mathbf{T}^T\mathbf{B}$, $\mathbf{B} \leftarrow \alpha \mathbf{B}\mathbf{T}^T$		
Triangular solve	$\mathbf{B} \leftarrow \alpha \mathbf{T}^{-1}\mathbf{B}$, $\mathbf{B} \leftarrow \alpha \mathbf{B}\mathbf{T}^{-1}$	TR, TB	SM_I
	$\mathbf{B} \leftarrow \alpha \mathbf{T}^{-T}\mathbf{B}$, $\mathbf{B} \leftarrow \alpha \mathbf{B}\mathbf{T}^{-T}$		

Table C.7: Matrix Operations – $O(n^3)$ floating point operations

Matrix copy	$\mathbf{B} \leftarrow \mathbf{A}$	GE,GB,SY,SB,SP,TR,TB,TP	_COPY_I
	$\mathbf{B} \leftarrow \mathbf{A}^T$	GE, GB	_COPY_I
Matrix transpose	$\mathbf{A} \leftarrow \mathbf{A}^T$	GE	_TRANS_I
Permute matrix	$\mathbf{A} \leftarrow \mathbf{PA}, \mathbf{A} \leftarrow \mathbf{AP}$	GE	_PERMUTE_I

Table C.8: Data Movement with Interval Matrices

Set Operations Involving Interval Vectors

Table C.9 lists set operations for interval vectors.

Enclosed	$\mathbf{x}$ is enclosed in $\mathbf{y}$ if $\mathbf{x} \subseteq \mathbf{y}$	ENCV_I
Interior	$\mathbf{x}$ is enclosed in the interior of $\mathbf{y}$	INTERIORV_I
Disjoint	$\mathbf{x}$ and $\mathbf{y}$ are disjoint if $\mathbf{x} \cap \mathbf{y} = \emptyset$	DISJV_I
Intersection	$\mathbf{y} \leftarrow \mathbf{x} \cap \mathbf{y}, \mathbf{z} \leftarrow \mathbf{x} \cap \mathbf{y}$	INTERV_I, WINTERV_I
Hull	the convex hull of $\mathbf{x}$ and $\mathbf{y}$	HULLV_I, WHULLV_I

Table C.9: Set Operations for Interval Vectors

Set Operations Involving Interval Matrices

Table C.10 lists set operations for interval matrices.

Enclosed	$\mathbf{A}$ is enclosed in $\mathbf{B}$ if $\mathbf{A} \subseteq \mathbf{B}$	GE,GB,SY,SB, SP,TR,TB,TP	_ENCM_I
Interior	$\mathbf{A}$ is enclosed in the interior of $\mathbf{B}$	GE,GB,SY,SB, SP,TR,TB,TP	_INTERIORM_I
Disjoint	$\mathbf{A}$ and $\mathbf{B}$ are disjoint if $\mathbf{A} \cap \mathbf{B} = \emptyset$	GE,GB,SY,SB, SP,TR,TB,TP	_DISJM_I
Intersection	$\mathbf{B} \leftarrow \mathbf{A} \cap \mathbf{B}$, $\mathbf{C} \leftarrow \mathbf{A} \cap \mathbf{B}$	GE,GB,SY,SB, SP,TR,TB,TP	_INTERM_I, _WINTERM_I
Hull	the convex hull of $\mathbf{A}$ and $\mathbf{B}$	GE,GB,SY,SB, SP,TR,TB,TP	_HULLM_I, _WHULLM_I

Table C.10: Set Operations for Interval Matrices

Utility Functions Involving Interval Vectors

Table C.11 lists some utility operations for interval vectors.

Utility Functions Involving Interval Matrices

Table C.12 lists some utility operations for interval matrices.

Empty element	k if $\mathbf{x}_k = \emptyset$; or -1	EMPTYELEV_I
Left endpoint	$v \leftarrow \underline{x}$	INFV_I
Right endpoint	$v \leftarrow \overline{x}$	SUPV_I
Midpoint	$v \leftarrow (\underline{x} + \overline{x})/2$	MIDV_I
Width	$v \leftarrow \overline{x} - \underline{x}$	WIDTHV_I
Construct	$\mathbf{x} \leftarrow u, v$	CONSTRUCTV_I

Table C.11: Utility Operations for Interval Vectors

Empty element	if $\mathbf{A}$ has an empty interval element	GE,GB,SY,SB, SP,TR,TB,TP	_EMPTYELEM_I
Left endpoint	$C \leftarrow \underline{A}$	GE,GB,SY,SB, SP,TR,TB,TP	_INFM_I
Right endpoint	$C \leftarrow \overline{A}$	GE,GB,SY,SB, SP,TR,TB,TP	_SUPM_I
Midpoint	$C \leftarrow (\underline{A} + \overline{A})/2$	GE,GB,SY,SB, SP,TR,TB,TP	_MIDM_I
Width	$C \leftarrow \overline{A} - \underline{A}$	GE,GB,SY,SB,	_WIDTHM_I
Construct	$\mathbf{A} \leftarrow B, C$	GE,GB,SY,SB, SP,TR,TB,TP	_CONSTRUCTM_I

Table C.12: Utility Operations

C.4.3 Interface Issues

Naming Conventions

The naming conventions are the same as described in section 2.3.1 except that the suffix _I (or _i) is added to indicate an interval BLAS routine.

Interface Issues for Fortran 95

Design of the Fortran 95 Interfaces

The Fortran 95 binding is defined in a module. The specific interfaces in this module should declare the default interval data type as `TYPE(INTERVAL)`.

> *Advice to implementors:* In the Fortran 95 interfaces, it is assumed that `INTERVAL` is a derived type. However, in compilers that support an intrinsic interval type, it is recommended that an alternate module that contains appropriately modified declarations also be supplied. For example, `TYPE(INTERVAL), INTENT(IN) :: ALPHA` could become `INTERVAL, INTENT(IN) :: ALPHA` in a recommended alternate module.

The Fortran 95 interval BLAS routines are consistent with regard to generic interfaces, precision, rank, assumed-shape arrays, derived types, operator arguments and `CMACH` values, and error handling as described in section 2.4 of this document. However, in the interval BLAS, α and β are intervals; and their default values are `alpha = [1,1], beta = [0,0]`.
Error handling is as defined in section 2.4.6.

Format of the Fortran 95 bindings

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

```
SUBROUTINE axpby_i( x, y [, alpha] [, beta] )
  TYPE(INTERVAL) (<wp>), INTENT (IN) :: x (:)
  TYPE(INTERVAL) (<wp>), INTENT (INOUT) :: y (:)
  TYPE(INTERVAL) (<wp>), INTENT (IN), OPTIONAL :: alpha, beta
```

Variables in interval BLAS routines should be specified as INTEGER, REAL, TYPE(INTERVAL) or types defined in MODULE blas_operator_arguments. The precision of a real or interval variable is denoted by <wp> where

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

Interface Issues for Fortran 77

The interval BLAS Fortran 77 binding is consistent with ANSI standard Fortran 77 except the following:

- Subroutine names are not limited to six significant characters.
- Subroutine names contain one or more underscores.
- Subroutines may use the INCLUDE statement for include files.

In interval BLAS Fortran 77 binding, α and β are intervals and their default values are:
ALPHA = [1.0, 1.0] and BETA = [0.0, 0.0]. Without assuming an intrinsic interval data type, an interval, say α, will be declared as REAL or DOUBLE PRECISION ALPHA(2); an interval vector will be stored as REAL or DOUBLE PRECISION X(2,*); and a general interval matrix will be defined as REAL or DOUBLE PRECISION A(2,LDA, *).

> *Advice to implementors:* On Fortran 77 compilers that have an intrinsic interval data type, an interval vector will be stored as INTERVAL X(*), and a general interval matrix will be defined as INTERVAL A(LDA, *).

The Fortran 77 interval BLAS routines are consistent with regard to indexing of vector and matrix operands, operator arguments and CMACH values, array arguments, matrix storage schemes, and error handling as described in section 2.5 of this document but with interval variables.
Error handling is as defined in section 2.5.6.

Format of the Fortran 77 bindings

Each interface is summarized in the form of a SUBROUTINE statement (or a FUNCTION statement). For example:

```
SUBROUTINE BLAS_xAXPBY_I( N, ALPHA, X, INCX, BETA, Y, INCY )
  INTEGER    INCX, INCY, N
  <type>     ALPHA(2), BETA(2)
  <type>     X(2,*), Y(2,*)
```

Floating point variables are denoted by the keyword <type> which may be REAL or DOUBLE PRECISION, and should agree with the x letter in the naming convention of the routine.

Interface Issues for C

The interface is expressed in terms of ANSI/ISO C. *All interval arguments are accepted as* `float *` or `double *`. An interval element consists of two consecutive memory locations of the underlying data type (i.e., `float` or `double`), where the first location contains the lower bound of the interval, and the second contains the upper bound of the interval.
The C interval BLAS routines are consistent with regard to indexing of vector and matrix operands, operator arguments and `CMACH` values, array arguments, matrix storage schemes, and error handling that described in section 2.6 of this document but with interval variables. The default value for intervals `alpha` and `beta` are `alpha` $= [1.0, 1.0]$ and `beta` $= [0.0, 0.0]$.
Error handling is as defined in section 2.6.9.

Format of the C bindings

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

```
void BLAS_xaxpby_i( int n, <interval> alpha, const <interval_array> x,
                    int incx, <interval> beta, <interval_array> y,
                    int incy)
```

In the C binding, we use the keywords <`interval`> and <`interval_array`> to indicate if an argument is a single interval or an interval vector/matrix. In fact, <`interval`> and <`interval_array`> can be `float *` or `double *`. A real number, not an interval, will be indicated by the keyword `SCALAR`. A vector/matrix of real numbers, not intervals, will be specified by `RARRAY`. The precisions of `SCALAR`, `RARRAY` can be `float` or `double`. They will agree with the `x` letter in the naming convention of the routine. However, in some routines, not all floating point variables will be the same type. If this is the case, then a variable may be denoted by the keywords `SCALAR_IN` or `SCALAR_INOUT`. `SCALAR_IN` can be `float` or `double`; and `SCALAR_INOUT` and `RARRAY` can be `float *` or `double *`.

C.4.4 Numerical Accuracy and Environmental Enquiry

The semantics of interval arithmetic require us to have another environmental enquiry function to supplement the routine FPINFO described in sections 1.6 and 2.7. Here we will specify the additional routine FPINFO_I to determine how tightly the containment property of interval arithmetic is maintained.
To establish notation, let $\mathbf{a} = [\underline{a}, \overline{a}]$ and $\mathbf{b} = [\underline{b}, \overline{b}]$ be machine intervals, let `op` be one of the operations $+$, $-$, $\ominus$, $\times$ and $\div$, let $\mathbf{c} = [\underline{c}, \overline{c}] = \mathbf{a}$ `op` $\mathbf{b}$ be the exact mathematical interval result of $\mathbf{a}$ `op` $\mathbf{b}$, and let $\mathbf{c}^* = [\underline{c}^*, \overline{c}^*] = fl(\mathbf{a}$ `op` $\mathbf{b})$ be the machine interval computed containing $\mathbf{c}$. Let $\epsilon_I > 0$ be defined as the smallest number such that for all $\mathbf{a}$, $\mathbf{b}$ and `op` where overflow and underflow do not occur in computing $\mathbf{c}^*$, then

$$
\begin{aligned}
\underline{c} &\geq \min\{\underline{c}^*(1+\epsilon_I), \underline{c}^*(1-\epsilon_I)\} \\
\overline{c} &\leq \max\{\overline{x}^*(1+\epsilon_I), \overline{c}^*(1-\epsilon_I)\}
\end{aligned}
$$

In other words, ϵ_I measures how much the exact mathematical interval bounds are rounded out to get the machine interval result. When the machine interval is tight, i.e. as narrow as possible, then $\epsilon_I = BASE^{1-T}$, where $BASE$ and T are values returned by FPINFO. But ϵ_I could be larger depending on the implementation, leading us to the following environmental enquiry:

Value of CMACH	Name of value returned by FPINFO_I	Description
blas_base	BASE	base of the machine
blas_t_i	T_I	effective number of base BASE digits, such that $\epsilon_I = BASE^{1-T_I}$
blas_rnd_i	RND_I	when interval arithmetic is implemented with correct IEEE-style directed rounding
blas_eps_i	EPS_I	ϵ_I as defined above.

C.4.5 Language Bindings

Each specification of a routine will correspond to an operation outlined in the functionality tables. Operations are organized analogous to the order in which they are presented in the functionality tables. The format of the language bindings is as described in section 2.8.

Overview

- Reduction Operations (section C.4.5)
  - DOT_I (Dot product)
  - NORM_I (Interval vector norms)
  - SUM_I (Sum)
  - AMIN_VAL_I (Min absolute value & location)
  - AMAX_VAL_I (Max absolute value & location)
  - SUMSQ_I (Sum of squares)
- Interval Vector Operations (section C.4.5)
  - RSCALE_I (Reciprocal Scale)
  - AXPBY_I (Scaled vector accumulation)
  - WAXPBY_I (Scaled vector addition)
  - CANCEL_I (Scaled cancellation)
  - WCANCEL_I (Scaled cancellation)
- Data Movement with Interval Vectors (section C.4.5)
  - COPY_I (Interval vector copy)
  - SWAP_I (Interval vector swap)
  - PERMUTE_I (Permute interval vector)
- Interval Matrix-Vector Operations (section C.4.5)
  - {GE,GB}MV_I (Interval matrix vector product)
  - {SY,SB,SP}MV_I (Interval symmetric matrix vector product)
  - {TR,TB,TP}MV_I (Interval triangular matrix vector product)
  - {TR,TB,TP}SV_I (Interval triangular solve)
  - GER_I (Rank one update)

  - {SY,SP}R_I (Symmetric rank one update)

- Interval Matrix Operations (section C.4.5)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_NORM_I (Interval matrix norms)
  - {GE,GB}_DIAG_SCALE_I (Diagonal scaling)
  - {GE,GB}_LRSCALE_I (Two-sided diagonal scaling)
  - {SY,SB,SP}_LRSCALE_I (Two-sided diagonal scaling of a symmetric interval matrix)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_ACC_I (Matrix accumulation and scale)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_ADD_I (Matrix add and scale)
- Interval Matrix-Matrix Operations (section C.4.5)
  - GEMM_I (General interval Matrix Matrix product)
  - SYMM_I (Symmetric interval matrix matrix product)
  - TRMM_I (Triangular interval matrix matrix multiply)
  - TRSM_I (Interval triangular solve)
- Data Movement with Interval Matrices (section C.4.5)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_COPY_I (Matrix copy)
  - GE_TRANS_I (Matrix transposition)
  - GE_PERMUTE_I (Permute an interval matrix)
- Set Operations Involving Interval Vectors (section C.4.5)
  - ENCV_I (Checks if an interval vector is enclosed in another interval vector)
  - INTERIORV_I (Checks if an interval vector is enclosed in the interior of another interval vector)
  - DISJV_I (Checks if two interval vectors are disjoint)
  - INTERV_I (Intersection of an interval vector with another)
  - WINTERV_I (Intersection of two interval vectors)
  - HULLV_I (Convex hull of an interval vector with another)
  - WHULLV_I (Convex hull of two interval vectors)
- Set Operations Involving Interval Matrices (section C.4.5)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_ENCM_I (Checks if an interval matrix is enclosed in another interval matrix)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_INTERIORM_I (Checks if an interval matrix is enclosed in the interior of another interval matrix)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_DISJM_I (Checks if two interval matrices are disjoint)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_INTERM_I (Elementwise intersection of an interval matrix with another)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_WINTERM_I (Elementwise intersection of two interval matrices)

  - {GE,GB,SY,SB,SP,TR,TB,TP}_HULLM_I (Convex hull of an interval matrix with another)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_WHULLV_I (Convex hull of two interval matrices)

- Utility Functions Involving Interval Vectors (section C.4.5)
  - EMPTYELEV_I (Empty entry and location)
  - INFV_I (The left endpoint of an interval vector)
  - SUPV_I (The right endpoint of an interval vector)
  - MIDV_I (The approximate midpoint of an interval vector)
  - WIDTHV_I (The elementwise width of an interval vector)
  - CONSTRUCTV_I (Constructs an interval vector from two floating point vectors)

- Utility Functions Involving Interval Matrices (section C.4.5)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_EMPTYELEM_I (Empty entry and location)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_INFM_I (The left endpoint of an interval matrix)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_SUPM_I (The right endpoint of an interval matrix)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_MIDM_I (The approximate midpoint of an interval matrix)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_WIDTHM_I (Elementwise width of an interval matrix)
  - {GE,GB,SY,SB,SP,TR,TB,TP}_CONSTRUCTM_I (Constructs an interval matrix from two given floating point matrices)

- Environmental Enquiry (section C.4.5)
  - FPINFO_I (Environmental enquiry)

Reduction Operations

DOT_I (Dot Product) $\mathbf{r} \leftarrow \beta\mathbf{r} + \alpha\mathbf{x}^T\mathbf{y}$

The routine `DOT_I` adds the scaled dot product of two interval vectors $\mathbf{x}$ and $\mathbf{y}$ into a scaled interval $\mathbf{r}$. The routine returns immediately if `n` is less than zero, or, if `beta` is equal to [1,1] and either `alpha` is equal to [0,0] or `n` is equal to zero. If `alpha` is equal to [0,0] then $\mathbf{x}$ and $\mathbf{y}$ are not read. Similarly, if `beta` is equal to [0,0], $\mathbf{r}$ is not referenced. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.
179, one.

- Fortran 95 binding:

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

- Fortran 77 binding:

```
SUBROUTINE BLAS_xDOT_I( N, ALPHA, X, INCX, BETA, Y, INCY, R )
INTEGER          INCX, INCY, N
<type>           ALPHA( 2 ), BETA( 2 ), R( 2 )
<type>           X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xdot_i( int n, const <interval> alpha, const <interval_array> x,
                  int incx, const <interval> beta, const <interval_array> y,
                  int incy, <interval> r );
```

Advice to users: The scaling parameters `alpha` and `beta` are intervals. If any one of them is a real number in applications, the user needs to convert it into its interval representation first, and then use the routine.

NORM_I (Interval vector norms) $r \leftarrow ||\mathbf{x}||_1,\, ||\mathbf{x}||_2, ||\mathbf{x}||_\infty$

The routine NORM_I computes the $||\cdot||_1$, $||\cdot||_2$, or $||\cdot||_\infty$ of a vector x depending on the value passed as the norm operator argument.
If `n` is less than or equal to zero, this routine returns immediately with the output scalar `r` set to zero. The resulting scalar `r` is always real and its value is as defined in section 2.1.1, provided that $|\mathbf{x}_i| = \max\{|\underline{x}_i|, |\overline{x}_i|\}$.
As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

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

- Fortran 77 binding:

```
<type>   FUNCTION BLAS_xNORM_I( NORM, N, X, INCX )
INTEGER           INCX, N, NORM
<type>            X( 2, * )
```

- C binding:

```
void BLAS_xnorm_i( enum blas_norm_type norm, int n, const <interval_array> x,
                   int incx, SCALAR_INOUT r );
```

Advice to implementors: In finite precision floating point arithmetic, an upper bound, preferably the least machine representable upper bound, for the mathematical value should be returned for the norms.

SUM_I (Sum the entries of an interval vector) $\mathbf{r} \leftarrow \sum_{i=0}^{n-1} \mathbf{x}_i$

The routine `SUM_I` returns the sum of the entries of an interval vector $\mathbf{x}$. If `n` is less than or equal to zero, this routine returns immediately with the output interval `r` set to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
SUBROUTINE sum_i( x, r )
  TYPE(INTERVAL)(<wp>), INTENT(IN) :: x(:)
  TYPE(INTERVAL)(<wp>), INTENT(OUT) :: r
where
  x has shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xSUM_I( N, INCX, X, R )
INTEGER           INCX, N
<type>            X( 2, * )
<type>            R( 2 )
```

- C binding:

```
void BLAS_xsum_i( int n, int incx, const <interval_array> x, <interval> r );
```

AMIN_VAL_I ($\min_{0 \le i < n}\{|\underline{x}_i|, |\overline{x}_i|\}$ & location) $k, r \leftarrow \min\{|\underline{x}_k|, |\overline{x}_k|\} = r = \min_{0 \le i < n}\{|\underline{x}_i|, |\overline{x}_i|\}$

The routine `AMIN_VAL_I` finds the index of the component of an interval vector such that the absolute value of the lower or upper bounds of the component is the smallest among the absolute values of the lower and upper bounds of all components of the interval vector. When the value of the `n` argument is less than or equal to zero, the routine should initialize the output `k` to negative one or zero, and `r` to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

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

- Fortran 77 binding:

```
SUBROUTINE BLAS_xAMIN_VAL_I( N, X, INCX, K, R )
INTEGER           INCX, K, N
<type>            X( 2, * )
<type>            R
```

- C binding:

```
void BLAS_xamin_val_i( int n, const <interval_array> x, int incx, int k,
                       SCALAR_INOUT r );
```

AMAX_VAL_I (Max absolute value & location) $k, r \leftarrow \max\{|\underline{x}_k|, |\overline{x}_k|\} = r = \max_{0 \le i < n} \{|\underline{x}_i|, |\overline{x}_i|\}$

The routine **AMAX_VAL_I** finds the index of the component of an interval vector such that the absolute value of the lower or upper bounds of the component has the largest value among the absolute values of the lower and upper bounds of all components of the interval vector. When the value of the **n** argument is less than or equal to zero, the routine should initialize the output **k** to negative one or zero, and **r** to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

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

- Fortran 77 binding:

```
SUBROUTINE BLAS_AMAX_VAL_I( N, X, INCX, K, R )
INTEGER           INCX, K, N
<type>            X( 2, * )
<type>            R
```

- C binding:

```
void BLAS_xamax_val_i( int n, const <interval_array> x, int incx, int k,
                       SCALAR_INOUT  r );
```

SUMSQ_I (Sum of squares) $(scl, ssq) \leftarrow \sum \mathbf{x}_i^2$

The routine SUMSQ_I returns the intervals *scl* and *ssq* such that

$$scl^2 * ssq = scale^2 * sumsq + \sum_{i=0}^{n-1} \mathbf{x}_i^2.$$

The value of *sumsq* is assumed to be at least unity and the value of *ssq* will then satisfy $1.0 \le ssq \le (sumsq + n)$. It is assumed that *scale* is to be non-negative, and *scl* returns the value

$$scl = \max_{0 \le i < n} (scale, |\mathbf{x}_i|).$$

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

- Fortran 95 binding:

```
        SUBROUTINE sumsq_i( x, ssq, scl )
          TYPE(INTERVAL)(<wp>), INTENT(IN) :: x(:)
          TYPE(INTERVAL)(<wp>), INTENT(INOUT) :: ssq, scl
        where
          x has shape (n)
```

- Fortran 77 binding:

```
        SUBROUTINE BLAS_xSUMSQ_I( N, X, INCX, SSQ, SCL )
        INTEGER            INCX, N
        <type>             X( 2, * )
        <type>             SCL( 2 ), SSQ( 2 )
```

- C binding:

```
void BLAS_xsumsq_i( int n, const <interval_array> x, int incx, <interval> ssq,
                    <interval> scl );
```

Interval Vector Operations

RSCALE_I (Reciprocal Scale of an interval vector) $\mathbf{x} \leftarrow \mathbf{x}/\alpha$

The routine RSCALE_I updates the entries of an interval vector $\mathbf{x}$ by the scale interval $1/\alpha$ provided that $0 \notin \alpha$. If n is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
        SUBROUTINE rscale_i( alpha, x )
          TYPE(INTERVAL)(<wp>), INTENT(INOUT) :: x(:)
          TYPE(INTERVAL)(<wp>), INTENT(IN) :: alpha
        where
          x has shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xRSCALE_I( N, ALPHA, X, INCX )
INTEGER          INCX, N
<type>           ALPHA( 2 )
<type>           X( 2, * )
```

- C binding:

```
void BLAS_xrscale_i( int n, <interval> alpha, <interval_array> x, int incx );
```

AXPBY_I (Scaled vector accumulation) $\mathbf{y} \leftarrow \alpha\mathbf{x} + \beta\mathbf{y}$

The routine AXPBY_I scales the interval vector $\mathbf{x}$ by the interval α and the interval vector $\mathbf{y}$ by β, adds these two vectors to one another and stores the result in the vector $\mathbf{y}$. If n is less than or equal to zero, or if α is equal to [0,0] and β equal to [1,1], this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

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

- Fortran 77 binding:

```
SUBROUTINE BLAS_xAXPBY_I( N, ALPHA, X, INCX, BETA, Y, INCY )
INTEGER           INCX, INCY, N
<type>            ALPHA( 2 ), BETA( 2 )
<type>            X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xaxpby_i( int n, <interval> alpha, <interval_array> x, int incx,
                    <interval> beta, <interval_array> y, int incy );
```

WAXPBY_I (Scaled vector addition) $\mathbf{w} \leftarrow \alpha\mathbf{x} + \beta\mathbf{y}$

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

- Fortran 95 binding:

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

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xWAXPBY_I( N, ALPHA, X, INCX, BETA, Y, INCY, W,
     $                           INCW )
      INTEGER            INCW, INCX, INCY, N
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             W( 2, * ), X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_wxaxpby_i( int n, <interval> alpha, const <interval_array> x,
                     int incx, <interval> beta, const <interval_array> y,
                     int incy, <interval_array> w, int incw );
```

CANCEL_I (Scaled cancellation) $\mathbf{y} \leftarrow \alpha\mathbf{x} \ominus \beta\mathbf{y}$

The operation cancel, $\ominus$, between two intervals $\mathbf{a}$ and $\mathbf{b}$ is defined as $\mathbf{a} \ominus \mathbf{b} = [\underline{a} - \underline{b}, \overline{a} - \overline{b}]$ if $(\underline{a} - \underline{b}) \leq (\overline{a} - \overline{b})$; Otherwise, $\emptyset$. The routine `CANCEL_I` scales the interval vector $\mathbf{x}$ by the interval α and the interval vector $\mathbf{y}$ by β, updates $\mathbf{y}_i$ with $\alpha\mathbf{x}_i \ominus \beta\mathbf{y}_i$, $\forall 0 \leq i < n$. If `n` is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

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

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xCANCEL_I( N, ALPHA, X, INCX, BETA, Y, INCY )
      INTEGER            INCX, INCY, N
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xcancel_i( int n, <interval> alpha, <interval_array> x, int incx,
                     <interval> beta, <interval_array> y, int incy );
```

WCANCEL_I (Scaled cancellation) $\mathbf{w} \leftarrow \alpha\mathbf{x} \ominus \beta\mathbf{y}$

The operation cancel, $\ominus$, between two intervals $\mathbf{a}$ and $\mathbf{b}$ is defined as $\mathbf{a} \ominus \mathbf{b} = [\underline{a} - \underline{b}, \overline{a} - \overline{b}]$ if $(\underline{a} - \underline{b}) \leq (\overline{a} - \overline{b})$; Otherwise, $\emptyset$. The routine WCANCEL_I scales the interval vector $\mathbf{x}$ by the interval α and the interval vector $\mathbf{y}$ by β, stores $\alpha\mathbf{x}_i \ominus \beta\mathbf{y}_i$ in $\mathbf{w}_i$ for $0 \leq i < n$. If n is less than or equal to zero, this routine returns immediately. As described in section 2.5.3, the value incx or incy or incw less than zero is permitted. However, if either incx or incy or incw is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

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

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xWCANCEL_I( N, ALPHA, X, INCX, BETA, Y, INCY, W,
     $                            INCW )
      INTEGER            INCW, INCX, INCY, N
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             W( 2, * ), X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xwcancel_i( int n, <interval> alpha, <interval_array> x, int incx,
                      <interval> beta, <interval_array> y, int incy,
                      <interval_array> w, int incw );
```

Data Movement with Interval Vectors

COPY_I (Interval vector copy) $\mathbf{y} \leftarrow \mathbf{x}$

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

- Fortran 95 binding:

```
SUBROUTINE copy_i( x, y )
  TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
  TYPE(INTERVAL) (<wp>), INTENT(OUT) :: y(:)
where
  x and y have shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xCOPY_I( N, X, INCX, Y, INCY )
INTEGER            INCX, INCY, N
<type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xcopy_i( int n, const <interval_array> x, int incx,
                   <interval_array> y, int incy );
```

SWAP_I (Interval vector swap) $\mathbf{y} \leftrightarrow \mathbf{x}$

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

- Fortran 95 binding:

```
SUBROUTINE swap_i( x, y )
  TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: x(:), y(:)
where
  x and y have shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xSWAP_I( N, X, INCX, Y, INCY )
INTEGER            INCX, INCY, N
<type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xswap_i( int n, <interval_array> x, int incx, <interval_array> y,
                   int incy );
```

PERMUTE_I (Permute interval vector) $\mathbf{x} \leftarrow P\mathbf{x}$

The routine `PERMUTE_I` permutes the entries of an interval vector $\mathbf{x}$ according to the permutation vector P. If `n` is less than or equal to zero, the routine returns immediately. As described in section 2.5.3, the value incx or incp less than zero is permitted. However, if either incx or incp is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

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

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xPERMUTE_I( N, P, INCP, X, INCX )
      INTEGER            INCP, INCX, N
      INTEGER            P( * )
      <type>             X( 2, * )
```

- C binding:

```
void BLAS_xpermute_i( int n, const int *p, int incp, <interval_array> x,
                      int incx );
```

Interval Matrix-Vector Operations

{GE,GB}MV_I (Interval matrix-vector multiplication) $\mathbf{y} \leftarrow \alpha \mathbf{A}\mathbf{x} + \beta \mathbf{y}, \mathbf{y} \leftarrow \alpha \mathbf{A}^T \mathbf{x} + \beta \mathbf{y}$

The routines multiply the interval vector $\mathbf{x}$ by a general (or general band) interval matrix $\mathbf{A}$ or its transpose, scales the resulting interval vector and adds it to the scaled interval vector operand $\mathbf{y}$. If m or n is less than or equal to zero or if beta is equal to [1,1] and alpha is equal to [0,0], the routine returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. For the routine GEMV_I, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine GBMV_I, if kl or ku is less than zero, or if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
      SUBROUTINE gemv_i( a, x, y [, transa] [, alpha] [, beta] )
General Band:
      SUBROUTINE gbmv_i( a, m, kl, x, y [, transa] [, alpha] [, beta] )
all:
        TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), x(:)
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: y(:)
        INTEGER INTENT(IN) :: m, kl
        TYPE(blas_trans_type), INTENT(IN), OPTIONAL :: transa
        TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      where
        a has shape (m,n) for general matrix
                    (l,n) for general banded matrix ( l > kl)
        x and y have shape n if transa = blas_no_trans (the default}
                           m if transa /= blas_no_trans
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGEMV_I( TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA,
     $                         Y, INCY )
General Band:
      SUBROUTINE BLAS_xGBMV_I( TRANS, M, N, KL, KU, ALPHA, A, LDA, X, INCX,
     $                         BETA, Y, INCY )
all:
      INTEGER            INCX, INCY, KL, KU, LDA, M, N, TRANS
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             A( 2, LDA, * ), X( 2, * ), Y( 2, * )
```

- C binding:

```
General:
void BLAS_xgemv_i( enum blas_order_type order, enum blas_trans_type trans,
                   int m, int n, <interval> alpha, const <interval_array> a,
                   int lda, const <interval_array> x, int incx, <interval> beta,
                   <interval_array> y, int incy );
General Band:
void BLAS_xgbmv_i( enum blas_order_type order, enum blas_trans_type trans,
                   int m, int n, int kl, int ku, <interval> alpha,
                   const <interval_array> a, int lda, const <interval_array> x,
                   int incx, <interval> beta, <interval_array> y, int incy );
```

{SY,SB,SP}MV_I (Interval symmetric matrix vector product) $\mathbf{y} \leftarrow \alpha \mathbf{A}\mathbf{x} + \beta \mathbf{y}$ with $\mathbf{A} = \mathbf{A}^T$

The routines multiply an interval vector **x** by a symmetric interval matrix **A**, scales the resulting interval vector and adds it to the scaled interval 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, the routine returns immediately. The operator argument `uplo` specifies if the matrix operand is an upper or lower triangular part of the symmetric matrix. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. For the routine SYMV_I, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine SBMV_I, if lda is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
Symmetric:
      SUBROUTINE symv_i( a, x, y [, uplo] [, alpha] [, beta] )
Symmetric Band:
      SUBROUTINE sbmv_i( a, x, y [, uplo] [, alpha] [, beta] )
Symmetric Packed:
      SUBROUTINE spmv_i( ap, x, y [, uplo] [, alpha] [, beta] )
```

```
all:
        TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), ap(:), x(:)
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: y(:)
        TYPE(blas_uplo_type),  INTENT(IN), OPTIONAL :: uplo
        TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      where
        x and y have shape (n)
       SY  a has shape (n,n)
       SB  a has shape (k+1,n), where k = band width
       SP  ap has shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
Symmetric:
      SUBROUTINE BLAS_xSYMV_I( UPLO, N, ALPHA, A, LDA, X, INCX, BETA, Y,
     $                        INCY )
Symmetric Band:
      SUBROUTINE BLAS_xSBMV_I( UPLO, N, K, ALPHA, A, LDA, X, INCX, BETA,
     $                        Y, INCY )
Symmetric Packed:
      SUBROUTINE BLAS_xSPMV_I( UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY )
all:
      INTEGER            INCX, INCY, K, LDA, N, UPLO
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             A( 2, LDA, * ) or AP( 2, * ), X( 2, * ),
     $                   Y( 2, * )
```

- C binding:

```
Symmetric:
void BLAS_xsymv_i( enum blas_order_type order, enum blas_uplo_type uplo, int n,
                   <interval> alpha, const <interval_array> a, int lda,
                   const <interval_array> x, int incx, <interval> beta,
                   <interval_array> y, int incy );
Symmetric Band:
void BLAS_xsbmv_i( enum blas_order_type order, enum blas_uplo_type uplo, int n,
                   int k, <interval> alpha, const <interval_array> a, int lda,
                   const <interval_array> x, int incx, <interval> beta,
                   <interval_array> y, int incy );
Symmetric Packed:
void BLAS_xspmv_i( enum blas_order_type order, enum blas_uplo_type uplo, int n,
                   <interval> alpha, const <interval_array> ap,
                   const <interval_array> x, int incx, <interval> beta,
                   <interval_array> y, int incy );
```

{TR,TB,TP}MV_I (Interval triangular matrix vector product) $\mathbf{x} \leftarrow \alpha\mathbf{T}\mathbf{x}, \mathbf{x} \leftarrow \alpha\mathbf{T}^T\mathbf{x}$

The routines multiply an interval vector **x** by a general triangular interval matrix **T** or its transpose, and copies the resulting vector in the vector operand **x**. If n is less than or equal to zero, the routine returns immediately. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler. For the routine TRMV_I, if ldt is less than one or ldt is less than n, an error flag is set and passed to the error handler. For the routine TBMV_I, if ldt is less than k plus one, an error flag is set and passed to the error handler.
The operator argument uplo specifies whether the matrix operand is upper or lower triangular. The operator argument diag specifies whether or not the matrix operand has unit diagonal entries.

- Fortran 95 binding:

```
Triangular:
      SUBROUTINE trmv_i( t, x [, uplo] [, transt] [, diag] [, alpha] )
Triangular Band:
      SUBROUTINE tbmv_i( t, x [, uplo] [, transt] [, diag] [, alpha] )
Triangular Packed:
      SUBROUTINE tpmv_i( tp, x [, uplo] [, transt] [, diag] [, alpha] )
all:
        TYPE(INTERVAL) (<wp>), INTENT(IN) :: t(:,:), tp(:)
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: x(:)
        TYPE(blas_uplo_type),  INTENT(IN), OPTIONAL :: uplo
        TYPE(blas_trans_type), INTENT(IN), OPTIONAL :: transt
        TYPE(blas_diag_type),  INTENT(IN), OPTIONAL :: diag
        TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha
      where
        x has shape (n)
       TR  t has shape (n,n)
       TB  t has shape (k+1,n) where k = band width
       TP  tp has shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
Triangular:
      SUBROUTINE BLAS_xTRMV_I( UPLO, TRANS, DIAG, N, ALPHA, T, LDT, X,
     $                         INCX )
Triangular Band:
      SUBROUTINE BLAS_xTBMV_I( UPLO, TRANS, DIAG, N, K, ALPHA, T, LDT, X,
     $                         INCX )
Triangular Packed:
      SUBROUTINE BLAS_xTPMV_I( UPLO, TRANS, DIAG, N, ALPHA, TP, X, INCX )
all:
      INTEGER            DIAG, INCX, K, LDA, N, TRANS, UPLO
      <type>             ALPHA( 2 )
      <type>             T( 2, LDA, * ) or TP( 2, * ), X( 2, * )
```

- C binding:

```
Triangular:
void BLAS_xtrmv_i( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, enum blas_diag_type diag, int n,
                   <interval> alpha, const <interval_array> t, int ldt,
                   <interval_array> x, int incx );
Triangular Band:
void BLAS_xtbmv_i( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, enum blas_diag_type diag, int n,
                   <interval> alpha, const <interval_array> t, int ldt,
                   <interval_array> x, int incx );
Triangular Packed:
void BLAS_xtpmv_i( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, enum blas_diag_type diag, int n,
                   <interval> alpha, const <interval_array> tp,
                   <interval_array> x, int incx );
```

{TR,TB,TP}SV_I (Interval triangular solve with a vector) $\mathbf{x} \leftarrow \alpha \mathbf{T}^{-1}\mathbf{x}$, $\mathbf{x} \leftarrow \alpha \mathbf{T}^{-T}\mathbf{x}$

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

Advice to users and implementors: Checking for singularity, or near singularity is not specified for these triangular solvers. Users should perform such a test before calling the triangular solver if their applications require such a test.

- Fortran 95 binding:

```
Triangular:
      SUBROUTINE trsv_i( t, x [, uplo] [, transt] [, diag] [, alpha] )
Triangular Band:
      SUBROUTINE tbsv_i( t, x [, uplo] [, transt] [, diag] [, alpha] )
Triangular Packed:
      SUBROUTINE tpsv_i( tp, x [, uplo] [, transt] [, diag] [, alpha] )
all:
        TYPE(INTERVAL) (<wp>), INTENT(IN) :: t(:,:), tp(:)
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: x(:)
        TYPE(blas_uplo_type),  INTENT(IN), OPTIONAL :: uplo
        TYPE(blas_trans_type), INTENT(IN), OPTIONAL :: transt
        TYPE(blas_diag_type),  INTENT(IN), OPTIONAL :: diag
        TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha
      where
        x has shape (n)
       TR  t has shape (n,n)
```

```
        TB  t has shape (k+1,n) where k = band width
        TP  tp has shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
Triangular:
      SUBROUTINE BLAS_xTRSV_I( UPLO, TRANS, DIAG, N, ALPHA, T, LDT, X,
     $                         INCX )
Triangular Band:
      SUBROUTINE BLAS_xTBSV_I( UPLO, TRANS, DIAG, N, K, ALPHA, T, LDT,
     $                         X, INCX )
Triangular Packed:
      SUBROUTINE BLAS_xTPSV_I( UPLO, TRANS, DIAG, N, ALPHA, TP, X, INCX )
all:
      INTEGER            DIAG, INCX, K, LDT, N, TRANS, UPLO
      <type>             ALPHA( 2 )
      <type>             T( 2, LDA, * ) or TP( 2, * ), X( 2, * )
```

- C binding:

```
Triangular:
void BLAS_xtrsv_i( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, enum blas_diag_type diag, int n,
                   const <interval> alpha, const <interval_array> t, int ldt,
                   <interval_array> x, int incx );
Triangular Band:
void BLAS_xtbsv_i( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, enum blas_diag_type diag, int n,
                   int k, const <interval> alpha, const <interval_array> t,
                   int ldt, <interval_array> x, int incx );
Triangular Packed:
void BLAS_xtpsv_i( enum blas_order_type order, enum blas_uplo_type uplo,
                   enum blas_trans_type trans, enum blas_diag_type diag, int n,
                   const <interval> alpha, const <interval_array> tp,
                   <interval_array> x, int incx );
```

GER_I (Rank one update) $\mathbf{A} \leftarrow \alpha \mathbf{x}\mathbf{y}^T + \beta \mathbf{A}$

This routine performs the operation $\mathbf{A} \leftarrow \alpha \mathbf{x}\mathbf{y}^T + \beta \mathbf{A}$, where α and β are intervals, $\mathbf{x}$ and $\mathbf{y}$ are interval vectors, and $\mathbf{A}$ is an interval matrix. This routine returns $\mathbf{A}$ immediately if $\alpha = [0, 0]$ and $\beta = [1, 1]$. If m or n is less than or equal to zero, this function returns immediately. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if either incx or incy is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or lda is less than m, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
      SUBROUTINE ger_i( a, x, y [, alpha] [, beta] )
      TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: a(:,:)
      TYPE(INTERVAL) (<wp>), INTENT(IN)    :: x(:), y(:)
      TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
      x and y have shape (n)
      a has shape (n,n)
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xGER_I( M, N, ALPHA, X, INCX, Y, INCY, BETA, A, LDA )
      INTEGER            INCX, INCY, LDA, M, N
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             A( 2, LDA, * ), X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xger_i( int m, int n, <interval> alpha, const <interval_array> x,
                  int incx, const <interval_array> y, int incy, <interval> beta,
                  <interval_array> a, int lda );
```

{SY,SP}R_I (Symmetric rank one update) $\mathbf{A} \leftarrow \alpha \mathbf{x}\mathbf{x}^T + \beta \mathbf{A}$ with $\mathbf{A} = \mathbf{A}^T$

This routine performs the symmetric update $\mathbf{A} \leftarrow \alpha \mathbf{x}\mathbf{y}^T + \beta \mathbf{A}$, where α and β are intervals, $\mathbf{x}$ is an interval vector, and $\mathbf{A}$ is a symmetric interval matrix. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx less than zero is permitted. However, if incx is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or lda is less than n, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
Symmetric:
      SUBROUTINE syr_i( a, x [, uplo] [, alpha] [, beta] )
Symmetric Packed:
      SUBROUTINE spr_i( ap, x [, uplo] [, alpha] [, beta] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: a(:,:), ap(:)
      TYPE(INTERVAL) (<wp>), INTENT(IN)    :: x(:)
      TYPE(blas_uplo_type),  OPTIONAL :: uplo
      TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
    where
      x has shape (n)
     SY  a has shape (n,n)
     SP  ap has shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
Symmetric:
      SUBROUTINE BLAS_xSYR_I( UPLO, N, ALPHA, X, INCX, BETA, A, LDA )
Symmetric Packed:
      SUBROUTINE BLAS_xSPR_I( UPLO, N, ALPHA, X, INCX, BETA, AP )
all:
      INTEGER            INCX, LDA, N, UPLO
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             A( 2, LDA, * ) or AP( 2, * ), X( 2, * )
```

- C binding:

```
Symmetric:
void BLAS_xsyr_i( enum blas_order_type order, enum blas_uplo_type uplo, int n,
                  <interval> alpha, const <interval_array> x, int incx,
                  <interval> beta, <interval_array> a, int lda );
Symmetric Packed:
void BLAS_xspr_i( enum blas_order_type order, enum blas_uplo_type uplo, int n,
                  <interval> alpha, const <interval_array> x, int incx,
                  <interval> beta, <interval_array> ap );
```

Interval Matrix Operations

{GE,GB,SY,SB,SP,TR,TB,TP}_NORM_I (Interval matrix norms)

$$r \leftarrow ||\mathbf{A}||_1,\ ||\mathbf{A}||_F,\ ||\mathbf{A}||_\infty,\ \text{ or } ||\mathbf{A}||_{\max}$$

These routines compute the one-norm, Frobenius-norm, infinity-norm, or max-norm of a general interval matrix **A** depending on the value passed as the `norm` operator argument. This routine returns immediately with the output scalar r set to zero if `m` (for nonsymmetric matrices) or `n` is less than or equal to zero. For the routine GE_NORM_I, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine GB_NORM_I, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler. For the routines SY_NORM_I and TR_NORM_I, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routines SB_NORM_I and TB_NORM_I, if lda is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
      REAL (<wp>)  FUNCTION ge_norm_i( a [, norm] )
General Band:
      REAL (<wp>)  FUNCTION gb_norm_i( a, m, kl [, norm] )
Symmetric:
      REAL (<wp>)  FUNCTION sy_norm_i( a [, norm] [, uplo] )
Symmetric Band:
      REAL (<wp>)  FUNCTION sb_norm_i( a [, norm] [, uplo] )
Symmetric Packed:
      REAL (<wp>)  FUNCTION sp_norm_i( ap [, norm] [, uplo] )
```

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

- Fortran 77 binding:

```
General:
      <type>  FUNCTION BLAS_xGE_NORM_I( NORM, M, N, A, LDA )
General Band:
      <type>  FUNCTION BLAS_xGB_NORM_I( NORM, M, N, KL, KU, A, LDA )
Symmetric:
      <type>  FUNCTION BLAS_xSY_NORM_I( NORM, UPLO, N, A, LDA )
Symmetric Band:
      <type>  FUNCTION BLAS_xSB_NORM_I( NORM, UPLO, N, K, A, LDA )
Symmetric Packed:
      <type>  FUNCTION BLAS_xSP_NORM_I( NORM, UPLO, N, AP )
Triangular:
      <type>  FUNCTION BLAS_xTR_NORM_I( NORM, UPLO, DIAG, N, A, LDA )
Triangular Band:
      <type>  FUNCTION BLAS_xTB_NORM_I( NORM, UPLO, DIAG, N, K, A, LDA )
Triangular Packed:
      <type>  FUNCTION BLAS_xTP_NORM_I( NORM, UPLO, DIAG, N, AP )
all:
      INTEGER            DIAG, K, KL, KU, LDA, M, N, NORM, UPLO
      <type>             A( 2, LDA, * ) or AP( 2, * )
```

- C binding:

```
General:
void BLAS_xge_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                      int m, int n, const <interval_array> a, int lda,
```

```
                            SCALAR_INOUT r );
General Band:
void BLAS_xgb_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                      int m, int n, int kl, int ku, const <interval_array> a,
                      int lda, SCALAR_INOUT r );
Symmetric:
void BLAS_xsy_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                      enum blas_uplo_type uplo, int n,
                      const <interval_array> a, int lda, SCALAR_INOUT  r );
Symmetric Band:
void BLAS_xsb_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                      enum blas_uplo_type uplo, int n, int k,
                      const <interval_array> a, int lda, SCALAR_INOUT r );
Symmetric Packed:
void BLAS_xsp_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                      enum blas_uplo_type uplo, int n,
                      const <interval_array> ap, SCALAR_INOUT r );
Triangular:
void BLAS_xtr_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                      enum blas_uplo_type uplo, enum blas_diag_type diag,
                      int n, const <interval_array> a, int lda,
                      SCALAR_INOUT r );
Triangular Band:
void BLAS_xtb_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                      enum blas_uplo_type uplo, enum blas_diag_type diag,
                      int n, int k, const <interval_array> a, int lda,
                      SCALAR_INOUT  r );
Triangular Packed:
void BLAS_xtp_norm_i( enum blas_order_type order, enum blas_norm_type norm,
                      enum blas_uplo_type uplo, enum blas_diag_type diag,
                      int n, const <interval_array> ap, SCALAR_INOUT  r );
```

Advice to implementors: In finite precision floating point arithmetic, an upper bound, preferably the least machine representable upper bound, for the mathematical value should be returned for the norms.

{GE,GB}_DIAG_SCALE_I (Diagonal scaling an interval matrix) $\mathbf{A} \leftarrow \mathbf{DA}, \mathbf{AD}$ with $\mathbf{D}$ diagonal

These routines scale a general (or banded) interval matrix $\mathbf{A}$ on the left side or the right side by a diagonal interval matrix $\mathbf{D}$. This routine returns immediately if m or n is less than or equal to zero. As described in section 2.5.3, the value incd less than zero is permitted. However, if incd is equal to zero, an error flag is set and passed to the error handler. For the routine GE_DIAG_SCALE_I, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine GB_DIAG_SCALE_I, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

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

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_DIAG_SCALE_I( SIDE, M, N, D, INCD, A, LDA )
General Band:
      SUBROUTINE BLAS_xGB_DIAG_SCALE_I( SIDE, M, N, KL, KU, D, INCD, A,
     $                                  LDA )
all:
      INTEGER            INCD, KL, KU, LDA, M, N, SIDE
      <type>             A( 2, LDA, * ), D( 2, * )
```

- C binding:

```
General:
void BLAS_xge_diag_scale_i( enum blas_order_type order,
                            enum blas_side_type side, int m, int n,
                            const <interval_array> d, int incd,
                            <interval_array> a, int lda );
General Band:
void BLAS_xgb_diag_scale_i( enum blas_order_type order,
                            enum blas_side_type side, int m, int n, int kl,
                            int ku, const <interval_array> d, int incd,
                            <interval_array> a, int lda );
```

{GE,GB}_LRSCALE_I (Two-sided diagonal scaling) $\mathbf{A} \leftarrow \mathbf{D}_L \mathbf{A} \mathbf{D}_R$ with $\mathbf{D}_L, \mathbf{D}_R$ diagonal

These routines scale a general (or banded) interval matrix $\mathbf{A}$ on the left side by an interval diagonal matrix $\mathbf{D}_L$ and on the right side by an interval diagonal matrix $\mathbf{D}_R$. This routine returns immediately if m or n is less than or equal to zero. As described in section 2.5.3, the value incdl or incdu less than zero is permitted. However, if either incdl or incdu is equal to zero, an error flag is set and passed to the error handler. For the routine GE_LRSCALE_I, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine GB_LRSCALE_I, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

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

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_LRSCALE_I( M, N, DL, INCDL, DR, INCDR, A, LDA )
General Band:
      SUBROUTINE BLAS_xGB_LRSCALE_I( M, N, KL, KU, DL, INCDL, DR, INCDR, A,
     $                               LDA )
all:
      INTEGER            INCDL, INCDR, KL, KU, LDA, M, N
      <type>             A( 2, LDA, * ), DL( 2, * ), DR( 2, * )
```

- C binding:

```
General:
void BLAS_xge_lrscale_i( enum blas_order_type order, int m, int n,
                         const <interval_array> dl, int incdl,
                         const <interval_array> dr, int incdr,
                         <interval_array> a, int lda );
General Band:
void BLAS_xgb_lrscale_i( enum blas_order_type order, int m, int n,
                         int kl, int ku, const <interval_array> dl,
                         int incdl, const <interval_array> dr, int incdr,
                         <interval_array> a, int lda );
```

{SY,SB,SP}_LRSCALE_I (Two-sided diagonal scaling) $\mathbf{A} \leftarrow \mathbf{DAD}$ with $\mathbf{A} = \mathbf{A}^T$.

These routines perform a two-sided scaling on a symmetric (or symmetric banded or symmetric packed) interval matrix **A** by an interval diagonal matrix **D**. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incd less than zero is permitted. However, if incd is equal to zero, an error flag is set and passed to the error handler. For the routines SY_LRSCALE_I and SP_LRSCALE_I, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine SB_LRSCALE_I, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

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

- Fortran 77 binding:

```
Symmetric:
      SUBROUTINE BLAS_xSY_LRSCALE_I( UPLO, N, D, INCD, A, LDA )
Symmetric Band:
      SUBROUTINE BLAS_xSB_LRSCALE_I( UPLO, N, K, D, INCD, A, LDA )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_LRSCALE_I( UPLO, N, D, INCD, AP )
all:
      INTEGER            INCD, K, LDA, N, UPLO
      <type>             A( 2, LDA, * ) or AP( 2, * ), D( 2, * )
```

- C binding:

```
Symmetric:
void BLAS_xsy_lrscale_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         int n, const <interval_array> d, int incd,
                         <interval_array> a, int lda );
Symmetric Band:
void BLAS_xsb_lrscale_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         int n, int k, const <interval_array> d, int incd,
```

```
                             <interval_array> a, int lda );
Symmetric Packed:
void BLAS_xsp_lrscale_i( enum blas_order_type order, enum blas_uplo_type uplo,
                             int n, const <interval_array> d, int incd,
                             <interval_array> ap );
```

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

These routines scale an interval matrix $\mathbf{A}$ (or its transpose) and scale an interval matrix $\mathbf{B}$ and accumulate the result in the interval matrix $\mathbf{B}$. Matrices $\mathbf{A}$ (or $\mathbf{A}^T$) and $\mathbf{B}$ have the same storage format. This routine returns immediately if m (for nonsymmetric matrices) or n or k (for symmetric band matrices) is less than or equal to zero. For the routine GE_ACC_I, if lda is less than one or lda is less than m, an error flag is set and passed to the error handler. For the routine SY_ACC_I, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine SB_ACC_I, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_acc_i( a, b [, transa] [, alpha] [, beta] )
Symmetric:
      SUBROUTINE sy_acc_i( a, b [, uplo], [, transa] [, alpha] [, beta] )
Symmetric Band:
      SUBROUTINE sb_acc_i( a, b [, uplo], [, transa] [, alpha] [, beta] )
Symmetric Packed:
      SUBROUTINE sp_acc_i( ap, bp [, uplo], [, transa] [, alpha] [, beta] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: b(:,:) | bp(:)
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:)
      TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      TYPE(blas_trans_type), INTENT(IN), OPTIONAL :: transa
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
    where
     GE  a and b have shape (m,n) if transa = blas_no_trans (the default)
         a has shape (n,m) and b has shape (m,n) if transa /= blas_no_trans
     SY  a and b have shape (n,n)
     SB  a and b have shape (p+1,n) (p = band width)
     SP  ap and bp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_ACC_I( TRANS, M, N, ALPHA, A, LDA, BETA, B, LDB )
Symmetric:
      SUBROUTINE BLAS_xSY_ACC_I( UPLO, TRANS, N, ALPHA, A, LDA, BETA, B,
```

```
     $                           LDB )
Symmetric Band:
      SUBROUTINE BLAS_xSB_ACC_I( UPLO, TRANS, N, K, ALPHA, A, LDA, BETA,
     $                           B, LDB )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_ACC_I( UPLO, TRANS, N, ALPHA, AP, BETA, BP )
all:
      INTEGER            K, LDA, LDB, M, N, TRANS, UPLO
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDB, * )
     $                   or BP( 2, * )
```

- C binding:

```
General:
void BLAS_xge_acc_i( enum blas_order_type order, enum blas_trans_type trans,
                     int m, int n, <interval> alpha, const <interval_array> a,
                     int lda, <interval> beta, <interval_array> b, int ldb );
Symmetric:
void BLAS_xsy_acc_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_trans_type trans, int n, <interval> alpha,
                     const <interval_array> a, int lda, <interval> beta,
                     <interval_array> b, int ldb );
Symmetric Band:
void BLAS_xsb_acc_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_trans_type trans, int n, int k, <interval> alpha,
                     const <interval_array> a, int lda, <interval> beta,
                     <interval_array> b, int ldb );
Symmetric Packed:
void BLAS_xsp_acc_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_trans_type trans, int n, <interval> alpha,
                     const <interval_array> ap, <interval> beta,
                     <interval_array> bp );
```

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

These routines scale interval matrices $\mathbf{A}$ and $\mathbf{B}$ and accumulate the result in the matrix $\mathbf{B}$. Matrices A and B have the same storage format. This routine returns immediately if m or kl or ku (for general band matrices) or n or k (for triangular band matrices) is less than or equal to zero. For the routine GB_ACC_I, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler. For the routines TR_ACC_I and TP_ACC_I, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routine TB_ACC_I, if lda is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General Band:
```

```
      SUBROUTINE gb_acc_i( a, m, kl, b [, alpha] [, beta] )
Triangular:
      SUBROUTINE tr_acc_i( a, b [, uplo], [, diag] [, alpha] [, beta] )
Triangular Band:
      SUBROUTINE tb_acc_i( a, b [, uplo], [, diag] [, alpha] [, beta] )
Triangular Packed:
      SUBROUTINE tp_acc_i( ap, bp [, uplo], [, diag] [, alpha] [, beta] )
all:

      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:)
      INTEGER, INTENT(IN) :: m, kl
      TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: b(:,:) | bp(:)
      TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
      a and b have shape (l,n) for general banded matrix (l > kl)
      a and b have shape (n,n) for triangular matrix
      a and b have shape (p+1,n) for triangular banded matrix
      ap and bp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General Band:
      SUBROUTINE BLAS_xGB_ACC_I( M, N, KL, KU, ALPHA, A, LDA, BETA, B,
     $                           LDB )
Triangular:
      SUBROUTINE BLAS_xTR_ACC_I( UPLO, DIAG, N, ALPHA, A, LDA, BETA, B,
     $                           LDB )
Triangular Band:
      SUBROUTINE BLAS_xTB_ACC_I( UPLO, DIAG, N, K, ALPHA, A, LDA, BETA,
     $                           B, LDB )
Triangular Packed:
      SUBROUTINE BLAS_xTP_ACC_I( UPLO, DIAG, N, ALPHA, AP, BETA, BP )
all:
      INTEGER           DIAG, K, KL, KU, LDA, LDB, M, N, UPLO
      <type>            ALPHA( 2 ), BETA( 2 )
      <type>            A( 2, LDA, * ) or AP( 2, * ), B( 2, LDB, * )
     $                  or BP( 2, * )
```

- C binding:

```
General Band:
void BLAS_xgb_acc_i( enum blas_order_type order, int m, int n, int kl, int ku,
                     <interval> alpha, <interval_array> a, int lda,
                     <interval> beta, <interval_array> b, int ldb );
Triangular:
```

```
void BLAS_xtr_acc_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_diag_type diag, int n, <interval> alpha,
                     <interval_array> a, int lda, <interval> beta,
                     <interval_array> b, int ldb );
Triangular Band:
void BLAS_xtb_acc_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_diag_type diag, int n, int k, <interval> alpha,
                     <interval_array> a, int lda, <interval> beta,
                     <interval_array> b, int ldb );
Triangular Packed:
void BLAS_xtp_acc_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_diag_type diag, int n, <interval> alpha,
                     <interval_array> ap, <interval> beta,
                     <interval_array> bp );
```

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

These routines scale two interval matrices **A** and **B** and store the sum in the matrix **C**. Matrices A, B, and C have the same storage format. This routine returns immediately if m or kl or ku (for general band matrices) or n or k (for symmetric or triangular band matrices) is less than or equal to zero. For the routine GE_ADD_I, if lda is less than one or less than m, an error flag is set and passed to the error handler. For the routine GB_ADD_I, if lda is less than kl plus ku plus one, an error flag is set and passed to the error handler. For the routines SY_ADD_I, TR_ADD_I, SP_ADD_I, and TP_ADD_I, if lda is less than one or lda is less than n, an error flag is set and passed to the error handler. For the routines SB_ADD_I and TB_ADD_I, if lda is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_add_i( a, b, c [, alpha] [, beta] )
General Band:
      SUBROUTINE gb_add_i( a, m, kl, b, c [, alpha] [, beta] )
Symmetric:
      SUBROUTINE sy_add_i( a, b, c [, uplo], [, alpha] [, beta] )
Symmetric Band:
      SUBROUTINE sb_add_i( a, b, c [, uplo], [, alpha] [, beta] )
Symmetric Packed:
      SUBROUTINE sp_add_i( ap, bp, cp [, uplo], [, alpha] [, beta] )
Triangular:
      SUBROUTINE tr_add_i( a, b, c [, uplo], [, diag] [, alpha] [, beta] )
Triangular Band:
      SUBROUTINE tb_add_i( a, b, c [, uplo], [, diag] [, alpha] [, beta] )
Triangular Packed:
      SUBROUTINE tp_add_i( ap, bp, cp [, uplo], [, diag] [, alpha] [, beta] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:), b(:,:) | bp(:)
```

```
      INTEGER, INTENT(IN) :: m, kl
      TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: c(:,:) | cp(:)
      TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
     assuming A, B and C all the same (general, banded or packed) with
     the same size.
      a, b and c have shape (m,n) for general matrix
                            (l,n) for general banded matrix (l > kl)
                            (n,n) for symmetric or triangular
                            (k+1,n) for symmetric banded or triangular
                                    banded (k=band width)
      ap, bp and cp have shape (n*(n+1)/2).
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_ADD_I( M, N, ALPHA, A, LDA, BETA, B, LDB, C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_ADD_I( M, N, KL, KU, ALPHA, A, LDA, BETA, B, LDB,
     $                           C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_ADD_I( UPLO, N, ALPHA, A, LDA, BETA, B, LDB, C,
     $                           LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_ADD_I( UPLO, N, K, ALPHA, A, LDA, BETA, B, LDB,
     $                           C, LDC )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_ADD_I( UPLO, N, ALPHA, AP, BETA, BP, CP )
Triangular:
      SUBROUTINE BLAS_xTR_ADD_I( UPLO, DIAG, N, ALPHA, A, LDA, BETA, B,
     $                           LDB, C, LDC )
Triangular Band:
      SUBROUTINE BLAS_xTB_ADD_I( UPLO, DIAG, N, K, ALPHA, A, LDA, BETA, B,
     $                           LDB, C, LDC )
Triangular Packed:
      SUBROUTINE BLAS_xTP_ADD_I( UPLO, DIAG, N, ALPHA, AP, BETA, BP, CP )
all:
      INTEGER            DIAG, K, KL, KU, LDA, LDB, M, N, TRANS, UPLO
      <type>             ALPHA( 2 ), BETA( 2 )
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDB, * )
     $                   or BP( 2, * ), C( 2, LDC, * ) or CP( 2, * )
```

- C binding:

```
General:
```

```
void BLAS_xge_add_i( enum blas_order_type order, int m, int n, <interval> alpha,
                     const <interval_array> a, int lda, <interval> beta,
                     const <interval_array> b, int ldb, <interval_array> c,
                     int ldc );
General Band:
void BLAS_xgb_add_i( enum blas_order_type order, int m, int n, int kl, int ku,
                     <interval> alpha, const <interval_array> a, int lda,
                     <interval> beta, const <interval_array> b, int ldb,
                     <interval_array> c, int ldc );
Symmetric:
void BLAS_xsy_add_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, <interval> alpha, const <interval_array> a,
                     int lda, <interval> beta, const <interval_array> b,
                     int ldb, <interval_array> c, int ldc );
Symmetric Band:
void BLAS_xsb_add_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, int k, <interval> alpha, const <interval_array> a,
                     int lda, <interval> beta, const <interval_array> b,
                     int ldb, <interval_array> c, int ldc );
Symmetric Packed:
void BLAS_xsp_add_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, <interval> alpha, const <interval_array> ap,
                     <interval> beta, const <interval_array> bp,
                     <interval_array> cp );
Triangular:
void BLAS_xtr_add_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_diag_type diag, int n, <interval> alpha,
                     const <interval_array> a, int lda, <interval> beta,
                     const <interval_array> b, int ldb, <interval_array> c,
                     int ldc );
Triangular Band:
void BLAS_xtb_add_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, enum blas_diag_type diag, int k, <interval> alpha,
                     const <interval_array> a, int lda, <interval> beta,
                     const <interval_array> b, int ldb, <interval_array> c,
                     int ldc );
Triangular Packed:
void BLAS_xtp_add_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, enum blas_diag_type diag, <interval> alpha,
                     const <interval_array> ap, <interval> beta,
                     const <interval_array> bp, <interval_array> cp );
```

Interval Matrix-Matrix Operations

In the following specifications, $\mathrm{op}(X)$ denotes X or X^T where X is a matrix.

GEMM_I (General interval matrix matrix product) $\mathbf{C} \leftarrow \alpha\ \mathrm{op}(\mathbf{A})\ \mathrm{op}(\mathbf{B}) + \beta\mathbf{C}$.

This routine performs a general interval matrix matrix multiply $\mathbf{C} \leftarrow \alpha \, \mathrm{op}(\mathbf{A}) \, \mathrm{op}(\mathbf{B}) + \beta\mathbf{C}$, where α and β are intervals, and $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are general interval matrices. This routine returns immediately if m or n or k is less than or equal to zero. If lda is less than one or less than m, or if ldb is less than one or less than k, or if ldc is less than one or less than m, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
    SUBROUTINE gemm_i( a, b, c [, transa] [, transb] [, alpha] [, beta] )
    TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), b(:,:)
    TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: c(:,:)
    TYPE(blas_trans_type), INTENT(IN), OPTIONAL :: transa, transb
    TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
  where
    c has shape (m,n)
    a has shape (m,k) if transa = blas_no_trans (the default)
                (k,m) if transa /= blas_no_trans
    b has shape (k,n) if transb = blas_no_trans (the default)
                (n,k) if transb /= blas_no_trans
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xGEMM_I( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA,
   $                        B, LDB, BETA, C, LDC )
    INTEGER           K, LDA, LDB, LDC, M, N, TRANSA, TRANSB
    <type>            ALPHA( 2 ), BETA( 2 )
    <type>            A( 2, LDA, * ), B( 2, LDB, * ),
   $                  C( 2, LDC, * )
```

- C binding:

```
void BLAS_xgemm_i( enum blas_order_type order, enum blas_trans_type transa,
                   enum blas_trans_type transb, int m, int n, int k,
                   <interval> alpha,  const <interval_array> a, int lda,
                   const <interval_array> b, int ldb, <interval> beta,
                   <interval_array> c, int ldc );
```

SYMM_I (Symmetric interval matrix matrix product) $\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta\mathbf{C}$ or $\mathbf{C} \leftarrow \alpha\mathbf{BA} + \beta\mathbf{C}$.

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

- Fortran 95 binding:

```
        SUBROUTINE symm_i( a, b, c [, side] [, uplo] [, alpha] [, beta] )
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: c(:,:)
        TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), b(:,:)
        TYPE(blas_side_type), INTENT(IN), OPTIONAL :: side
        TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
        TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      where
        c has shape (m,n), b same shape as c
         SY  a has shape (m,m) if side = blas_left_side (the default)
             a has shape (n,n) if side /= blas_left_side
```

- Fortran 77 binding:

```
        SUBROUTINE BLAS_xSYMM_I( SIDE, UPLO, M, N, ALPHA, A, LDA, B,
       $                         LDB, BETA, C, LDC )
        INTEGER            LDA, LDB, LDC, M, N, SIDE, UPLO
        <type>             ALPHA( 2 ), BETA( 2 )
        <type>             A( 2, LDA, * ), B( 2, LDB, * ),
       $                   C( 2, LDC, * )
```

- C binding:

```
void BLAS_xsymm_i( enum blas_order_type order, enum blas_side_type side,
                   enum blas_uplo_type uplo, int m, int n, <interval> alpha,
                   const <interval_array> a, int lda, const <interval_array> b,
                   int ldb, <interval> beta, <interval_array> c, int ldc );
```

TRMM_I (Triangular interval matrix matrix product)

$$\mathbf{B} \leftarrow \alpha\mathbf{TB}, \mathbf{B} \leftarrow \alpha\mathbf{T}^T\mathbf{B}, \mathbf{B} \leftarrow \alpha\mathbf{BT}, \text{ or } \mathbf{B} \leftarrow \alpha\mathbf{BT}^T$$

These routines perform one of the interval matrix operations $\mathbf{B} \leftarrow \alpha\mathbf{TB}, \mathbf{B} \leftarrow \alpha\mathbf{T}^T\mathbf{B}$, $\mathbf{B} \leftarrow \alpha\mathbf{BT}$, or $\mathbf{B} \leftarrow \alpha\mathbf{BT}^T$, where α is an interval, $\mathbf{T}$ is a unit, or non-unit, upper or lower triangular interval matrix, and $\mathbf{B}$ is a general interval matrix. This routine returns immediately if m or n is less than or equal to zero. For side equal to blas_left_side, and if ldt is less than one or less than m, or if ldb is less than one or less than m, an error flag is set and passed to the error handler. For side equal to blas_right_side, and if ldt is less than one or less than n, or if ldb is less than one or less than m, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
        SUBROUTINE trmm_i( t, b [, side] [, uplo] [, transt] [, diag] &
                           [, alpha] )
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: b(:,:)
        TYPE(INTERVAL) (<wp>), INTENT(IN) :: t(:,:)
```

```
          TYPE(blas_side_type),  INTENT(IN), OPTIONAL :: side
          TYPE(blas_trans_type), INTENT(IN), OPTIONAL :: transt
          TYPE(blas_diag_type),  INTENT(IN), OPTIONAL :: diag
          TYPE(blas_uplo_type),  INTENT(IN), OPTIONAL :: uplo
          TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
        where
          b has shape (m,n)
           TR  t has shape (m,m) if side = blas_left_side (the default)
               t has shape (n,n) if side /= blas_left_side
```

- Fortran 77 binding:

```
          SUBROUTINE BLAS_xTRMM_I( SIDE, UPLO, TRANST, DIAG, M, N, ALPHA,
         $                         T, LDT, B, LDB )
          INTEGER            DIAG, LDA, LDB, M, N, SIDE, TRANST, UPLO
          <type>             ALPHA( 2 )
          <type>             T( 2, LDA, * ), B( 2, LDB, * )
```

- C binding:

```
void BLAS_xtrmm_i( enum blas_order_type order, enum blas_side_type side,
                   enum blas_uplo_type uplo, enum blas_trans_type transt,
                   enum blas_diag_type diag, int m, int n, <interval> alpha,
                   const <interval_array> t, int ldt, <interval_array> b,
                   int ldb );
```

TRSM_I (Interval triangular solve)

$$\mathbf{B} \leftarrow \alpha\mathbf{T}^{-1}\mathbf{B},\ \mathbf{B} \leftarrow \alpha(\mathbf{T}^{-1})^T\mathbf{B},\ \mathbf{B} \leftarrow \alpha\mathbf{B}\mathbf{T}^{-1} \text{ or } \mathbf{B} \leftarrow \alpha\mathbf{B}(\mathbf{T}^{-1})^T$$

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

- Fortran 95 binding:

```
          SUBROUTINE trsm_i( t, b [, side] [, uplo] [, transt] [, diag] [, alpha] )
          TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: b(:,:)
          TYPE(INTERVAL) (<wp>), INTENT(IN) :: t(:,:)
          TYPE(blas_side_type),  INTENT(IN), OPTIONAL :: side
          TYPE(blas_trans_type), INTENT(IN), OPTIONAL :: transt
          TYPE(blas_diag_type),  INTENT(IN), OPTIONAL :: diag
          TYPE(blas_uplo_type),  INTENT(IN), OPTIONAL :: uplo
```

```
        TYPE(INTERVAL) (<wp>), INTENT(IN), OPTIONAL :: alpha, beta
      where
        b has shape (m,n)
         TR  t has shape (m,m) if side = blas_left_side (the default)
             t has shape (n,n) if side /= blas_left_side
```

- Fortran 77 binding:

```
        SUBROUTINE BLAS_xTRSM_I( SIDE, UPLO, TRANST, DIAG, M, N, ALPHA,
       $                         ALPHA, T, LDT, B, LDB )
        INTEGER            DIAG, LDB, LDT, M, N, SIDE, TRANST, UPLO
        <type>             ALPHA( 2 )
        <type>             T( 2, LDA, * ), B( 2, LDB, * )
```

- C binding:

```
void BLAS_xtrsm_i( enum blas_order_type order, enum blas_side_type side,
                   enum blas_uplo_type uplo, enum blas_trans_type transt,
                   enum blas_diag_type diag, int m, int n, <interval> alpha,
                   const <interval_array> t, int ldt, <interval_array> b,
                   int ldb );
```

Data Movement with Interval Matrices

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

This routine copies an interval matrix (or its transpose) **A** and stores the result in an interval matrix **B**. Matrices A (or $\mathbf{A}^T$) and B have the same storage format. This routine returns immediately if m (for nonsymmetric matrices), n, k (for symmetric band matrices), or kl or ku (for general band matrices), is less than or equal to zero. For the routine GE_COPY_I, if trans equal to blas_no_trans, and if lda is less than one or less than m, or if ldb is less than one or less than m, an error flag is set and passed to the error handler. For the routine GE_COPY_I, if trans equal to blas_trans, and if lda is less than one or less than m, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For the routine GB_COPY_I, if lda is less than kl plus ku plus one, or if ldb is less than kl plus ku plus one, an error flag is set and passed to the error handler. For the routines SY_COPY_I and TR_COPY_I, if lda is less than one or less than n, or if ldb is less than one or less than n, an error flag is set and passed to the error handler. For the routines SB_COPY_I and TB_COPY_I, if lda is less than k plus one, or if ldb is less than k plus one, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_copy_i( a, b [, transa] )
General Band:
      SUBROUTINE gb_copy_i( a, kl, b [, transa] )
Symmetric:
```

```
            SUBROUTINE sy_copy_i( a, b [, uplo] )
      Symmetric Band:
            SUBROUTINE sb_copy_i( a, b [, uplo] )
      Symmetric Packed:
            SUBROUTINE sp_copy_i( ap, bp [, uplo] )
      Triangular:
            SUBROUTINE tr_copy_i( a, b [, uplo], [, trans] [, diag] )
      Triangular Band:
            SUBROUTINE tb_copy_i( a, b [, uplo], [, trans] [, diag] )
      Triangular Packed:
            SUBROUTINE tp_copy_i( ap, bp [, uplo], [, trans] [, diag] )
      all:
            TYPE(INTERVAL) (<wp>), INTENT(OUT) :: b(:,:) | bp(:)
            TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:)
            INTEGER, INTENT(IN) :: kl
            TYPE(blas_trans_type),INTENT(IN), OPTIONAL :: trans
            TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
            TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
          where
           If trans = blas_no_trans (the default)
              a, b have shape (m,n) for general matrix
                              (l,n) for general banded matrix (l > kl)
                              (n,n) for symmetric or triangular
                              (k+1,n) for symmetric banded or triangular
                                      banded (k=band width)
              ap and bp have shape (n*(n+1)/2).

           If trans \= blas_no_trans
              a has shape (m,n) and b has shape (n,m) for general matrix
                          (l,n) and b has shape (l,m) for general banded matrix (l>kl)
              a and b have shape (n,n) for symmetric or triangular
                                 (k+1,n) for symmetric banded or triangular
                                         banded (k=band width)
              ap and bp have shape (n*(n+1)/2).
```

- Fortran 77 binding:

```
      General:
            SUBROUTINE BLAS_xGE_COPY_I( TRANS, M, N, A, LDA, B, LDB )
      General Band:
            SUBROUTINE BLAS_xGB_COPY_I( TRANS, M, N, KL, KU, A, LDA, B, LDB )
      Symmetric:
            SUBROUTINE BLAS_xSY_COPY_I( UPLO, N, A, LDA, B, LDB )
      Symmetric Band:
            SUBROUTINE BLAS_xSB_COPY_I( UPLO, N, K, A, LDA, B, LDB )
      Symmetric Packed:
            SUBROUTINE BLAS_xSP_COPY_I( UPLO, N, AP, BP )
      Triangular:
```

```
      SUBROUTINE BLAS_xTR_COPY_I( UPLO, TRANS, DIAG, N, A, LDA, B, LDB )
Triangular Band:
      SUBROUTINE BLAS_xTB_COPY_I( UPLO, TRANS, DIAG, N, K, A, LDA, B, LDB )
Triangular Packed:
      SUBROUTINE BLAS_xTP_COPY_I( UPLO, TRANS, DIAG, N, AP, BP )
all:
      INTEGER            DIAG, LDA, LDB, N, K, KL, KU, TRANS, UPLO
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDB, * )
     $                   or BP( 2, * )
```

- C binding:

```
General:
void BLAS_xge_copy_i( enum blas_order_type order, enum blas_trans_type transa,
                      int m, int n, const <interval_array> a, int lda,
                      <interval_array> b, int ldb );
General Band:
void BLAS_xgb_copy_i( enum blas_order_type order, enum blas_trans_type transa,
                      int m, int n, int kl, int ku, const <interval_array> a,
                      int lda, <interval_array> b, int ldb );
Symmetric:
void BLAS_xsy_copy_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, const <interval_array> a, int lda,
                      <interval_array> b, int ldb );
Symmetric Band:
void BLAS_xsb_copy_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, int k, const <interval_array> a, int lda,
                      <interval_array> b, int ldb );
Symmetric Packed:
void BLAS_xsp_copy_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, const <interval_array> ap, <interval_array> bp );
Triangular:
void BLAS_xtr_copy_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> a, int lda,
                      <interval_array> b, int ldb );
Triangular Band:
void BLAS_xtb_copy_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, int k, const <interval_array> a, int lda,
                      <interval_array> b, int ldb );
Triangular Packed:
void BLAS_xtp_copy_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> ap, <interval_array> bp );
```

GE_TRANS_I (Matrix transposition) $\mathbf{A} \leftarrow \mathbf{A}^T$

This routine performs the transposition of a square interval matrix $\mathbf{A}$ and overwrites the matrix $\mathbf{A}$. This routine returns immediately if n is less than or equal to zero. If lda is less than one or less than n, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
        SUBROUTINE  ge_trans_i( a )
        TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: a(:,:)
      where
        a has shape (n,n)
```

- Fortran 77 binding:

```
        SUBROUTINE BLAS_xGE_TRANS_I( N, A, LDA )
        INTEGER            LDA, N
        <type>             A( 2, LDA, * )
```

- C binding:

```
void BLAS_xge_trans_i( int n, <interval_array> a, int lda );
```

GE_PERMUTE_I (Permute an interval matrix) $\mathbf{A} \leftarrow P\mathbf{A}$ or $\mathbf{A} \leftarrow \mathbf{A}P$

This routine permutes the rows or columns of an interval matrix $\mathbf{A}$ by the permutation matrix P. This routine returns immediately if m or n is less than or equal to zero. As described in section 2.5.3, the value incp less than zero is permitted. However, if incp is equal to zero, an error flag is set and passed to the error handler. If lda is less than one or less than m, an error flag is set and passed to the error handler.

- Fortran 95 binding:

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

- Fortran 77 binding:

```
        SUBROUTINE BLAS_xGE_PERMUTE_I( SIDE, M, N, P, INCP, A, LDA )
        INTEGER            INCP, LDA, M, N, SIDE
        INTEGER            P( * )
        <type>             A( 2, LDA, * )
```

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

- C binding:

```
void BLAS_xge_permute_i( enum blas_order_type order, enum blas_side_type side,
                         int m, int n, const int *p, int incp,
                         <interval_array> a, int lda );
```

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

Set Operations Involving Interval Vectors

ENCV_I (Checks if an interval vector is enclosed in another interval vector) True if $\mathbf{x} \subseteq \mathbf{y}$

This routine checks if an interval vector $\mathbf{x}$ is enclosed in another interval vector $\mathbf{y}$. We say that an interval vector $\mathbf{x}$ is enclosed in $\mathbf{y}$, denoted as $\mathbf{x} \subseteq \mathbf{y}$, if and only if $\underline{y_i} \leq \underline{x_i} \leq \overline{x_i} \leq \overline{y_i} \forall i$. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
     LOGICAL FUNCTION encv_i(x, y)
     TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:), y(:)
   where
     x and y have shape (n)
```

- Fortran 77 binding:

```
    LOGICAL FUNCTION BLAS_xENCV_I( N, X, INCX, Y, INCY )
    INTEGER            N, INCX, INCY
    <type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
int BLAS_xencv_i( int n, const <interval_array> x, int incx,
                  const <interval_array> y, int incy );
```

INTERIORV_I (If an interval vector is in the interior of another interval vector) True if $\mathbf{x} \subset \mathbf{y}$

This routine checks if an interval vector $\mathbf{x}$ is enclosed in the interior of another interval vector $\mathbf{y}$. We say that an interval vector $\mathbf{x}$ is enclosed in the interior of $\mathbf{y}$, denoted as $\mathbf{x} \subset \mathbf{y}$, if and only if $\underline{y_i} < \underline{x_i} \leq \overline{x_i} < \overline{y_i} \forall i$. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
     LOGICAL FUNCTION interiorv_i(x, y)
     TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:), y(:)
   where
     x and y have shape (n)
```

- Fortran 77 binding:

```
    LOGICAL FUNCTION BLAS_xINTERIORV_I( N, X, INCX, Y, INCY )
    INTEGER            N, INCX, INCY
    <type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
int BLAS_xinteriorv_i( int n, const <interval_array> x, int incx,
                       const <interval_array> y, int incy );
```

DISJV_I (Checks if two interval vectors disjoint) True if $\mathbf{x} \cap \mathbf{y} = \emptyset$

This routine checks if two interval vectors $\mathbf{x}$ and $\mathbf{y}$ are disjoint, which means that $\mathbf{x}_i \cap \mathbf{y}_i = \emptyset$ for some i. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
     LOGICAL FUNCTION disjv_i(x, y)
     TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:), y(:)
   where
     x and y have shape (n)
```

- Fortran 77 binding:

```
    LOGICAL FUNCTION BLAS_xDISJV_I( N, X, INCX, Y, INCY )
    INTEGER            N, INCX, INCY
    <type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
int BLAS_xdisjv_i( int n, const <interval_array> x, int incx,
                   const <interval_array> y, int incy );
```

INTERV_I (Intersection of an interval vector with another) $\mathbf{y} \leftarrow \mathbf{x} \cap \mathbf{y}$.

This routine finds the intersection of two interval vectors $\mathbf{x}$ and $\mathbf{y}$, and stores the result in $\mathbf{y}$. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
 SUBROUTINE interv_i( x, y )
 TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
 TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: y(:)
where
 x and y have shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xINTERV_I( N, X, INCX, Y, INCY )
INTEGER            N, INCX, INCY
<type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xinterv_i( int n, const <interval_array> x, int incx,
                     <interval_array> y, int incy );
```

WINTERV_I (Intersection of two interval vectors) $\mathbf{z} \leftarrow \mathbf{x} \cap \mathbf{y}$.

This routine finds the intersection of two interval vectors $\mathbf{x}$ and $\mathbf{y}$, and stores the result in another interval vector $\mathbf{z}$. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx or incy or incz less than zero is permitted. However, if incx, incy, or incz is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
 SUBROUTINE winterv_i(x, y, z )
 TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:), y(:)
 TYPE(INTERVAL) (<wp>), INTENT(OUT) :: z(:)
where
 x, y and z have shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xWINTERV_I( N, X, INCX, Y, INCY, Z, INCZ )
INTEGER            SIDE, LDA, M, N
INTEGER            N, INCX, INCY, INCZ
<type>             X( 2, * ), Y( 2,* ), Z( 2, * )
```

- C binding:

```
void BLAS_xwinterv_i( int n, const <interval_array> x, int incx,
                      const <interval_array> y, int incy,
                      <interval_array> z, int incz );
```

HULLV_I (Convex hull of an interval vector with another) $\mathbf{y} \leftarrow$ a convex set which contains $\mathbf{x} \cup \mathbf{y}$

This routine computes a convex set which contains both interval vectors $\mathbf{x}$ and $\mathbf{y}$, and overwrites the input interval vector $\mathbf{y}$ with the result. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx or incy less than zero is permitted. However, if incx or incy is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
  SUBROUTINE hullv_i( x, y )
  TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
  TYPE(INTERVAL) (<wp>), INTENT(INOUT) :: y(:)
where
  x and y have shape (n)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xHULLV_I( N, X, INCX, Y, INCY )
INTEGER            N, INCX, INCY
<type>             X( 2, * ), Y( 2, * )
```

- C binding:

```
void BLAS_xhullv_i( int n, const <interval_array> x, int incx,
                    <interval_array> y, int incy );
```

WHULLV_I (Convex hull of two interval vectors) $\mathbf{z} \leftarrow$ a convex set which contains $\mathbf{x} \cup \mathbf{y}$.

This routine finds a convex hull of two interval vectors $\mathbf{x}$ and $\mathbf{y}$, and stores the result in another interval vector $\mathbf{z}$. This routine returns immediately if n is less than or equal to zero. As described in section 2.5.3, the value incx incy, or incz less than zero is permitted. However, if incx or incy or incz is equal to zero, an error flag is set and passed to the error handler.

- Fortran 95 binding:

```
  SUBROUTINE whullv_i( x, y, z )
  TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:), y(:)
  TYPE(INTERVAL) (<wp>), INTENT(OUT) :: z(:)
where
  x, y and z have shape (n)
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xWHULLV_I( N, X, INCX, Y, INCY, Z, INCZ )
      INTEGER            N, INCX, INCY, INCZ
      <type>             X( 2, * ), Y( 2, * ), Z( 2, * )
```

- C binding:

```
void BLAS_xwhullv_i( int n, const <interval_array> x, int incx,
                     const <interval_array> y, int incy, <interval_array> z,
                     int incz );
```

Set Operations Involving Interval Matrices

{GE,GB,SY,SB,SP,TR,TB,TP}_ENCM_I (If an interval matrix is enclosed in another) True if $\mathbf{A} \subseteq \mathbf{B}$

This routine checks if an interval matrix $\mathbf{A}$ is enclosed in another interval matrix $\mathbf{B}$. We say that an interval matrix $\mathbf{A}$ is enclosed in another interval matrix $\mathbf{B}$, denoted as $\mathbf{A} \subseteq \mathbf{B}$, if and only if $\mathbf{a}_{i,j} \subseteq \mathbf{b}_{i,j}$ $\forall i$ and $\forall j$. Matrices $\mathbf{A}$ and $\mathbf{B}$ have the same storage format.

- Fortran 95 binding:

```
General:
      LOGICAL FUNCTION ge_encm_i( a, b )
General Band:
      LOGICAL FUNCTION gb_encm_i( a, m, kl, b )
Symmetric:
      LOGICAL FUNCTION sy_encm_i( a, b [, uplo] )
Symmetric Band:
      LOGICAL FUNCTION sb_encm_i( a, b [, uplo] )
Symmetric Packed:
      LOGICAL FUNCTION sp_encm_i( ap, bp [, uplo] )
Triangular:
      LOGICAL FUNCTION tr_encm_i( a, b [, uplo] [, diag] )
Triangular Band:
      LOGICAL FUNCTION tb_encm_i( a, b [, uplo] [, diag] )
Triangular Packed:
      LOGICAL FUNCTION tp_encm_i( ap, bp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:),  b(:,:) | bp(:)
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
      a and b have shape (m, n) for general matrix
```

```
                                  (l, n) for general banded matrix (l > kl)
                                  (n, n) for symmetric or triangular
                                  (p+1, n) for symmetric banded or triangular
                                         banded (p = band width)
             ap and bp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      LOGICAL FUNCTION BLAS_xGE_ENCM_I( M, N, A, LDA, B, LDB)
General Band:
      LOGICAL FUNCTION BLAS_xGB_ENCM_I( M, N, KL, KU, A, LDA, B, LDB )
Symmetric:
      LOGICAL FUNCTION BLAS_xSY_ENCM_I( N, A, LDA, B, LDB )
Symmetric Band:
      LOGICAL FUNCTION BLAS_xSB_ENCM_I( N, K, A, LDA, B, LDB )
Symmetric Packed:
      LOGICAL FUNCTION BLAS_xSP_ENCM_I( N, AP, BP )
Triangular:
      LOGICAL FUNCTION BLAS_xTR_ENCM_I( N, A, LDA, B, LDB )
Triangular Band:
      LOGICAL FUNCTION BLAS_xTB_ENCM_I( N, K, A, LDA, B, LDB )
Triangular Packed:
      LOGICAL FUNCTION BLAS_xTP_ENCM_I(  N, AP, BP )
all:
      INTEGER           UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, B, LDB
      <type>            A( 2, LDA, * ) or AP( 2, * ), B( 2, LDA, * )
     $                  or BP( 2, * )
```

- C binding:

```
General:
int BLAS_xge_encm_i( enum blas_order_type order, int m, int n,
                     const <interval_array> a, int lda,
                     const <interval_array> b, int ldb );
General Band:
int BLAS_xgb_encm_i( enum blas_order_type order, int m, int n, int kl,
                     int ku, const <interval_array> a, int lda,
                     const <interval_array> b, int ldb );
Symmetric:
int BLAS_xsy_encm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, const <interval_array> a, int lda,
                     const <interval_array> b, int ldb );
Symmetric Band:
int BLAS_xsb_encm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, int k, const <interval_array> a, int lda,
                     const <interval_array> b, int ldb );
```

```
Symmetric Packed:
int BLAS_xsp_encm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     int n, const <interval_array> ap,
                     const <interval_array> bp );
Triangular:
int BLAS_xtr_encm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_diag_type diag, int n,
                     const <interval_array> a, int lda,
                     const <interval_array> b, int ldb );
Triangular Band:
int BLAS_xtb_encm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_diag_type diag, int n, int k,
                     const <interval_array> a, int lda,
                     const <interval_array> b, int ldb );
Triangular Packed:
int BLAS_xtp_encm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                     enum blas_diag_type diag, int n,
                     const <interval_array> ap, <interval_array> bp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_INTERIORM_I (If an interval matrix is in the interior of another interval matrix) True if $\mathbf{A} \subset \mathbf{B}$

This routine checks if an interval matrix **A** is enclosed in the interior of another interval matrix **B**. We say that an interval matrix **A** is enclosed in the interior of an interval vatrix **B**, if and only if $\mathbf{a}_{i,j} \subset \mathbf{b}_{i,j}$ $\forall i$ and $\forall j$. Matrices **A** and **B** have the same storage format.

- Fortran 95 binding:

```
General:
      LOGICAL FUNCTION ge_interiorm_i( a, b )
General Band:
      LOGICAL FUNCTION gb_interiorm_i( a, m, kl, b )
Symmetric:
      LOGICAL FUNCTION sy_interiorm_i( a, b [, uplo] )
Symmetric Band:
      LOGICAL FUNCTION sb_interiorm_i( a, b [, uplo] )
Symmetric Packed:
      LOGICAL FUNCTION sp_interiorm_i( ap, bp [, uplo] )
Triangular:
      LOGICAL FUNCTION tr_interiorm_i( a, b [, uplo] [, diag] )
Triangular Band:
      LOGICAL FUNCTION tb_interiorm_i( a, b [, uplo] [, diag] )
Triangular Packed:
      LOGICAL FUNCTION tp_interiorm_i( ap, bp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:), b(:,:) | bp(:)
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
```

```
        TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
      where
         a and b have shape (m, n) for general matrix
                            (l, n) for general banded matrix (l > kl)
                            (n, n) for symmetric or triangular
                            (p+1, n) for symmetric banded or triangular
                                   banded (p = band width)
         ap and bp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      LOGICAL FUNCTION BLAS_xGE_INTERIORM_I( M, N, A, LDA, B, LDB )
General Band:
      LOGICAL FUNCTION BLAS_xGB_INTERIORM_I( M, N, KL, KU, A, LDA, B, LDB )
Symmetric:
      LOGICAL FUNCTION BLAS_xSY_INTERIORM_I( N, A, LDA, B, LDB )
Symmetric Band:
      LOGICAL FUNCTION BLAS_xSB_INTERIORM_I( N, K, A, LDA, B, LDB )
Symmetric Packed:
      LOGICAL FUNCTION BLAS_xSP_INTERIORM_I( N, AP, BP )
Triangular:
      LOGICAL FUNCTION BLAS_xTR_INTERIORM_I( N, A, LDA, B, LDB )
Triangular Band:
      LOGICAL FUNCTION BLAS_xTB_INTERIORM_I( N, K, A, LDA, B, LDB )
Triangular Packed:
      LOGICAL FUNCTION BLAS_xTP_INTERIORM_I(  N, AP, BP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, B, LDB
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDA, * )
     $                   or BP( 2, * )
```

- C binding:

```
General:
int BLAS_xge_interiorm_i( enum blas_order_type order, int m, int n,
                          const <interval_array> a, int lda,
                          const <interval_array> b, int ldb );
General Band:
int BLAS_xgb_interiorm_i( enum blas_order_type order, int m, int n, int kl,
                          int ku, const <interval_array> a, int lda,
                          const <interval_array> b, int ldb );
Symmetric:
int BLAS_xsy_interiorm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          int n, const <interval_array> a, int lda,
                          const <interval_array> b, int ldb );
Symmetric Band:
```

```
int BLAS_xsb_interiorm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          int n, int k, const <interval_array> a, int lda,
                          const <interval_array> b, int ldb );
Symmetric Packed:
int BLAS_xsp_interiorm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          int n, const <interval_array> ap,
                          const <interval_array> bp );
Triangular:
int BLAS_xtr_interiorm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          enum blas_diag_type diag, int n,
                          const <interval_array> a, int lda,
                          const <interval_array> b, int ldb );
Triangular Band:
int BLAS_xtb_interiorm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          enum blas_diag_type diag, int n, int k,
                          const <interval_array> a, int lda,
                          const <interval_array> b, int ldb );
Triangular Packed:
int BLAS_xtp_interiorm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          enum blas_diag_type diag, int n,
                          const <interval_array> ap, <interval_array> bp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_DISJM_I (If two interval matrices disjoint) True if $\mathbf{A} \cap \mathbf{B} = \emptyset$

This routine checks if two interval matrices $\mathbf{A}$ and $\mathbf{B}$ disjoint, which means that if for some i, j, $\mathbf{a}_{i,j} \cap \mathbf{b}_{i,j} = \emptyset$. Matrices $\mathbf{A}$ and $\mathbf{B}$ have the same storage format.

- Fortran 95 binding:

```
General:
      LOGICAL FUNCTION ge_disjm_i( a, b )
General Band:
      LOGICAL FUNCTION gb_disjm_i( a, m, kl, b )
Symmetric:
      LOGICAL FUNCTION sy_disjm_i( a, b [, uplo] )
Symmetric Band:
      LOGICAL FUNCTION sb_disjm_i( a, b [, uplo] )
Symmetric Packed:
      LOGICAL FUNCTION sp_disjm_i( ap, bp [, uplo] )
Triangular:
      LOGICAL FUNCTION tr_disjm_i( a, b [, uplo] [, diag] )
Triangular Band:
      LOGICAL FUNCTION tb_disjm_i( a, b [, uplo] [, diag] )
Triangular Packed:
      LOGICAL FUNCTION tp_disjm_i( ap, bp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:), b(:,:) | bp(:)
```

```
          TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
          TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
        where
           a and b have shape (m, n) for general matrix
                              (l, n) for general banded matrix (l > kl)
                              (n, n) for symmetric or triangular
                              (p+1, n) for symmetric banded or triangular
                                     banded (p = band width)
           ap and bp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      LOGICAL FUNCTION BLAS_xGE_DISJM_I( M, N, A, LDA, B, LDB )
General Band:
      LOGICAL FUNCTION BLAS_xGB_DISJM_I( M, N, KL, KU, A, LDA, B, LDB )
Symmetric:
      LOGICAL FUNCTION BLAS_xSY_DISJM_I( N, A, LDA, B, LDB )
Symmetric Band:
      LOGICAL FUNCTION BLAS_xSB_DISJM_I( N, K, A, LDA, B, LDB )
Symmetric Packed:
      LOGICAL FUNCTION BLAS_xSP_DISJM_I( N, AP, BP )
Triangular:
      LOGICAL FUNCTION BLAS_xTR_DISJM_I( N, A, LDA, B, LDB )
Triangular Band:
      LOGICAL FUNCTION BLAS_xTB_DISJM_I( N, K, A, LDA, B, LDB )
Triangular Packed:
      LOGICAL FUNCTION BLAS_xTP_DISJM_I(  N, AP, BP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, B, LDB
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDA, * )
     $                   or BP( 2, * )
```

- C binding:

```
General:
int BLAS_xge_disjm_i( enum blas_order_type order, int m, int n,
                      const <interval_array> a, int lda,
                      const <interval_array> b, int ldb );
General Band:
int BLAS_xgb_disjm_i( enum blas_order_type order, int m, int n, int kl,
                      int ku, const <interval_array> a, int lda,
                      const <interval_array> b, int ldb );
Symmetric:
int BLAS_xsy_disjm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, const <interval_array> a, int lda,
                      const <interval_array> b, int ldb );
```

```
Symmetric Band:
int BLAS_xsb_disjm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, int k, const <interval_array> a, int lda,
                      const <interval_array> b, int ldb );
Symmetric Packed:
int BLAS_xsp_disjm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, const <interval_array> ap,
                      const <interval_array> bp );
Triangular:
int BLAS_xtr_disjm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_diag_type diag, int n,
                      const <interval_array> a, int lda,
                      const <interval_array> b, int ldb );
Triangular Band:
int BLAS_xtb_disjm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_diag_type diag, int n, int k,
                      const <interval_array> a, int lda,
                      const <interval_array> b, int ldb );
Triangular Packed:
int BLAS_xtp_disjm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_diag_type diag, int n,
                      const <interval_array> ap, <interval_array> bp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_INTERM_I (Elementwise intersection of two interval matrices)
$\mathbf{B} \leftarrow \mathbf{A} \cap \mathbf{B}$

This routine finds the elementwise intersection of two interval matrices **A** and **B**, and stores the result in **B**. Matrices **A** and **B** have the same storage format.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_interm_i( a, b )
General Band:
      SUBROUTINE gb_interm_i( a, m, kl, b )
Symmetric:
      SUBROUTINE sy_interm_i( a, b [, uplo] )
Symmetric Band:
      SUBROUTINE sb_interm_i( a, b [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_interm_i( ap, bp [, uplo] )
Triangular:
      SUBROUTINE tr_interm_i( a, b [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_interm_i( a, b [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_interm_i( ap, bp [, uplo], [, diag] )
```

```
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:) | ap(:), b(:,:) | bp(:)
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a and b have shape (m, n) for general matrix
                          (l, n) for general banded matrix (l > kl)
                          (n, n) for symmetric or triangular
                          (p+1, n) for symmetric banded or triangular
                                 banded (p = band width)
       ap and bp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_INTERM_I( M, N, A, LDA, B, LDB )
General Band:
      SUBROUTINE BLAS_xGB_INTERM_I( M, N, KL, KU, A, LDA, B, LDB )
Symmetric:
      SUBROUTINE BLAS_xSY_INTERM_I( N, A, LDA, B, LDB )
Symmetric Band:
      SUBROUTINE BLAS_xSB_INTERM_I( N, K, A, LDA, B, LDB )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_INTERM_I( N, AP, BP )
Triangular:
      SUBROUTINE BLAS_xTR_INTERM_I( N, A, LDA, B, LDB )
Triangular Band:
      SUBROUTINE BLAS_xTB_INTERM_I( N, K, A, LDA, B, LDB )
Triangular Packed:
      SUBROUTINE BLAS_xTP_INTERM_I(  N, AP, BP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, B, LDB
      <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDA, * )
     $                   or BP( 2, * )
```

- C binding:

```
General:
void BLAS_xge_interm_i( enum blas_order_type order, int m, int n,
                        const <interval_array> a, int lda,
                        <interval_array> b, int ldb );
General Band:
void BLAS_xgb_interm_i( enum blas_order_type order, int m, int n, int kl,
                        int ku, const <interval_array> a, int lda,
                        <interval_array> b, int ldb );
Symmetric:
void BLAS_xsy_interm_i( enum blas_order_type order, enum blas_uplo_type uplo,
```

```
                         int n, const <interval_array> a, int lda,
                         <interval_array> b, int ldb );
Symmetric Band:
void BLAS_xsb_interm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         int n, int k, const <interval_array> a, int lda,
                         <interval_array> b, int ldb );
Symmetric Packed:
void BLAS_xsp_interm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         int n, const <interval_array> ap,
                         <interval_array> bp );
Triangular:
void BLAS_xtr_interm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_diag_type diag, int n,
                         const <interval_array> a, int lda,
                         <interval_array> b, int ldb );
Triangular Band:
void BLAS_xtb_interm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_diag_type diag, int n, int k,
                         const <interval_array> a, int lda,
                         <interval_array> b, int ldb );
Triangular Packed:
void BLAS_xtp_interm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_diag_type diag, int n,
                         const <interval_array> ap, <interval_array> bp );
```

GE_WINTERM_I (Intersection of two interval matrices) $\mathbf{C} \leftarrow \mathbf{A} \cap \mathbf{B}$

This routine finds the intersection of two interval matrices $\mathbf{A}$ and $\mathbf{B}$, and stores the result in another interval matrix $\mathbf{C}$. Matrices $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ have the same storage format.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_winterm_i( a, b, c )
General Band:
      SUBROUTINE gb_winterm_i( a, m, kl, b, c )
Symmetric:
      SUBROUTINE sy_winterm_i( a, b, c [, uplo] )
Symmetric Band:
      SUBROUTINE sb_winterm_i( a, b, c [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_winterm_i( ap, bp, cp [, uplo] )
Triangular:
      SUBROUTINE tr_winterm_i( a, b, c [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_winterm_i( a, b, c [, uplo] [, diag] )
Triangular Packed:
```

```
      SUBROUTINE tp_winterm_i( ap, bp, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), b(:,:),
                                           c(:,:), ap(:), bp(:), cp(:)
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a, b and c have shape (m, n) for general matrix
                             (l, n) for general banded matrix (l > kl)
                             (n, n) for symmetric or triangular
                             (p+1, n) for symmetric banded or triangular
                                    banded (p = band width)
       ap, bp and cp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_WINTERM_I( M, N, A, LDA, B, LDB, C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_WINTERM_I( M, N, KL, KU, A, LDA, B, LDB,
     $                               C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_WINTERM_I( N, A, LDA, B, LDB, C, LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_WINTERM_I( N, K, A, LDA, B, LDB, C, LDC )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_WINTERM_I( N, AP, BP, CP )
Triangular:
      SUBROUTINE BLAS_xTR_WINTERM_I( N, A, LDA, B, LDB, C, LDC )
Triangular Band:
      SUBROUTINE BLAS_xTB_WINTERM_I( N, K, A, LDA, B, LDB, C, LDC )
Triangular Packed:
      SUBROUTINE BLAS_xTP_WINTERM_I(  N, AP, BP, CP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, B, LDB,
     $                   C, LDC
       <type>            A( 2, LDA, * ) or AP( 2, * ), B( 2, LDA, *)
     $                   or BP(2,*), C(2, LDC, *) or CP(2,*)
```

- C binding:

```
General:
void BLAS_xge_winterm_i( enum blas_order_type order, int m, int n,
                         const <interval_array> a, int lda,
                         const <interval_array> b, int ldb,
                         <interval_array> c, int ldc );
General Band:
```

```
void BLAS_xgb_winterm_i( enum blas_order_type order, int m, int n, int kl,
                         int ku, const <interval_array> a, int lda,
                         const <interval_array> b, int ldb,
                         <interval_array> c, int ldc );
Symmetric:
void BLAS_xsy_winterm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         int n, const <interval_array> a, int lda,
                         const <interval_array> b, int ldb,
                         <interval_array> c, int ldc );
Symmetric Band:
void BLAS_xsb_winterm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         int n, int k, const <interval_array> a, int lda,
                         const <interval_array> b, int ldb,
                        <interval_array> c, int ldc );
Symmetric Packed:
void BLAS_xsp_winterm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         int n, const <interval_array> ap,
                         const <interval_array> bp, <interval_array> cp );
Triangular:
void BLAS_xtr_winterm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_diag_type diag, int n,
                         const <interval_array> a, int lda,
                         const <interval_array> b, int ldb,
                         <interval_array> c, int ldc );
Triangular Band:
void BLAS_xtb_winterm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_diag_type diag, int n, int k,
                         const <interval_array> a, int lda,
                         const <interval_array> b, int ldb,
                         <interval_array> c, int ldc );
Triangular Packed:
void BLAS_xtp_winterm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                         enum blas_diag_type diag, int n,
                         const <interval_array> ap, <interval_array> bp,
                         <interval_array> cp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_HULLM_I (Convex hull of an interval matrix with another) $\mathbf{B} \leftarrow$ the convex hull contains $\mathbf{A} \cup \mathbf{B}$

This routine finds an interval matrix which contains both interval matrices **A** and **B**, and stores the result in **B**. Matrices **A** and **B** have the same storage format.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_hullm_i( a, b )
General Band:
```

```
            SUBROUTINE gb_hullm_i( a, m, kl, b )
      Symmetric:
            SUBROUTINE sy_hullm_i( a, b [, uplo] )
      Symmetric Band:
            SUBROUTINE sb_hullm_i( a, b [, uplo] )
      Symmetric Packed:
            SUBROUTINE sp_hullm_i( ap, bp [, uplo] )
      Triangular:
            SUBROUTINE tr_hullm_i( a, b [, uplo] [, diag] )
      Triangular Band:
            SUBROUTINE tb_hullm_i( a, b [, uplo] [, diag] )
      Triangular Packed:
            SUBROUTINE tp_hullm_i( ap, bp [, uplo], [, diag] )
      all:
            TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), b(:,:), ap(:), bp(:)
            TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
            TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
          where
             a and b have shape (m, n) for general matrix
                                (l, n) for general banded matrix (l > kl)
                                (n, n) for symmetric or triangular
                                (p+1, n) for symmetric banded or triangular
                                       banded (p = band width)
             ap and bp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
      General:
            SUBROUTINE BLAS_xGE_HULLM_I( M, N, A, LDA, B, LDB )
      General Band:
            SUBROUTINE BLAS_xGB_HULLM_I( M, N, KL, KU, A, LDA, B, LDB )
      Symmetric:
            SUBROUTINE BLAS_xSY_HULLM_I( N, A, LDA, B, LDB )
      Symmetric Band:
            SUBROUTINE BLAS_xSB_HULLM_I( N, K, A, LDA, B, LDB )
      Symmetric Packed:
            SUBROUTINE BLAS_xSP_HULLM_I( N, AP, BP )
      Triangular:
            SUBROUTINE BLAS_xTR_HULLM_I( N, A, LDA, B, LDB )
      Triangular Band:
            SUBROUTINE BLAS_xTB_HULLM_I( N, K, A, LDA, B, LDB )
      Triangular Packed:
            SUBROUTINE BLAS_xTP_HULLM_I(  N, AP, BP )
      all:
            INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, B, LDB
            <type>             A( 2, LDA, * ) or AP( 2, * ), B( 2, LDA, * )
           $                   or BP( 2, * )
```

- C binding:

```
General:
void BLAS_xge_hullm_i( enum blas_order_type order, int m, int n,
                       const <interval_array> a, int lda,
                       <interval_array> b, int ldb );
General Band:
void BLAS_xgb_hullm_i( enum blas_order_type order, int m, int n, int kl,
                       int ku, const <interval_array> a, int lda,
                       <interval_array> b, int ldb );
Symmetric:
void BLAS_xsy_hullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                       int n, const <interval_array> a, int lda,
                       <interval_array> b, int ldb );
Symmetric Band:
void BLAS_xsb_hullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                       int n, int k, const <interval_array> a, int lda,
                       <interval_array> b, int ldb );
Symmetric Packed:
void BLAS_xsp_hullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                       int n, const <interval_array> ap,
                       <interval_array> bp );
Triangular:
void BLAS_xtr_hullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                       enum blas_diag_type diag, int n,
                       const <interval_array> a, int lda,
                       <interval_array> b, int ldb );
Triangular Band:
void BLAS_xtb_hullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                       enum blas_diag_type diag, int n, int k,
                       const <interval_array> a, int lda,
                       <interval_array> b, int ldb );
Triangular Packed:
void BLAS_xtp_hullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                       enum blas_diag_type diag, int n,
                       const <interval_array> ap, <interval_array> bp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_WHULLM_I (Convex hull of two interval matrices)

$$\mathbf{C} \leftarrow \text{the convex hull contains } \mathbf{A} \cup \mathbf{B}$$

This routine finds the convex set which contains both interval matrices $\mathbf{A}$ and $\mathbf{B}$, and stores the result in an interval matrix $\mathbf{C}$. Matrices $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ have the same storage format.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_whullm_i( a, b, c )
```

```
General Band:
      SUBROUTINE gb_whullm_i( a, m, kl, b, c )
Symmetric:
      SUBROUTINE sy_whullm_i( a, b, c [, uplo] )
Symmetric Band:
      SUBROUTINE sb_whullm_i( a, b, c [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_whullm_i( ap, bp, cp [, uplo] )
Triangular:
      SUBROUTINE tr_whullm_i( a, b, c [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_whullm_i( a, b, c [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_whullm_i( ap, bp, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), b(:,:),
                                           c(:,:), ap(:), bp(:), cp(:)
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a, b and c have shape (m, n) for general matrix
                             (l, n) for general banded matrix (l > kl)
                             (n, n) for symmetric or triangular
                             (p+1, n) for symmetric banded or triangular
                                    banded (p = band width)
       ap, bp and cp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_WHULLM_I( M, N, A, LDA, B, LDB, C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_WHULLM_I( M, N, KL, KU, A, LDA, B, LDB,
     $                              C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_WHULLM_I( N, A, LDA, B, LDB, C, LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_WHULLM_I( N, K, A, LDA, B, LDB, C, LDC )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_WHULLM_I( N, AP, BP, CP )
Triangular:
      SUBROUTINE BLAS_xTR_WHULLM_I( N, A, LDA, B, LDB, C, LDC )
Triangular Band:
      SUBROUTINE BLAS_xTB_WHULLM_I( N, K, A, LDA, B, LDB, C, LDC )
Triangular Packed:
      SUBROUTINE BLAS_xTP_WHULLM_I(  N, AP, BP, CP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, B, LDB
```

```
      $                    C, LDC
       <type>              A( 2, LDA, * ) or AP( 2, * ), B( 2, LDA, *)
      $                    or BP(2,*), C(2, LDC, *) or CP(2,*)
```

- C binding:

```
General:
void BLAS_xge_whullm_i( enum blas_order_type order, int m, int n,
                        const <interval_array> a, int lda,
                        const <interval_array> b, int ldb,
                        <interval_array> c, int ldc );
General Band:
void BLAS_xgb_whullm_i( enum blas_order_type order, int m, int n, int kl,
                        int ku, const <interval_array> a, int lda,
                        const <interval_array> b, int ldb,
                        <interval_array> c, int ldc );
Symmetric:
void BLAS_xsy_whullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        int n, const <interval_array> a, int lda,
                        const <interval_array> b, int ldb,
                        <interval_array> c, int ldc );
Symmetric Band:
void BLAS_xsb_whullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        int n, int k, const <interval_array> a, int lda,
                        const <interval_array> b, int ldb,
                        <interval_array> c, int ldc );
Symmetric Packed:
void BLAS_xsp_whullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        int n, const <interval_array> ap,
                        const <interval_array> bp, <interval_array> cp );
Triangular:
void BLAS_xtr_whullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        enum blas_diag_type diag, int n,
                        const <interval_array> a, int lda,
                        const <interval_array> b, int ldb,
                        <interval_array> c, int ldc );
Triangular Band:
void BLAS_xtb_whullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        enum blas_diag_type diag, int n, int k,
                        const <interval_array> a, int lda,
                        const <interval_array> b, int ldb,
                        <interval_array> c, int ldc );
Triangular Packed:
void BLAS_xtp_whullm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        enum blas_diag_type diag, int n,
                        const <interval_array> ap, <interval_array> bp,
                        <interval_array> cp );
```

Utility Functions Involving Interval Vectors

EMPTYELEV_I (Empty entry & location) $k \leftarrow \mathbf{x}_k = \emptyset$; or -1

This routine checks if an interval vector, $\mathbf{x}$, contains an empty interval entry. If $\mathbf{x}$ contains empty interval entries, then the routine returns the smallest offset or index k such that $\mathbf{x}_k = [\text{NaN\_empty}, \text{NaN\_empty}]$. Otherwise, the routine returns -1.

- Fortran 95 binding:

```
    INTEGER FUNCTION emptyelev_i( x )
    TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
  where
    x has shape (n)
```

- Fortran 77 binding:

```
    INTEGER FUNCTION BLAS_xEMPTYELEV_I( N, X, INCX )
    INTEGER            N, INCX
    <type>             X( 2, * )
```

- C binding:

```
int BLAS_xemptyelev_i( int n, const <interval_array> x, int incx);
```

INFV_I (The left endpoint of an interval vector) $v \leftarrow \underline{x}$

This routine finds the real vector v such that $v_i = \underline{x}_i \; \forall i$.

- Fortran 95 binding:

```
    SUBROUTINE infv_i( x, v )
    REAL (<wp>), INTENT(OUT) :: v(:)
    TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
  where
    v and x have shape (n)
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xINFV_I( N, X, INCX, V )
    INTEGER            N, INCX
    <type>             X( 2, * ), V( * )
```

- C binding:

```
void BLAS_xinfv_i( int n, const <interval_array> x, int incx, RARRAY v );
```

SUPV_I (The right endpoint of an interval vector) $v \leftarrow \overline{x}$

This routine finds the real vector v such that $v_i = \overline{x}_i \; \forall i$.

- Fortran 95 binding:

```
    SUBROUTINE supv_i( x, v )
    REAL (<wp>), INTENT(OUT) :: v(:)
    TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
  where
    v and x have shape (n)
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xSUPV_I( N, X, INCX, V )
    INTEGER            N, INCX
    <type>             X( 2, * ), V( * )
```

- C binding:

```
void BLAS_xsupv_i( int n, const <interval_array> x, int incx, RARRAY v );
```

MIDV_I (The approximate midpoint of an interval vector) $v \leftarrow (\overline{x} + \underline{x})/2$

This routine finds the real vector v such that $v_i = \dfrac{\overline{x}_i + \underline{x}_i}{2} \; \forall i$.

- Fortran 95 binding:

```
    SUBROUTINE midv_i( x, v )
    REAL (<wp>), INTENT(OUT) :: v(:)
    TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
  where
    v and x have shape (n)
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xMIDV_I( N, X, INCX, V )
    INTEGER            N, INCX
    <type>             X( 2, * ), V( * )
```

- C binding:

```
void BLAS_xmidv_i( int n, const <interval_array> x, int incx, RARRAY v );
```

WIDTHV_I (The elementwise width of an interval vector) $v \leftarrow \overline{x} - \underline{x}$

This routine finds the real vector v such that $v_i = \overline{x}_i - \underline{x}_i \; \forall i$.

- Fortran 95 binding:

```
    SUBROUTINE widthv_i( x, v )
    REAL (<wp>), INTENT(OUT) :: v(:)
    TYPE(INTERVAL) (<wp>), INTENT(IN) :: x(:)
  where
    v and x have shape (n)
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xWIDTHV_I( N, X, INCX, V )
    INTEGER            N, INCX
    <type>             X( 2, * ), V( * )
```

- C binding:

```
void BLAS_xwidthv_i( int n, const <interval_array> x, int incx, RARRAY v );
```

CONSTRUCTV_I (Constructs an interval vector from two floating point vectors)

$$\mathbf{x} \leftarrow [\min\{u, v\}, \max\{u, v\}]$$

This routine constructs an interval vector $\mathbf{x}$ from two floating point vectors u and v such that $\mathbf{x}_i$ contains the interval $[\min\{u_i, v_i\}, \max\{u_i, v_i\}] \; \forall i$. By letting $u = v$, the routine constructs an interval vector from a single floating point vector.

- Fortran 95 binding:

```
    SUBROUTINE constructv_i( x, u, v )
    REAL (<wp>), INTENT(IN) :: u(:), v(:)
    TYPE(INTERVAL) (<wp>), INTENT(OUT) :: x(:)
  where
    u, v and x have shape (n)
```

- Fortran 77 binding:

```
    SUBROUTINE BLAS_xCONSTRUCTV_I( N, U, INCU, V, INCV, X, INCX )
    INTEGER            N, INCU, INCV, INCX
    <type>             X( 2, * ), U( * ), V( * )
```

- C binding:

```
void BLAS_xconstructv_i( int n, RARRAY u, int incu, RARRAY v, int incv,
                         <interval_array> x, int incx );
```

Utility Functions Involving Interval Matrices

{GE,GB,SY,SB,SP,TR,TB,TP}_EMPTYELEM_I (Empty entry & location) $l \leftarrow \mathbf{a}_{l,j} = \emptyset$; or -1

This routine checks if an interval matrix, $\mathbf{A}$, contains an empty interval entry. If $\mathbf{A}$ contains empty interval entries, then the routine returns the smallest offset or index l (according to the first index) such that $\mathbf{a}_{l,j} = [\text{NaN\_empty}, \text{NaN\_empty}]$. Otherwise, it returns -1.

- Fortran 95 binding:

```
General:
      INTEGER FUNCTION  ge_emptyelem_i( a )
General Band:
      INTEGER FUNCTION  gb_emptyelem_i( a, m, kl )
Symmetric:
      INTEGER FUNCTION  sy_emptyelem_i( a [, uplo] )
Symmetric Band:
      INTEGER FUNCTION  sb_emptyelem_i( a, kl [, uplo] )
Symmetric Packed:
      INTEGER FUNCTION  sp_emptyelem_i( ap [, uplo] )
Triangular:
      INTEGER FUNCTION  tr_emptyelem_i( a [, uplo] [, diag] )
Triangular Band:
      INTEGER FUNCTION  tb_emptyelem_i( a, kl [, uplo] [, diag] )
Triangular Packed:
      INTEGER FUNCTION  tp_emptyelem_i( ap, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), ap(:)
      INTEGER, INTENT(OUT) :: i, j
      INTEGER, INTENT(IN) :: kl
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a has shape (m, n) for general matrix
                   (l, n) for general banded matrix (l > kl)
                   (n, n) for symmetric or triangular
                   (p+1, n) for symmetric banded or triangular
                            banded (p = band width)
       ap has shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      INTEGER FUNCTION  BLAS_xGE_EMPTYELEM_I( M, N, A, LDA, B, LDB )
General Band:
      INTEGER FUNCTION  BLAS_xGB_EMPTYELEM_I( M, N, KL, KU, A, LDA )
Symmetric:
      INTEGER FUNCTION  BLAS_xSY_EMPTYELEM_I( UPLO, N, A, LDA )
```

```
Symmetric Band:
      INTEGER FUNCTION  BLAS_xSB_EMPTYELEM_I( UPLO, N, K, A, LDA )
Symmetric Packed:
      INTEGER FUNCTION  BLAS_xSP_EMPTYELEM_I( UPLO, N, AP )
Triangular:
      INTEGER FUNCTION  BLAS_xTR_EMPTYELEM_I( UPLO, TRANS, DIAG, N, A, LDA )
Triangular Band:
      INTEGER FUNCTION  BLAS_xTB_EMPTYELEM_I( UPLO, TRANS, DIAG, N, K, A, LDA )
Triangular Packed:
      INTEGER FUNCTION  BLAS_xTP_EMPTYELEM_I( UPLO, TRANS, DIAG, N, AP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, I, J
      <type>             A( 2, LDA, * ) or AP( 2, * )
```

- C binding:

```
General:
int BLAS_xge_emptyelem_i( enum blas_order_type order, int m, int n,
                          const <interval_array> a, int lda );
General Band:
int BLAS_xgb_emptyelem_i( enum blas_order_type order, int m, int n, int kl,
                          int ku, const <interval_array> a, int lda, int i,
                          int j);
Symmetric:
int BLAS_xsy_emptyelem_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          int n, const <interval_array> a, int lda );
Symmetric Band:
int BLAS_xsb_emptyelem_i( enum blas_order_type order, int n, int k,
                          const <interval_array> a, int lda );
Symmetric Packed:
int BLAS_xsp_emptyelem_i( enum blas_order_type order, int n,
                          const <interval_array> ap );
Triangular:
int BLAS_xtr_emptyelem_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          enum blas_trans_type trans, enum blas_diag_type diag,
                          int n, const <interval_array> a, int lda );
Triangular Band:
int BLAS_xtb_emptyelem_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          enum blas_trans_type trans, enum blas_diag_type diag,
                          int n, int k, const <interval_array> a, int lda );
Triangular Packed:
int BLAS_xtp_emptyelem_i( enum blas_order_type order, enum blas_uplo_type uplo,
                          enum blas_trans_type trans, enum blas_diag_type diag,
                          int n, const <interval_array> ap, int i, int j);
```

{GE,GB,SY,SB,SP,TR,TB,TP}_INFM_I (Left endpoint of an interval matrix) $C \leftarrow \underline{A}$

This routine finds the real matrix C such that $c_{i,j} = \underline{a}_{i,j}$ $\forall i$ and $\forall j$, where $\mathbf{A} = \{\mathbf{a}_{i,j}\}$ is a general (or general banded, or symmetric, or symmetric banded, symmetric packed, or triangular, triangular banded, triangular packed) interval matrix.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_infm_i( a, c )
General Band:
      SUBROUTINE gb_infm_i( a, m, kl, c )
Symmetric:
      SUBROUTINE sy_infm_i( a, c [, uplo] )
Symmetric Band:
      SUBROUTINE sb_infm_i( a, kl, c [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_infm_i( ap, cp [, uplo] )
Triangular:
      SUBROUTINE tr_infm_i( a, c [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_infm_i( a, kl, c [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_infm_i( ap, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), ap(:)
      REAL (<wp>), INTENT(OUT) :: c(:,:), cp(:)
      INTEGER, INTENT(IN) :: kl
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a and c have shape
           (m, n) for general matrix
           (l, n) for general banded matrix (l > kl)
           (n, n) for symmetric or triangular
           (p+1, n) for symmetric banded or triangular
                    banded (p = band width)
       ap and cp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_INFM_I( M, N, A, LDA C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_INFM_I( M, N, KL, KU, A, LDA, C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_INFM_I( UPLO, N, A, LDA, C, LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_INFM_I( UPLO, N, K, A, LDA, C, LDC )
Symmetric Packed:
```

```
          SUBROUTINE BLAS_xSP_INFM_I( UPLO, N, AP, CP )
     Triangular:
          SUBROUTINE BLAS_xTR_INFM_I( UPLO, TRANS, DIAG, N, A, LDA, C, LDC )
     Triangular Band:
          SUBROUTINE BLAS_xTB_INFM_I( UPLO, TRANS, DIAG, N, K, A, LDA, C, LDC )
     Triangular Packed:
          SUBROUTINE BLAS_xTP_INFM_I( UPLO, TRANS, DIAG, N, AP, CP )
     all:
          INTEGER           UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, LDC
          <type>            A( 2, LDA, * ) or AP( 2, * ), C( LDC, * ) or CP( * )
```

- C binding:

```
General:
void BLAS_xge_infm_i( enum blas_order_type order, int m, int n,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
General Band:
void BLAS_xgb_infm_i( enum blas_order_type order, int m, int n, int kl, int ku,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
Symmetric:
void BLAS_xsy_infm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Symmetric Band:
void BLAS_xsb_infm_i( enum blas_order_type order, int n, int k,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
Symmetric Packed:
void BLAS_xsp_infm_i( enum blas_order_type order, int n,
                      const <interval_array> ap, RARRAY cp );
Triangular:
void BLAS_xtr_infm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Triangular Band:
void BLAS_xtb_infm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, int k, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Triangular Packed:
void BLAS_xtp_infm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> ap, RARRAY cp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_SUPM_I (Right endpoint of an interval matrix) $C \leftarrow \overline{A}$

This routine finds the real matrix C such that $c_{i,j} = \overline{a}_{i,j}\ \forall i$ and $\forall j$, where $\mathbf{A} = \{\mathbf{a}_{i,j}\}$ is a general (or general banded, or symmetric, or symmetric banded, symmetric packed, or triangular, triangular banded, triangular packed) interval matrix.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_supm_i( a, c )
General Band:
      SUBROUTINE gb_supm_i( a, m, kl, c )
Symmetric:
      SUBROUTINE sy_supm_i( a, c [, uplo] )
Symmetric Band:
      SUBROUTINE sb_supm_i( a, kl, c [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_supm_i( ap, cp [, uplo] )
Triangular:
      SUBROUTINE tr_supm_i( a, c [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_supm_i( a, kl, c [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_supm_i( ap, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), ap(:)
      REAL (<wp>), INTENT(OUT) :: c(:,:), cp(:)
      INTEGER, INTENT(IN) :: kl
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a and c have shape
           (m, n) for general matrix
           (l, n) for general banded matrix (l > kl)
           (n, n) for symmetric or triangular
           (p+1, n) for symmetric banded or triangular
                    banded (p = band width)
       ap and cp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_SUPM_I( M, N, A, LDA, C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_SUPM_I( M, N, KL, KU, A, LDA, C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_SUPM_I( UPLO, N, A, LDA, C, LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_SUPM_I( UPLO, N, K, A, LDA,  C, LDC )
Symmetric Packed:
```

```
      SUBROUTINE BLAS_xSP_SUPM_I( UPLO, N, AP, CP )
Triangular:
      SUBROUTINE BLAS_xTR_SUPM_I( UPLO, TRANS, DIAG, N, A, LDA, C, LDC )
Triangular Band:
      SUBROUTINE BLAS_xTB_SUPM_I( UPLO, TRANS, DIAG, N, K, A, LDA, C, LDC )
Triangular Packed:
      SUBROUTINE BLAS_xTP_SUPM_I( UPLO, TRANS, DIAG, N, AP, CP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, LDC
      <type>             A( 2, LDA, * ) or AP( 2, * ), C( LDC, * ) or CP( * )
```

- C binding:

```
General:
void BLAS_xge_supm_i( enum blas_order_type order, int m, int n,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
General Band:
void BLAS_xgb_supm_i( enum blas_order_type order, int m, int n, int kl, int ku,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
Symmetric:
void BLAS_xsy_supm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Symmetric Band:
void BLAS_xsb_supm_i( enum blas_order_type order, int n, int k,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
Symmetric Packed:
void BLAS_xsp_supm_i( enum blas_order_type order, int n,
                      const <interval_array> ap, RARRAY cp );
Triangular:
void BLAS_xtr_supm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Triangular Band:
void BLAS_xtb_supm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, int k, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Triangular Packed:
void BLAS_xtp_supm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> ap, RARRAY cp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_MIDM_I (Midpoint matrix of an interval matrix) $C \leftarrow (\mathbf{A}+\mathbf{B})/2$

This routine finds the real matrix C such that $BLAS_{i,j} = \dfrac{\underline{a}_{i,j} + \overline{a}_{i,j}}{2}$ $\forall i$ and $\forall j$, where $\mathbf{A} = \{\mathbf{a}_{i,j}\}$ is a general (or general banded, or symmetric, or symmetric banded, symmetric packed, or triangular, triangular banded, triangular packed) interval matrix.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_midm_i( a, c)
General Band:
      SUBROUTINE gb_midm_i( a, m, kl, c )
Symmetric:
      SUBROUTINE sy_midm_i( a, c [, uplo] )
Symmetric Band:
      SUBROUTINE sb_midm_i( a, kl, c [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_midm_i( ap, cp [, uplo] )
Triangular:
      SUBROUTINE tr_midm_i( a, c [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_midm_i( a, kl, c [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_midm_i( ap, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), ap(:)
      REAL (<wp>), INTENT(OUT) :: c(:,:), cp(:)
      INTEGER, INTENT(IN) :: kl
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a and c have shape
           (m, n) for general matrix
           (l, n) for general banded matrix (l > kl)
           (n, n) for symmetric or triangular
           (p+1, n) for symmetric banded or triangular
                    banded (p = band width)
       ap and cp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_MIDM_I( M, N, A, LDA, C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_MIDM_I( M, N, KL, KU, A, LDA, C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_MIDM_I( UPLO, N, A, LDA, C, LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_MIDM_I( UPLO, N, K, A, LDA, C, LDC )
```

```
Symmetric Packed:
      SUBROUTINE BLAS_xSP_MIDM_I( UPLO, N, AP, CP )
Triangular:
      SUBROUTINE BLAS_xTR_MIDM_I( UPLO, TRANS, DIAG, N, A, LDA, C, LDC )
Triangular Band:
      SUBROUTINE BLAS_xTB_MIDM_I( UPLO, TRANS, DIAG, N, K, A, LDA, C, LDC )
Triangular Packed:
      SUBROUTINE BLAS_xTP_MIDM_I( UPLO, TRANS, DIAG, N, AP, CP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, LDC
      <type>             A( 2, LDA, * ) or AP( 2, * ), C( 2, LDA, * ) or
     $                   CP( 2, * )
```

- C binding:

```
General:
void BLAS_xge_midm_i( enum blas_order_type order, int m, int n,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
General Band:
void BLAS_xgb_midm_i( enum blas_order_type order, int m, int n, int kl, int ku,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
Symmetric:
void BLAS_xsy_midm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      int n, const <interval_array> a, int lda, RARRAY c,
                      int ldc );
Symmetric Band:
void BLAS_xsb_midm_i( enum blas_order_type order, int n, int k,
                      const <interval_array> a, int lda, RARRAY c, int ldc );
Symmetric Packed:
void BLAS_xsp_midm_i( enum blas_order_type order, int n,
                      const <interval_array> ap, RARRAY cp );
Triangular:
void BLAS_xtr_midm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Triangular Band:
void BLAS_xtb_midm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, int k, const <interval_array> a, int lda,
                      RARRAY c, int ldc );
Triangular Packed:
void BLAS_xtp_midm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                      enum blas_trans_type trans, enum blas_diag_type diag,
                      int n, const <interval_array> ap, RARRAY cp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_WIDTHM_I (Elementwise width of an interval matrix) $C \leftarrow \overline{A} - \underline{A}$

This routine finds the real matrix C such that $c_{i,j} = \overline{a}_{i,j} - \underline{a}_{i,j}$ $\forall i$ and $\forall j$, where $\mathbf{A} = \{\mathbf{a}_{i,j}\}$ is a general (or general banded, or symmetric, or symmetric banded, symmetric packed, or triangular, triangular banded, triangular packed) interval matrix.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_widthm_i( a, c )
General Band:
      SUBROUTINE gb_widthm_i( a, m, kl, c )
Symmetric:
      SUBROUTINE sy_widthm_i( a, c [, uplo] )
Symmetric Band:
      SUBROUTINE sb_widthm_i( a, kl, c [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_widthm_i( ap, cp [, uplo] )
Triangular:
      SUBROUTINE tr_widthm_i( a, c [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_widthm_i( a, kl, c [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_widthm_i( ap, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(IN) :: a(:,:), ap(:)
      REAL (<wp>), INTENT(OUT) :: c(:,:), cp(:)
      INTEGER, INTENT(IN) :: kl
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a and c have shape
           (m, n) for general matrix
           (l, n) for general banded matrix (l > kl)
           (n, n) for symmetric or triangular
           (p+1, n) for symmetric banded or triangular
                    banded (p = band width)
       ap and cp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_WIDTHM_I( M, N, A, LDA, C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_WIDTHM_I( M, N, KL, KU, A, LDA, C, LDC )
Symmetric:
      SUBROUTINE BLAS_xSY_WIDTHM_I( UPLO, N, A, LDA, C, LDC )
Symmetric Band:
```

```
      SUBROUTINE BLAS_xSB_WIDTHM_I( UPLO, N, K, A, LDA, C, LDC )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_WIDTHM_I( UPLO, N, AP, CP )
Triangular:
      SUBROUTINE BLAS_xTR_WIDTHM_I( UPLO, TRANS, DIAG, N, A, LDA, C, LDC )
Triangular Band:
      SUBROUTINE BLAS_xTB_WIDTHM_I( UPLO, TRANS, DIAG, N, K, A, LDA, C, LDC )
Triangular Packed:
      SUBROUTINE BLAS_xTP_WIDTHM_I( UPLO, TRANS, DIAG, N, AP, CP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, LDC
      <type>             A( 2, LDA, * ) or AP( 2, * ), C( LDC, * ) or
     $                   CP( * )
```

- C binding:

```
General:
void BLAS_xge_widthm_i( enum blas_order_type order, int m, int n,
                        const <interval_array> a, int lda, RARRAY c,
                        int ldc );
General Band:
void BLAS_xgb_widthm_i( enum blas_order_type order, int m, int n, int kl,
                        int ku, const <interval_array> a, int lda, RARRAY c,
                        int ldc );
Symmetric:
void BLAS_xsy_widthm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        int n, const <interval_array> a, int lda,
                        RARRAY c, int ldc );
Symmetric Band:
void BLAS_xsb_widthm_i( enum blas_order_type order, int n, int k,
                        const <interval_array> a, int lda, RARRAY c, int ldc );
Symmetric Packed:
void BLAS_xsp_widthm_i( enum blas_order_type order, int n,
                        const <interval_array> ap, RARRAY cp );
Triangular:
void BLAS_xtr_widthm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        enum blas_trans_type trans, enum blas_diag_type diag,
                        int n, const <interval_array> a, int lda,
                        RARRAY c, int ldc );
Triangular Band:
void BLAS_xtb_widthm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        enum blas_trans_type trans, enum blas_diag_type diag,
                        int n, int k, const <interval_array> a, int lda,
                        RARRAY c, int ldc );
Triangular Packed:
void BLAS_xtp_widthm_i( enum blas_order_type order, enum blas_uplo_type uplo,
                        enum blas_trans_type trans, enum blas_diag_type diag,
                        int n, const <interval_array> ap, RARRAY cp );
```

{GE,GB,SY,SB,SP,TR,TB,TP}_CONSTRUCTM_I (Constructs an interval matrix from two floating point matrices) $\mathbf{A} \supseteq B, C$

This routine constructs an interval matrix from two floating point matrices B and C such that $\mathbf{a}_{i,j} = [\min\{b_{i,j}, BLAS_{i,j}\}, \max\{b_{i,j}, BLAS_{i,j}\}]\ \forall i \in \{0, 1, \cdots, m-1\}$ and $\forall j \in \{0, 1, \cdots, n-1\}$. Both floating point matrices B and C have the same storage format.

- Fortran 95 binding:

```
General:
      SUBROUTINE ge_constructm_i( a, b, c )
General Band:
      SUBROUTINE gb_constructm_i( a, b, m, kl, c )
Symmetric:
      SUBROUTINE sy_constructm_i( a, b, c [, uplo] )
Symmetric Band:
      SUBROUTINE sb_constructm_i( a, b, kl, c [, uplo] )
Symmetric Packed:
      SUBROUTINE sp_constructm_i( ap, bp, cp [, uplo] )
Triangular:
      SUBROUTINE tr_constructm_i( a, b, c [, uplo] [, diag] )
Triangular Band:
      SUBROUTINE tb_constructm_i( a, b, kl, c [, uplo] [, diag] )
Triangular Packed:
      SUBROUTINE tp_constructm_i( ap, bp, cp [, uplo], [, diag] )
all:
      TYPE(INTERVAL) (<wp>), INTENT(OUT) :: a(:,:), ap(:)
      REAL (<wp>), INTENT(IN) :: b(:,:), c(:,:), cp(:)
      INTEGER, INTENT(IN) :: kl
      TYPE(blas_uplo_type), INTENT(IN), OPTIONAL :: uplo
      TYPE(blas_diag_type), INTENT(IN), OPTIONAL :: diag
    where
       a, b and c have shape
           (m, n) for general matrix
           (l, n) for general banded matrix (l > kl)
           (n, n) for symmetric or triangular
           (p+1, n) for symmetric banded or triangular
                    banded (p = band width)
       ap and cp have shape (n*(n+1)/2)
```

- Fortran 77 binding:

```
General:
      SUBROUTINE BLAS_xGE_CONSTRUCTM_I( M, N, A, LDA, B, LDB, C, LDC )
General Band:
      SUBROUTINE BLAS_xGB_CONSTRUCTM_I( M, N, KL, KU, A, LDA, B, LDB, C, LDC )
```

```
Symmetric:
      SUBROUTINE BLAS_xSY_CONSTRUCTM_I( UPLO, N, A, LDA, B, LDB, C, LDC )
Symmetric Band:
      SUBROUTINE BLAS_xSB_CONSTRUCTM_I( UPLO, N, K, A, LDA, B. LDB, C, LDC )
Symmetric Packed:
      SUBROUTINE BLAS_xSP_CONSTRUCTM_I( UPLO, N, AP, BP, CP )
Triangular:
      SUBROUTINE BLAS_xTR_CONSTRUCTM_I( UPLO, TRANS, DIAG, N, A, LDA,
     $                                  B, LDB, C, LDC )
Triangular Band:
      SUBROUTINE BLAS_xTB_CONSTRUCTM_I( UPLO, TRANS, DIAG, N, K, A, LDA,
     $                                  B, LDB, C, LDC )
Triangular Packed:
      SUBROUTINE BLAS_xTP_CONSTRUCTM_I( UPLO, TRANS, DIAG, N, AP, BP, CP )
all:
      INTEGER            UPLO, TRANS, DIAG, M, N, K, KL, KU, LDA, LDC
      <type>             A( 2, LDA, * ) or AP( 2, * ),  B( LDC, * )
     $                   or BP( * ), C( LDC, * ) or CP( * ),
```

- C binding:

```
General:
void BLAS_xge_constructm_i( enum blas_order_type order, int m, int n,
                            <interval_array> a, int lda, RARRAY b, int ldb
                            RARRAY c, int ldc );
General Band:
void BLAS_xgb_constructm_i( enum blas_order_type order, int m, int n, int kl,
                            int ku, <interval_array> a, int lda, RARRAY c,
                            int ldc );
Symmetric:
void BLAS_xsy_constructm_i( enum blas_order_type order,
                            enum blas_uplo_type uplo, int n,
                            <interval_array> a, int lda, RARRAY b,
                            int ldb, RARRAY c, int ldc );
Symmetric Band:
void BLAS_xsb_constructm_i( enum blas_order_type order, int n, int k,
                            <interval_array> a, int lda, RARRAY b, int ldb,
                            RARRAY c, int ldc );
Symmetric Packed:
void BLAS_xsp_constructm_i( enum blas_order_type order, int n,
                            <interval_array> ap, RARRAY bp, RARRAY cp );
Triangular:
void BLAS_xtr_constructm_i( enum blas_order_type order,
                            enum blas_uplo_type uplo,
                            enum blas_trans_type trans,
                            enum blas_diag_type diag, int n,
                            <interval_array> a, int lda, RARRAY b,
                            int ldb, RARRAY c, int ldc );
```

```
Triangular Band:
void BLAS_xtb_constructm_i( enum blas_order_type order,
                            enum blas_uplo_type uplo,
                            enum blas_trans_type trans,
                            enum blas_diag_type diag, int n, int k,
                            <interval_array> a, int lda, RARRAY b,
                            int ldb, RARRAY c, int ldc );
Triangular Packed:
void BLAS_xtp_constructm_i( enum blas_order_type order,
                            enum blas_uplo_type uplo,
                            enum blas_trans_type trans,
                            enum blas_diag_type diag, int n,
                            <interval_array> ap, RARRAY bp, RARRAY cp );
```

Environmental Enquiry

FPINFO_I (Environmental enquiry)

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

- Fortran 95 binding:

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

- Fortran 77 binding:

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

- C binding:

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