Chapter 3

Sparse BLAS

3.1 Overview

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

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

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

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

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

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

1. create an internal sparse matrix representation and return its handle (Sections 3.8.6, 3.8.7, and 3.8.8).
2. use this handle as a parameter in computational Sparse BLAS routines (Sections 3.8.2, 3.8.3, and 3.8.4).
3. when the matrix is no longer needed, call a cleanup routine to free resources associated with the handle (Section 3.8.10).

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

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

3.2 Naming Conventions

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

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

3.3 Functionality

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

3.3.1 Scalar and Vector Operations

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

Table 3.1: Sparse Vector Operations

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

3.3.2 Matrix-Vector Operations

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

Table 3.2: Sparse Matrix-Vector Operations

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

3.3.3 Matrix-Matrix Operations

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

Table 3.3: Sparse Matrix-Matrix Operations

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

3.4 Describing sparsity

3.4.1 Sparse Vectors

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

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

can be represented by two vectors as

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

although the permutation

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

or any other such permutation is equally valid.

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

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

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

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

3.4.2 Index bases

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

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

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

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

3.4.3 Repeated Indices

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

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

may be interpreted as a representation of the vector

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

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

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

3.5 Sparse BLAS Matrices

3.5.1 Creation Routines

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

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

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

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

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

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

Likewise, for variable block matrices, the declaration

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

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

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

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

3.5.2 Specifying matrix properties

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

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

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

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

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

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

3.5.3 Sparse Matrices: Inserting a Single Entry

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

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

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

3.5.4 Sparse Matrices: Inserting List of Entries

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

To illustrate this, consider the following matrix:

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

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

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

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

3.5.5 Sparse Matrices: Inserting Row and Column Vectors

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

3.5.6 Sparse Matrices: Inserting Cliques

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

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

and its associated row and column scattering vectors as

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

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

3.6 Interface Issues

3.6.1 Interface Issues for Fortran 95

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

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

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

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

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

PROGRAM F95_EX
USE blas_sparse

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

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

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

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

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

3.6.2 Interface Issues for Fortran 77

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

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

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

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

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

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

      END
```

3.6.3 Interface Issues for C

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

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

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

    #include "blas_sparse.h"

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

    blas_sparse_matrix A;
    int i;
    double alpha = 1.0;

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

    A = BLAS_duscr_begin( N, N );

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

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

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

    BLAS_uscr_end(A);

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

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

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

```
    BLAS_usds(A);

    return 0;
}
```

3.7 Numerical Accuracy and Environmental Enquiry

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

3.8 Language Bindings

3.8.1 Overview

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

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

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

3.8.2 Level 1 Computational Routines

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

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

Note that, as stated in Section 3.4.3, the result of a Level 1 BLAS operation called with repeated indices in array indx will be undefined. The actual return will be dependent on the implementation.

USDOT (Sparse dot product) $r \leftarrow x^T y$

The function USDOT computes the dot product of sparse vector x with dense vector y. The routine returns a real zero if the length of arrays x and indx are less than or equal to zero. When x and y are complex vectors, the vector components x_i are used unconjugated or conjugated as specified by the operator argument conj. If x and y are real vectors, the operator argument conj has no effect. For the C binding, the lack of a complex data type forces us to return the result in the parameter r.

- Fortran 95 binding:

```
<type>(<wp> FUNCTION usdot( x, indx, y [, conj] )
  INTEGER, INTENT(IN) :: indx(:)
  <type>(<wp>), INTENT(IN) :: x(:), y(:)
  TYPE(blas_conj_type), INTENT(IN), OPTIONAL :: conj
```

- Fortran 77 binding:

```
<type> FUNCTION BLAS_xUSDOT( CONJ, NZ, X, INDX, Y, INCY )
<type>          X( * ), Y( * )
INTEGER         NZ, INDX( * ), INCY
INTEGER         CONJ
```

- C binding:

```
void BLAS_xusdot( enum blas_conj_type conj, int nz, const ARRAY x,
                  const int *indx, const ARRAY y, int incy,
                  SCALAR_INOUT r, enum blas_base_type index_base );
```

USAXPY (Sparse vector update) $y \leftarrow \alpha x + y$

The routine USAXPY scales the sparse vector x by α and adds the result to the dense vector y. If the length of arrays x and indx are less than or equal to zero or if α is equal to zero, this routine returns without modifying y. Note that we do not allow a scaling on the vector y (that is, we do not implement a USAXPBY) as this would change the complexity of our routine because scaling a dense vector requires n operations while the sparse operations are only $O(nz)$. If the dense vector y is to be scaled, the appropriate Level 1 dense BLAS kernel should be used.

- Fortran 95 binding:

```
SUBROUTINE usaxpy( x, indx, y [, alpha] )
  <type>(<wp>), INTENT(IN) :: x(:)
  <type>(<wp>), INTENT(INOUT) :: y(:)
  INTEGER, INTENT(IN) :: indx(:)
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
```

The default value for α is `1.0`.

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSAXPY( NZ, ALPHA, X, INDX, Y, INCY )
<type>          ALPHA
<type>          X( * ), Y( * )
INTEGER         NZ, INDX( * ), INCY
```

- C binding:

```
void BLAS_xusaxpy( int nz, SCALAR_IN alpha, const ARRAY x, const int *indx,
                   ARRAY y, int incy, enum blas_base_type index_base );
```

USGA (Sparse gather into compressed form) $x \leftarrow y|_x$

Using indx to denote the list of indices of the sparse vector x, for each component i in this list, the routine USGA assigns x(i) = y(indx(i)). For example, if x is a sparse vector with nonzeros $\{3.1, 4.9\}$ and indices $\{1, 4\}$ (using 1-based offsets), and y is the dense vector $\{12.7, 68.1, 38.1, 54.0\}$, then the USGA routine changes x to $\{12.7, 54.0\}$. If the length of x and indx is non-positive, this routines returns without any modification to its parameters.

- Fortran 95 binding:

```
SUBROUTINE usga( y, x, indx )
  <type>(<wp>), INTENT(IN) :: y(:)
  <type>(<wp>), INTENT(OUT) :: x(:)
  INTEGER, INTENT(IN) :: indx(:)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSGA( NZ, Y, INCY, X, INDX )
INTEGER           NZ, INDX( * ), INCY
<type>            Y( * ), X( * )
```

- C binding:

```
void BLAS_xusga( int nz, const ARRAY y, int incy, ARRAY x, const int *indx,
                 enum blas_base_type index_base );
```

USGZ (Sparse gather and zero) $x \leftarrow y|_x, \;\; y|_x \leftarrow 0$

This routine combines two operations: (1) a sparse gather of y into x. (see USGA above), followed by (2) setting the corresponding values of y (y(indx(i)) to zero. For example, if x is a sparse vector with nonzeros $\{3.1, 4.9\}$ and indices $\{1, 4\}$ (using 1-based offsets), and y is the dense vector $\{12.7, 68.1, 38.1, 54.0\}$, then the USGA routine changes the nonzero values of x to $\{12.7, 54.0\}$ and changes y to $\{0.0, 68.1, 38.1, 0.0\}$.

- Fortran 95 binding:

```
SUBROUTINE usgz( y, x, indx )
  <type>(<wp>), INTENT(INOUT) :: y(:)
  <type>(<wp>), INTENT(OUT) :: x(:)
  INTEGER, INTENT(IN) :: indx(:)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSGZ( NZ, Y, INCY, X, INDX )
INTEGER           NZ, INDX( * ), INCY
<type>            Y( * ), X( * )
```

- C binding:

```
void BLAS_xusgz( int nz, ARRAY y, int incy, ARRAY x, const int *indx,
                 enum blas_base_type index_base );
```

USSC (Sparse scatter) $y|_x \leftarrow x$

This routine copies the nonzero values of x into the corresponding locations in the dense vector y. For example, if x is a sparse vector with nonzeros $\{3.1, 4.9\}$ and indices $\{1, 4\}$ (using 1-based offsets), and y is the dense vector $\{12.7, 68.1, 38.1, 54.0\}$, then the USSC routine changes y to $\{3.1, 68.1, 38.1, 4.9\}$. If the length of arrays x and indx are less than or equal to zero, this routine returns without any modification to its parameters.

- Fortran 95 binding:

```
SUBROUTINE ussc( x, y, indx )
  <type>(<wp>), INTENT(IN) :: x(:)
  <type>(<wp>), INTENT(INOUT) :: y(:)
  INTEGER, INTENT(IN) :: indx(:)
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSSC( NZ, X, Y, INCY, INDX )
INTEGER          NZ, INDX( * ), INCY
<type>           X( * ), Y( * )
```

- C binding:

```
void BLAS_xussc( int nz, const ARRAY x, ARRAY y, int incy, const int *indx,
                 enum blas_base_type index_base );
```

3.8.3 Level 2 Computational Routines

USMV (Sparse Matrix/Vector Multiply)

$$y \leftarrow \alpha\, Ax + y$$
$$y \leftarrow \alpha\, A^T x + y$$

This routine multiplies a dense vector x by a sparse matrix A (or its transpose), and adds it to the vector operand y. The matrix handle A must be valid, i.e. USGP(A, blas_valid_handle) must be true, and the precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE usmv( a, x, y, istat [, transa] [, alpha] )
  INTEGER, INTENT(IN) :: a
  <type>(<wp>), INTENT(IN) :: x(:)
  <type>(<wp>), INTENT(INOUT) :: y(:)
  INTEGER, INTENT(OUT) :: istat
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transa
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
```

Default values for transa and α are blas_no_trans and 1.0, respectively.

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSMV( TRANSA, ALPHA, A, X, INCX, Y, INCY, ISTAT )
INTEGER           INCX, INCY, A, TRANSA, ISTAT
<type>            ALPHA
<type>            X( * ), Y( * )
```

- C binding:

```
int BLAS_xusmv( enum blas_trans_type transa, SCALAR_IN alpha,
        blas_sparse_matrix A, const ARRAY x, int incx, ARRAY y, int incy );
```

USSV (Sparse Triangular Solve) $x \leftarrow \alpha\, T^{-1} x$
$x \leftarrow \alpha\, T^{-T} x$

This routine solves one of the systems of equations $x \leftarrow \alpha T^{-1} x$ or $x \leftarrow \alpha T^{-T} x$, where x is a dense vector and the matrix T is a triangular sparse matrix. The matrix handle T must be valid, i.e. USGP(T, blas_valid_handle) is true, must represent a valid triangular matrix, i.e. either USGP(T, blas_lower_triangular or USGP(T, blas_upper_triangular) must be true, and the precision type of the sparse matrix represented by the handle T must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE ussv( t, x, istat, [, transt] [, alpha] )
  INTEGER, INTENT(IN) :: t
  <type>(<wp>), INTENT(INOUT) :: x(:)
  INTEGER, INTENT(OUT) :: istat
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transt
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
```

Default values for transt and α are .TRUE. and 1.0 respectively.

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSSV( TRANST, ALPHA, T, X, INCX, ISTAT )
INTEGER           T, INCX, TRANST, ISTAT
<type>            ALPHA
<type>            X( * )
```

- C binding:

```
int BLAS_xussv( enum blas_trans_type transt, SCALAR_IN alpha,
        blas_sparse_matrix T, ARRAY x, int incx );
```

3.8.4 Level 3 Computational Routines

USMM (Sparse Matrix Multiply)

$$C \leftarrow \alpha\, AB + C$$
$$C \leftarrow \alpha\, A^T B + C$$

This routine multiplies a dense matrix B by a sparse matrix A (or its transpose), and adds it to a dense matrix operand C. A is of size m by n, B is of size of n by nrhs, and C is of size m by nrhs. The input argument nrhs must be greater than zero, and the matrix handle A must be valid, i.e. `USGP(A, blas_valid_handle)` must be true, and the precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE usmm( a, b, c, istat, [, transa] [, alpha] )
  INTEGER, INTENT(IN) :: a
  <type>(<wp>), INTENT(IN) :: b(:,:)
  <type>(<wp>), INTENT(INOUT) :: c(:,:)
  INTEGER, INTENT(OUT) :: istat
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transa
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
```

  Default values for `transa` and α are `.TRUE.` and `1.0`, respectively.

- Fortran 77 binding:

```
 SUBROUTINE BLAS_xUSMM( TRANSA, NRHS, ALPHA, A, B, LDB, C, LDC,
$                        ISTAT )
 INTEGER          NRHS, A, LDB, LDC, TRANSA, ISTAT
 <type>           ALPHA
 <type>           B( LDB, * ), C( LDC, * )
```

- C binding:

```
int BLAS_xusmm( enum blas_order_type order, enum blas_trans_type transa,
                 int nrhs, SCALAR_IN alpha, blas_sparse_matrix A,
                 const ARRAY B, int ldb, ARRAY C, int ldc );
```

USSM (Sparse Triangular Solve)

$$B \leftarrow \alpha\, T^{-1} B$$
$$B \leftarrow \alpha\, T^{-T} B$$

This routine solves one of the systems of equations $B \leftarrow \alpha T^{-1} B$ or $B \leftarrow \alpha T^{-T} B$, where B is a dense matrix and T is a triangular sparse matrix. T is of size n by n, B is of size of n by nrhs, and C is of size n by nrhs. The input argument nrhs must be greater than zero, and the matrix handle T must be valid, i.e. `USGP(T, blas_valid_handle)` must be true, and represent a valid triangular matrix, i.e. either `USGP(T, blas_lower_triangular` or `USGP(T, blas_upper_triangular)` must be true. The precision type of the sparse matrix represented by the handle T must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE ussm( t, b, istat [, transt] [, alpha] )
  INTEGER, INTENT(IN) :: t
  <type>(<wp>), INTENT(INOUT) :: b(:,:)
  INTEGER, INTENT(OUT) :: istat
  TYPE (blas_trans_type), INTENT(IN), OPTIONAL :: transt
  <type>(<wp>), INTENT(IN), OPTIONAL :: alpha
```

  Default values for `transt` and α are `.TRUE.` and `1.0` respectively.

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSSM( TRANST, NRHS, ALPHA, T, B, LDB, ISTAT )
INTEGER          NRHS, T, LDB, TRANST, ISTAT
<type>           ALPHA
<type>           B( LDB, * )
```

- C binding:

```
int BLAS_xussm( enum blas_order_type order, enum blas_trans_type transt,
          int nrhs, SCALAR_IN alpha, blas_sparse_matrix T, ARRAY B, int ldb
);
```

3.8.5 Handle Management

The Handle Management routines can be divided into five sets; the creation routines (Section 3.8.6), the insertion routines (Section 3.8.7), the completion routine (Section 3.8.8), matrix property routines (Section 3.8.9), and the destruction routine (Section 3.8.10). A brief discussion of these routines was given in Section 3.5.1.

3.8.6 Creation Routines

USCR_BEGIN (begin point-entry construction) $A \leftarrow (...)$

USCR_BEGIN is used to create a sparse matrix handle where the matrix is held in normal point-wise form (by single scalar entries). m and n must be greater than zero. The `x` prefix in the binding names specifies the scalar type and precision of the matrix, as described in 3.2. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns a new handle as its function return value; this handle is void, i.e. `USGP(return_value, blas_void_handle)` is true, if the routine did not execute successfully.

- Fortran 95 binding:

```
SUBROUTINE xuscr_begin( m, n, a, istat )
  INTEGER, INTENT(IN) :: m, n
  INTEGER, INTENT(OUT) :: a, istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSCR_BEGIN( M, N, A, ISTAT )
INTEGER           M, N, A, ISTAT
```

- C binding:

```
blas_sparse_matrix BLAS_xuscr_begin( int m, int n );
```

USCR_BLOCK_BEGIN (begin constant block-entry construction) $A \leftarrow (...)$

USCR_BLOCK_BEGIN is used to create a sparse matrix handle referring to a block-entry matrix where the blocksize of all entries is constant, that is block entries are $k \times l$. Mb, Nb, k and l must all be greater than zero. The `x` prefix in the binding names specifies the scalar type and precision of the matrix, as described in 3.2. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns a new handle as its function return value; this handle is void, i.e. `USGP(return_value, blas_void_handle)` is true, if the routine did not execute successfully.

- Fortran 95 binding:

```
SUBROUTINE xuscr_block_begin( Mb, Nb, k, l, a, istat )
  INTEGER, INTENT(IN) :: Mb, Nb, k, l
  INTEGER, INTENT(OUT) :: a, istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSCR_BLOCK_BEGIN( MB, NB, K, L, A, ISTAT )
INTEGER           MB, NB, K, L, A, ISTAT
```

- C binding:

```
blas_sparse_matrix BLAS_xuscr_block_begin( int Mb, int Nb, int k, int l );
```

USCR_VARIABLE_BLOCK_BEGIN (begin variable block-entry construction) $A \leftarrow (...)$

USCR_VARIABLE_BLOCK_BEGIN is used to create a sparse matrix handle referring to a block-entry matrix whose entries may have variable block sizes. The blocksizes are given by the integer arrays K and L such that the dimension of the (i, j) block entry is $K(i) \times L(j)$. Mb, Nb, and all elements of K and L must be greater than zero. The `x` prefix in the binding names specifies the scalar type and precision of the matrix, as described in 3.2. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns a new handle as its function return value; this handle is void, i.e. `USGP(return_value, blas_void_handle)` is true, if the routine did not execute successfully.

- Fortran 95 binding:

```
SUBROUTINE xuscr_variable_block_begin( Mb, Nb, k, l, a, istat )
  INTEGER, INTENT(IN) :: Mb, Nb, k(:), l(:)
  INTEGER, INTENT(OUT) :: a, istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSCR_VARIABLE_BLOCK_BEGIN( MB, NB, K, L, A, ISTAT )
INTEGER           MB, NB, A, ISTAT
INTEGER           K( * ), L( * )
```

- C binding:

```
blas_sparse_matrix BLAS_xuscr_variable_block_begin( int Mb, int Nb,
                                                    const int *k,
                                                    const int *l );
```

3.8.7 Insertion routines

USCR_INSERT_ENTRY (insert single value at coordinate (i, j)) $A \leftarrow (val, i, j)$

USCR_INSERT_ENTRY is used to build a sparse matrix, passing in one scalar entry at a time. This routine may only be called on a matrix handle that was opened via the USCR_BEGIN routine and has not yet been closed via the USCR_END routine. Furthermore, matrix properties cannot be modified after any insertions, so this call must follow all settings made to the matrix via the USSP routine. The matrix handle must be in a new state (i.e `USPG(A, blas_new_handle`) is true) upon the first call to this routine. Upon successful completion, the matrix handle is an open state (i.e. `USGP(A, blas_open_handle)` is true) and subsequent calls to this routine will keep the matrix in this state, until a call to `USCR_END` is issued. The precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE uscr_insert_entry( a, val, i, j, istat )
  INTEGER, INTENT(IN) :: a, i, j
  <type>(<wp>), INTENT(IN) :: val
  INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSCR_INSERT_ENTRY ( A, VAL, I, J, ISTAT )
INTEGER           A, I, J, ISTAT
<type>            VAL
```

- C binding:

```
int BLAS_xuscr_insert( blas_sparse_matrix A, SCALAR val, int i, int j );
```

USCR_INSERT_ENTRIES (insert a list of values in coordinate form (val, i, j)) $A \leftarrow (val, i, j)$

USCR_INSERT_ENTRIES is used to build a sparse matrix, passing in a list of point entries. This routine may only be called on a matrix handle that was opened via the USCR_BEGIN routine and has not yet been closed via the USCR_END routine. Furthermore, matrix properties cannot be modified after any insertions, so this call must follow all settings made to the matrix via the `USSP` routine. The matrix handle must be in a new state (i.e `USPG(A, blas_new_handle)` is true) upon the first call to this routine. Upon successful completion, the matrix handle is an open state (i.e. `USGP(A, blas_open_handle)` is true) and subsequent calls to this routine will keep the matrix in this state, until a call to `USCR_END` is issued. The precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE uscr_insert_entries( a, val, indx, jndx, istat )
  INTEGER, INTENT(IN) :: a, indx( : ), jndx( : )
  <type>(<wp>), INTENT(IN) :: val ( : )
  INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xUSCR_INSERT_ENTRIES( A, NZ, VAL, INDX, JNDX,
     $                                      ISTAT )
      INTEGER          A, NZ, INDX( * ), JNDX( * ), ISTAT
      <type>           VAL( * )
```

- C binding:

```
int BLAS_xuscr_insert_entries( blas_sparse_matrix A, int nz,
                               const ARRAY val,
                               const int *indx, const int *jndx );
```

USCR_INSERT_COL (insert a compressed column) $A \leftarrow (...)$

USCR_INSERT_COL is used to build a sparse matrix, passing in one column at a time. This routine may only be called on a matrix handle that was opened via the USCR_BEGIN routine and has not yet been closed via the USCR_END routine. Furthermore, matrix properties cannot be modified after any insertions, so this call must follow all settings made to the matrix via the `USSP` routine. The matrix handle must be in a new state (i.e `USPG(A, blas_new_handle)` is true) upon the first call to this routine. Upon successful completion, the matrix handle is an open state (i.e. `USGP(A, blas_open_handle)` is true) and subsequent calls to this routine will keep the matrix in this state, until a call to `USCR_END` is issued. The precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE uscr_insert_col( a, j, val, indx, istat )
  INTEGER, INTENT(IN) ::  a, j, indx(:)
  <type>(<wp>), INTENT(IN) ::  val(:)
  INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSCR_INSERT_COL( A, J, NZ, VAL, INDX, ISTAT )
INTEGER          A, J, NZ, INDX( * ), ISTAT
<type>           VAL( * )
```

- C binding:

```
int BLAS_xuscr_insert_col( blas_sparse_matrix A, int j, int nz,
                           const ARRAY val, const int *indx );
```

USCR_INSERT_ROW (insert a compressed row) $A \leftarrow (...)$

USCR_INSERT_ROW is used to build a sparse matrix, passing in one row at a time. This routine may only be called on a matrix handle that was opened via the USCR_BEGIN routine and has not yet been closed via the USCR_END routine. Furthermore, matrix properties cannot be modified after any insertions, so this call must follow all settings made to the matrix via the `USSP` routine. The matrix handle must be in a new state (i.e `USPG(A, blas_new_handle)` is true) upon the first call to this routine. Upon successful completion, the matrix handle is an open state (i.e. `USGP(A, blas_open_handle)` is true) and subsequent calls to this routine will keep the matrix in this state, until a call to `USCR_END` is issued. The precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE uscr_insert_row( a, i, val, indx, istat )
  INTEGER, INTENT(IN) ::  a, i, indx(:)
  <type>(<wp>), INTENT(IN) ::  val(:)
  INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_xUSCR_INSERT_ROW( A, I, NZ, VAL, INDX, ISTAT )
INTEGER          A, I, NZ, INDX( * ), ISTAT
<type>           VAL( * )
```

- C binding:

```
int BLAS_xuscr_insert_row( blas_sparse_matrix A, int i, int nz,
                           const ARRAY val, const int *indx );
```

USCR_INSERT_CLIQUE (insert a dense matrix clique) $A \leftarrow (val, i, j)$

USCR_INSERT_CLIQUE is used to build a sparse matrix, passing in a dense matrix val of dimension $\mathsf{k} \times \mathsf{l}$ and corresponding integer arrays containing the list of (i, j) indices describing the clique. This routine may only be called on a matrix handle that was opened via the USCR_BEGIN routine and has not yet been closed via the USCR_END routine. Furthermore, matrix properties cannot be modified after any insertions, so this call must follow all settings made to the matrix via the `USSP` routine. The matrix handle must be in a new state (i.e `USPG(A, blas_new_handle)` is true) upon the first call to this routine. Upon successful completion, the matrix handle is an open state (i.e. `USGP(A, blas_open_handle)` is true) and subsequent calls to this routine will keep the matrix in this state, until a call to `USCR_END` is issued. The precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE uscr_insert_clique( a, val, indx, jndx, istat )
  INTEGER, INTENT(IN) :: a, indx(:), jndx(:)
  <type>(<wp>), INTENT(IN) :: val(:,:)
  INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_xUSCR_INSERT_CLIQUE( A, K, L, VAL, LDV, INDX,
     $                                     JNDX, ISTAT )
      INTEGER          A, K, L, LDV, INDX( * ), JNDX( * ), ISTAT
      <type>           VAL( LDV, * )
```

- C binding:

```
int BLAS_xuscr_insert_clique( blas_sparse_matrix A, const int k,
                              const int l, const ARRAY val,
                              const int row_stride, const int col_stride,
                              const int *indx,
                              const int *jndx );
```

USCR_INSERT_BLOCK (insert a block entry at block coordinate (bi, bj)) $A \leftarrow (val, bi, bj)$

USCR_INSERT_BLOCK is used to insert a block entry into a block-entry matrix. This routine may only be called on a matrix handle that was opened with one of the block creation routines (USCR_BLOCK_BEGIN or USCR_VARIABLE_BLOCK_BEGIN) and has not yet been closed via the USCR_END routine. Furthermore, matrix properties cannot be modified after any insertions, so this call must follow all settings made to the matrix via the USSP routine. The matrix handle must be in a new state (i.e USPG(A, blas_new_handle) is true) upon the first call to this routine. Upon successful completion, the matrix handle is an open state (i.e. USGP(A, blas_open_handle) is true) and subsequent calls to this routine will keep the matrix in this state, until a call to USCR_END is issued. The dimensions of the block entry are determined from the blocksize information passed to USCR_BLOCK_BEGIN or USCR_VARIABLE_BLOCK_BEGIN. In the Fortran 77 binding, LDV denotes the leading dimension of the dense array VAL. The precision type of the sparse matrix represented by the handle A must match the remaining floating-point arguments. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE uscr_insert_block( a, val, bi, bj, istat )
  INTEGER, INTENT(IN) :: a, bi, bj
  INTEGER, INTENT(OUT) :: istat
  <type>(<wp>), INTENT(IN) :: val(:,:)
```

- Fortran 77 binding:

```
SUBROUTINE F_xUSCR_INSERT_BLOCK( A, VAL, LDV, BI, BJ, ISTAT )
INTEGER          A, LDV, BI, BJ, ISTAT
<type>           VAL( LDV, * )
```

- C binding:

```
int BLAS_xuscr_insert_block( int a, const ARRAY val, int row_stride,
                             int col_stride, int bi, int bj );
```

3.8.8 Completion of construction routine

USCR_END (entries completed; build valid matrix handle) $A \leftarrow (...)$

USCR_END is used to complete the construction phase and build a valid sparse matrix handle. This routine may be called only with a sparse matrix handle that was previously created via the routines USCR_BEGIN, USCR_BLOCK_BEGIN or USCR_VARIABLE_BLOCK_BEGIN. The matrix handle must be in an open or new state, i.e. either USGP(A, blas_open_handle) or USGP(A, blas_new_handle) is true. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE uscr_end( a, istat )
  INTEGER, INTENT(IN) :: a
  INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_USCR_END( A, ISTAT )
      INTEGER           A, ISTAT
```

- C binding:

```
int BLAS_uscr_end( blas_sparse_matrix A );
```

3.8.9 Matrix property routines

USGP (get/test matrix property) *property-value*← A

For a given sparse matrix A, the routine USGP returns the value of the given property name. The first argument is the matrix handle and the second argument is one of the properties listed in in Table 3.5. Each grouping denotes a subset of mutually exclusive properties. The properties `blas_num_rows`, `blas_num_cols`, and `blas_num_nonzeros` return integer values, all other properties return 1 if true, and 0 otherwise. If the matrix handle is corrupt, i.e. `USGP(A, blas_void_handle)` is true, all other Boolean properties are false, and integer valued properties (`blas_num_rows`, `blas_num_cols`, and `blas_num_nonzeros`) return 0.

- Fortran 95 binding:

```
      SUBROUTINE usgp( a, pname, m )
        INTEGER, INTENT(IN) :: a
        INTEGER, INTENT(IN) :: pname
        INTEGER, INTENT(OUT) :: m
```

- Fortran 77 binding:

```
      SUBROUTINE BLAS_USGP( A, PNAME, M )
      INTEGER           A, PNAME, M
```

- C binding:

```
int BLAS_usgp( blas_sparse_matrix A, int pname );
```

USSP (set matrix property) A ← *property-value*

For a given valid sparse matrix handle A, the routine USSP sets the value of the given matrix property. This routine must be called after the handle has been created, and before any of the INSERT routines have been called. That is, the matrix handle must be in a new state, i.e. `USGP(A, blas_new_handle)` is true. istat is used as an error flag and will be zero if the routine executes successfully and is set to -1 if the handle is corrupt, i.e. if `USGP(A, blas_void_handle)` is true. The C binding returns istat as the function return value.

The first argument is the matrix handle; the second argument is one of the properties listed in in Table 3.4. Each grouping denotes a subset of mutually exclusive properties.

If two incompatible properties from the same group are set, the results are undefined. For example, the sequence

```
BLAS_ussp(A, blas_zero_base);
BLAS_ussp(A, blas_one_base);
```

leads to an ambiguity and the resulting handle is void (i.e. `USGP(A, blas_void_handle)` is true). It is possible to guard against this by testing the properties first.

- Fortran 95 binding:

```
SUBROUTINE ussp( a, pname, istat )
INTEGER, INTENT(INOUT) :: a
INTEGER, INTENT(IN) :: pname
INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_USSP( A, PNAME, ISTAT )
INTEGER           A, PNAME, ISTAT
```

- C binding:

```
int BLAS_ussp( blas_sparse_matrix A, int pname );
```

3.8.10 Destruction routine

USDS (release matrix handle) $(...) \leftarrow A$

The routine USDS releases any memory internally used by the sparse matrix handle A. The handle must have been previously closed by the `USCR_END` routine, i.e. `USGP(A, blas_valid_handle)` must be true. It turns this into a handle that is no longer in use, i.e. `USGP(A, blas_void_handle)` is true. istat is used as an error flag and will be zero if the routine executes successfully. The C binding returns istat as the function return value.

- Fortran 95 binding:

```
SUBROUTINE usds( a, istat )
INTEGER, INTENT(IN) :: a
INTEGER, INTENT(OUT) :: istat
```

- Fortran 77 binding:

```
SUBROUTINE BLAS_USDS( A, ISTAT )
INTEGER           A, ISTAT
```

- C binding:

```
int BLAS_usds( blas_sparse_matrix A );
```
