The Sparse BLAS[1]

Iain S. Duff[2], Michael A. Heroux[3], and Roldan Pozo[4]

ABSTRACT

We discuss the interface design for the Sparse Basic Linear Algebra Subprograms (BLAS), the kernels in the recent standard from the BLAS Technical Forum that are concerned with unstructured sparse matrices. The motivation for such a standard is to encourage portable programming while allowing for library-specific optimizations. In particular, we show how this interface can shield one from concern over the specific storage scheme for the sparse matrix. This design makes it easy to add further functionality to the sparse BLAS in the future.
We illustrate the use of the Sparse BLAS with examples in the three supported programming languages, Fortran 95, Fortran 77, and C.

Keywords: sparse BLAS, sparse matrices, sparse iterative methods, computational kernels, algorithms, software.

AMS(MOS) subject classifications: 65F05, 65F50.

---

[1]Current reports available by anonymous ftp to ftp.numerical.rl.ac.uk in directory pub/reports. This report is available in compressed postscript as file duRAL2001032.ps.gz or as the PDF file duRAL2001032.pdf. Report also available through URL http://www.numerical.rl.ac.uk/reports/reports.html.
Also published as Technical Report TR/PA/01/24 from CERFACS, 42 Ave G. Coriolis, 31057 Toulouse Cedex, France.
[2]i.s.duff@rl.ac.uk, Rutherford Appleton Laboratory, Chilton, Didcot, Oxon OX11 9LN, England. The work of this author was supported in part by the EPSRC Grant GR/M78502.
[3]mheroux@cs.sandia.gov, Sandia National Laboratories, Albuquerque, NM 87185-1110.
[4]pozo@nist.gov, National Institute of Standards and Technology, Gaithersburg, MD 20899.

Computational Science and Engineering Department
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

September 10, 2001

Contents

1 Introduction **1**

2 Handle-based generic interface **2**

3 Functionality **3**

4 Sparse vector representation **5**

5 Sparse matrix management **5**
5.1 Matrix construction 6
5.1.1 Creating a matrix handle 6
5.1.2 Asserting matrix properties 8
5.1.3 Inserting matrix values 9
5.1.4 Ending Insertion 11
5.2 Using Sparse BLAS matrices 11
5.3 Destroying a Sparse BLAS matrix 11
5.4 Power method example 12

6 Error handling and matrix properties **12**

7 Interface issues **13**
7.1 Interface issues for Fortran 95 14
7.2 Interface issues for Fortran 77 15
7.3 Interface issues for ANSI C 18

8 Conclusions **20**

1 Introduction

In this paper, we consider the interface design of the Basic Linear Algebra Subprograms for sparse matrices: the *Sparse BLAS*. This work is part of the effort of the BLAS Technical Forum and appears as Chapter 3 in the standard specification [1].

A matrix is termed *sparse* if many of its entries are zero. However, for this to be a meaningful distinction, it is also necessary that some advantage can be taken of this fact, either for storage or operations with the matrix, or both. Clearly band matrices are sparse, but this proposal is concerned with sparse matrices that may have much more irregular sparsity patterns. Although the dense BLAS [7, 8] have support for some sparse matrices, specifically, banded and packed matrices, and LAPACK [2] supports tridiagonal matrices, neither library provides functionality for unstructured sparse matrices. The omission of sparse matrix support is not because sparse matrices are uncommon, or because sparse matrix operations cannot be optimized. In fact, the opposite is true. Sparse matrices are ubiquitous in many applications in science, engineering and commerce. For example, the discretization of partial differential equations by finite elements or finite differences gives rise to a sparse matrix, as do representations of structures and networks. Also, numerous studies have shown that careful attention to the implementation of sparse matrix operations, for example sparse matrix-vector multiplication, can have a very significant impact, often improving performance by an order of magnitude.

Unlike regularly structured sparse and dense matrices, whose data access patterns are known *a priori*, data access patterns for unstructured sparse matrices are known only after the matrix has been constructed. Thus, static scheduling techniques that are the foundation for high-performance dense BLAS operations are ineffective for unstructured sparse computations. At the same time, sparse matrix operations, for example in the context of an iterative solution method, are often called repeatedly with the same data access pattern. Therefore it is sensible to spend some time computing the most efficient data access patterns for a given sparse matrix and a common operation such as matrix-vector multiplication. Performance improvements for sparse matrix operations typically come from reordering data access to increase the degree of fine grain parallelism [4], or improving cache performance [15, 16]. The Sparse BLAS standard allows complete freedom for selecting the data access pattern because the standard does not prescribe a data structure for storing matrix entries and uses a matrix construction process that enables analysis of data access patterns.

Unfortunately, almost every application area has a different way of storing and accessing the nonzero entries of the sparse matrix so that it is difficult to design software to apply to all areas. In this paper, we will discuss in detail how the Sparse BLAS interface addresses these problems via a generic handle-based representation.

As with the other BLAS, the Sparse BLAS are designed with more than one

objective in mind. In this case, the main constituency are people writing and using iterative methods for large sparse system solution or the eigenproblem.

The Sparse BLAS have been discussed for many years, even before the BLAS Technical Forum started (see, for example, the introduction in [10]) so that they do represent a good consensus of what is required by the community. The current specification represents a small, accepted list of operations which may grow over time. Indeed, we hope that a feature of the Sparse BLAS is that they lend themselves easily to future extensions.

Computational kernels in the Sparse BLAS standard are grouped in three major categories, similar to the other BLAS interfaces. Level 1 describes vector and scalar operations, Level 2 describes operations involving a sparse matrix and dense vector, and Level 3 describes operations involving a sparse matrix and a dense (typically tall and thin) matrix.

We discuss our handle-based generic interface in Section 2 and describe the functionality that we support in Section 3. We consider the representation of sparse vectors in Section 4, the management of handles and sparse matrices in Section 5, and comment on our error handling and use of the matrix property enquiry routines in Section 6. We consider the interface to Fortran 77, C, and Fortran 95 programming languages in Section 7, giving short examples illustrating the use of each interface, and make some concluding remarks in Section 8. Finally, we illustrate the use of the Sparse BLAS in a more realistic application in the Appendix.

2 Handle-based generic interface

One important difference between this and the other BLAS interfaces in the standard is the support for generic or abstract matrix representations. That is, Level 2 and Level 3 operations take as input not the matrix entries themselves, but rather a pointer, or a *handle* to a generic representation to a previously created sparse matrix object. This allows algorithms to be coded using Sparse BLAS primitives without requiring the underlying matrix storage format. The result is more general code that can be run under different situations and storage optimization strategies without modification. This means that the entry data needs to be converted and held internally within an efficient storage representation. The internal representation, however, is determined by the BLAS implementation and is not specified by the standard. Library developers are then able to create highly optimized versions of these kernels for differing computer architectures or specific application areas.

This strong reliance on referencing matrices solely by their matrix handle represents a significant departure from earlier efforts on the sparse BLAS (for example, [5, 10, 17]) and is quite different from our earlier drafts for a BLAS standard for sparse matrices where some of the more common data structures were explicitly presented. Of course, our current approach is not without its dangers. Far more power is placed in the hands of the library developer. For example, if

an implementation utilizes an internal global table to manage matrix handles, care must be taken if a threadsafe implementation is required.

Managing these handles typically requires a three-step process. First, a new BLAS sparse matrix is created and initialized with values; it can then be used in subsequent Level 2 and Level 3 computational kernels; finally, when that matrix is no longer needed, it can be freed to reclaim computer memory and associated resources (see Section 5).

The Sparse BLAS standard does not impose a limit on the number of sparse matrix handles that can exist at the same time. That number is constrained by the available memory and other computer resources which will vary on different platforms. It is the intent of the standard, however that, when a sparse matrix handle is released, all resources associated with the matrix are also freed.

3 Functionality

As in the case of the dense BLAS, we have developed the Sparse BLAS to support Level 1, 2, and 3 operations. We present in Tables 3.1, 3.2, and 3.3, the functionality supported by the Sparse BLAS at these three levels respectively. Before we discuss the functionality further, we first define some of the symbols used in the tables. For the Level 1 BLAS, y represents a dense vector while x represents a vector in packed form (as detailed in Section 4). The symbol $y|_x$ selects from the dense vector y those components which are referenced by the integer array associated with x. r and α are scalars. For the Level 2 BLAS, both the vectors x and y represent dense vectors. A and T are always sparse matrices, with T being triangular; B and C are dense matrices. The notation Z^T, Z^H and Z^{-1} refer to the transpose, Hermitian transpose and inverse of a matrix Z, respectively.

The functionality reflects the belief in the main purposes for which the Sparse BLAS will be used combined with a desire to keep the set of kernels as small as possible, both to encourage their use and their support by the vendors. The Sparse BLAS has been designed in a flexible way so that more functionality can be easily added later.

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: Level 1 Sparse BLAS: sparse vector operations.

The first thing to note in the tables is that not only is much less functionality supported than in the dense case but some of the kernels are simpler than in the

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

Table 3.2: Level 2 Sparse BLAS: sparse matrix-vector 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: Level 3 Sparse BLAS: sparse matrix-matrix operations.

dense case. For example, the kernel for matrix-vector multiplication in the dense case, GEMV, is of the form $y \leftarrow \alpha Ax + \beta y$, where β is a scalar. The sparse verion of this, USMV, does not have a scalar β (that is, it is assumed to have value 1.0). The reason for this is that, in the sparse case, the scaling operation by β is an $O(n)$ operation for vectors and an $O(n^2)$ operation on matrices so that this operation would often dominate the amount of work performed on the sparse data structures. Since these operations represent scaling on dense vectors and matrices, if they are needed in an application, they can be handled separately using appropriate routines from the dense BLAS.

We note that all Level 2 and Level 3 operations involve the product (or inverse product) of a sparse matrix with a dense vector or matrix. We do not support operations on two sparse structures. One reason for this is that of complexity, another is that of potentially mixing different representations for the two sparse structures. These are generally deemed too complicated for low-level kernels. For example, merely determining the fill-in for a sparse matrix - sparse matrix multiplication requires complicated graph analysis and is beyond the scope of a low-level computational kernel. In addition, it is often computationally more efficient to avoid such operations. For example, when forming the product ABx, where A and B are sparse matrices and x a dense vector, it is usually far better to compute this as $A * (B * x)$ rather than $(A * B) * x$. Furthermore, we know of vendors who have already optimized Sparse BLAS in our current functionality, but we know of no cases where this optimization has been extended to the case of two sparse structures. This does not mean that such a facility is never requested. For example, algebraic multi-level methods (see for example Tong and Tuminaro [18]) often form a coarse operator via explicit construction of $A_c = RA_fP$, where A_f, P and R are given

fine grid, prolongation and restriction operators, respectively. The paper of Bank and Douglas [3] describes a kernel for use in a similar situation. Thus, there are situations where such kernels could be useful, and indeed a sparse-sparse framework may become an extension of this standard in the future.

As mentioned earlier, the principal constituency for this interface are those developing and using iterative methods to solve sparse linear equations and we believe that our kernels address most of the needs of this community. We have not attempted to address the often more complicated kernels in sparse direct methods because these methods use a far wider range of data structures (not just for storage but at the innermost loop). For many years, some proponents of direct methods have argued that the correct kernels in this case should be based on the dense BLAS [9] and many sparse direct approaches have been designed to exploit this including frontal [12] and multifrontal [11] techniques.

4 Sparse vector representation

Sparse vectors are represented in Level 1 operations by two separate arrays: one with the real or complex nonzero entries, and the other an integer array holding their respective offsets[1]. Thus, the vector $\{0.0, 4.3, 0.0, 2.1, 1.9, 0.0\}$ can be represented in packed form as two arrays $\{4.3, 2.1, 1.9\}$ and $\{2, 4, 5\}$. We do not specify the ordering of the entries so there are several equivalent storage representations for the same sparse vector. The *index base* (that is whether one starts counting at zero or one) is part of the *semantics* of the underlying representation and should be explicitly stated. In our example, we listed the elements in increasing index order and have used the Fortran convention of starting with an index of 1 for the first component.

The standard does not allow repeated indices in sparse vectors, thus calling a Level 1 routine with such a representation results in undefined behavior. (Note that repeated indices in sparse matrices *are* allowed, since the structures are constructed internally by the BLAS framework. See Section 5.1.1.)

5 Sparse matrix management

The Sparse BLAS standard promotes an object-oriented approach to the construction and use of sparse matrices. The construction process uses opaque interfaces in the sense that, as the user inputs matrix entries, the entries are copied to internal storage, but how the entries will be stored is not specified. In practice, the matrix storage will depend on the nonzero structure of the matrix, the properties that the user sets (if any), and the type of computer system being used.

[1]Several other representations, such as an array of C structures or a Fortran 95 derived data type holding nonzero and index values, were examined but the separate array scheme can be easily represented in all language bindings and is one of the most commonly used formats.

Once created, a Sparse BLAS matrix is referenced by its *handle*, the equivalent of an object instance in object-oriented programming terminology. We note that the actual handle value will be held as an integer, for portability across language implementations. In the remainder of this section we will discuss the three basic phases of constructing, using, and destroying a Sparse BLAS matrix.

5.1 Matrix construction

Constructing a Sparse BLAS matrix requires three or four basic steps:

- create the matrix handle,
- set matrix properties (optional),
- insert matrix values, and then
- end insertion.

In this subsection we discuss the details of this construction.

5.1.1 Creating a matrix handle

There are Sparse BLAS matrix handles for three types of data, corresponding to the three types of matrix entries that we support. Note that, as soon as the handle is initialized, all subsequent references and operations on the matrix referenced by that handle must be to the same matrix type as originally defined. The three types of matrix are:

1. Point entry matrices.
   The entries of this type of matrix are simple scalar values. The sparsity structure describes the layout of these entries in the matrix.

2. Block entry matrices with constant block size.
   Entries in these matrices are dense submatrices (blocks), each with the same row and column dimension. Block entry matrices have three different types of dimensions. The first is the number of block rows and columns, the second is the number of row and columns in each block, and the third is the number of equations and variables (standard row and column dimensions, respectively). Of course, any one of the types of dimensions can be obtained from knowing the other two types.

3. Block entry matrices with varying block sizes.
   Entries in these matrices are also dense submatrices, but block sizes may vary from block row to block row. In this case, both the number of blocks and their dimensions must be specified.

Block entry matrices occur naturally for problems where multiple degrees of freedom are associated with a single physical location. For example, in a fluid dynamics calculation, we may track the x, y and z direction velocities at a single mesh point. Variable block entry matrices occur when some degrees of freedom are tracked in one subregion and not another, for example chemical species concentration. Our definition of block entries enforces a two-dimensional partitioning of the matrix as illustrated in Figure 5.1 for a 4×4 matrix. The first matrix in this figure has no partitioning and corresponds to a point entry matrix. In this case, each nonzero value is stored as a separate entry. The middle version has the same values, but we impose a 2×2 block partitioning. The rightmost version has a variable partition size that creates a 3×3 block matrix whose block entry sizes vary.

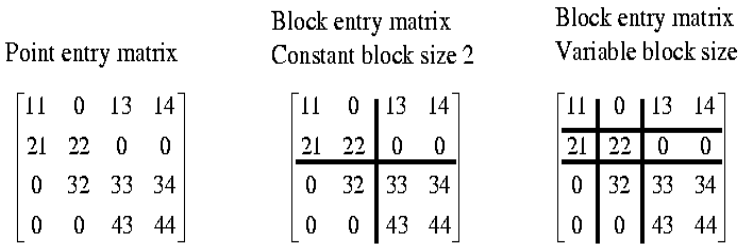

Figure 5.1: Three possible types of entries for a matrix.

When a matrix has an underlying block structure, it is often advantageous to exploit this structure for the following reasons.

1. Integer storage and arithmetic costs are substantially reduced.
   Integer storage and computational complexity is reduced by a factor equal to the square of the average block entry dimension. For example, if the average block entry dimension is 3×3, the integer storage and computation is reduced by a factor of 9.

2. Computations involving dense matrices often perform well on modern processors.
   Block entries have a predictable memory access pattern that can be exploited to increase instruction pipelining, latency hiding, and cache memory hits. Furthermore, if the block entry dimensions are large enough, dense BLAS kernels can be used. If the block entry dimensions are at least 10, a very large fraction of peak performance can be obtained for Sparse BLAS computations.

3. Block entry matrices enable hybrid sparse-dense algorithms.
   Block entry matrices provide an explicit hierarchy that can be exploited by two-level algorithms. For example, block Gauss-Seidel or block Jacobi

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 in row major order (default for C)
blas_colmajor	dense block stored in column 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 5.4: Matrix properties.

algorithms [6] can be used such that the outer algorithm is performed using the graph of the matrix and the inner algorithm exploits the dense block entries.

5.1.2 Asserting matrix properties

Once a matrix handle is created, it is possible to assert certain matrix properties that may have an impact on performance or storage, or may be required for a matrix to be used with certain operations. Table 5.4 lists the properties that can be set for a given matrix handle.

The pair of parameters `blas_non_unit_diag`, `blas_unit_diag` asserts whether or not the matrix diagonal values are assumed to be one, in which case the user will not insert them into the matrix. If a matrix is unit triangular and will be used for triangular solves, asserting `blas_unit_diag` can significantly reduce both storage and computation.

The pair `blas_no_repeated_indices`, `blas_repeated_indices` asserts whether or not some matrix entries may be listed more than once. If indices are repeated, all values will be summed into a single entry. Asserting `blas_no_repeated_indices` can permit more efficient matrix construction. Asserting `blas_repeated_indices` allows for correct assembly of matrix entries

for finite-element computations and related methods.

The parameters `blas_lower_symmetric`, `blas_upper_symmetric`, `blas_lower_hermitian`, and `blas_lower_symmetric` allow users to store only the upper or lower triangle of a symmetric or Hermitian matrix and still have the matrix considered as square rather than triangular.

Parameters `blas_zero_base`, and `blas_one_base` allow the user to declare that indices start at zero (C-style indexing) or one (Fortran-style indexing). `blas_rowmajor` and `blas_colmajor` indicate whether dense blocks, as might be used in the block storage schemes, are held in row major or column major order, respectively.

Finally, the parameters `blas_irregular`, `blas_regular`, `blas_block_irregular`, `blas_block_regular`, and `blas_unassembled` allow the user to describe possible regularity in the structure. This could be used if the matrix were known to come from a regular grid or other structured data. These five parameters are intended to provide optimization hints and are completely optional; their omission should not affect program correctness.

5.1.3 Inserting matrix values

Once matrix properties (if any) have been set, we can insert values into the matrix in a variety of ways. For a point entry matrix, insertion of matrix values is always in terms of a single entry or a collection of entries. Entries can be inserted using the following five methods.

1. Single entry: a single matrix entry together with its row and column index.

2. List of entries: in this case a list of values with corresponding row and column indices is submitted for insertion.

3. List of row entries: a list of values with corresponding column indices is submitted for insertion in a specified row.

4. List of column entries: a list of values with corresponding row indices is submitted for insertion in a specified column.

5. Clique: a two-dimensional array of values with two integer arrays that describe the row and column indices associated with the rows and columns of the two-dimensional array. Such a data structure is appropriate for finite-element computations. The example below illustrates this method.

The five different methods of inserting matrix entries may be used in any combination. If insertion is done more than once for any matrix entry and `blas_repeated_indices` is enabled, repeated values will be summed into a single value for such an entry.

Consider the following matrix:

$$A = \begin{pmatrix} 1.1 & 0 & 1.3 & 0 \\ 0 & 2.2 & 0 & 2.4 \\ 3.1 & 0 & 3.3 & 0 \\ 0 & 4.2 & 0 & 4.4 \end{pmatrix}. \tag{5.1}$$

We can pass in all entries (following a call to the handle initialization routines) by defining a list of eight entries with the values, row indices and column indices as follows:

$$\begin{aligned} \text{values} &= (\ 1.1\ \ 1.3\ \ 2.2\ \ 2.4\ \ 3.1\ \ 3.3\ \ 4.2\ \ 4.4\ ) \\ \text{row indices} &= (\ 1\ \ 1\ \ 2\ \ 2\ \ 3\ \ 3\ \ 4\ \ 4\ ) \\ \text{column indices} &= (\ 1\ \ 3\ \ 2\ \ 4\ \ 1\ \ 3\ \ 2\ \ 4\ ). \end{aligned}$$

Note that calls to the C interface would default to using 0-based indices. The ordering of the entries is arbitrary. Also, any partial list of the above entries could be passed, as long as all entries were eventually submitted. We can build A row-by-row (or column-by-column) in the obvious way by submitting the entries corresponding to each row (or column) one row (column) at a time.

As an alternative to the above insertion methods, the entries of A can be inserted by cliques in the following way. First submit a two-dimensional array (whose major ordering depends on the matrix properties as defined in Table 5.4) as:

$$\text{values} = \begin{pmatrix} 1.1 & 1.3 \\ 3.1 & 3.3 \end{pmatrix} \tag{5.2}$$

and two associated index arrays:

$$\begin{aligned} \text{row indices} &= (\ 1\ \ 3\ ) \\ \text{column indices} &= (\ 1\ \ 3\ ). \end{aligned}$$

To complete the insertion, we submit the array

$$\text{values} = \begin{pmatrix} 2.2 & 2.4 \\ 4.2 & 4.4 \end{pmatrix} \tag{5.3}$$

and its associated row and column scattering vectors

$$\begin{aligned} \text{row indices} &= (\ 2\ \ 4\ ) \\ \text{column indices} &= (\ 2\ \ 4\ ). \end{aligned}$$

Insertion for a block entry matrix must be performed using the block entry insertion routine. This routine accepts a single block as a two-dimensional array of values and a corresponding block row and block column index. The two-dimensional array will be summed into the existing matrix. The exact definition of the two-dimensional array depends on the language binding. Defining data structures for lists of block entries was deemed to be too complicated and language dependent, and also unnecessary since the block entry already passes in several matrix values and therefore amortizes the overhead of the call.

`blas_num_rows`	returns the number of rows of the matrix
`blas_num_cols`	returns the number of columns of the 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_single_precision`	matrix values are single precision
`blas_double_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_void_handle`	handle not currently in use
`blas_new_handle`	before any items have been inserted
`blas_open_handle`	after the first item has been inserted
`blas_valid_handle`	after the `USCR_END` routine has been called

Table 5.5: Matrix properties that can be queried.

5.1.4 Ending Insertion

Once all matrix entries have been input, we signal the end of insertion by calling a routine that will process all entries with the intent of reorganizing the storage of the matrix and performing any preprocessing that will optimize the performance of operations using this matrix. The exact process performed by calling the end insertion routine is very much implementation dependent.

5.2 Using Sparse BLAS matrices

Once a Sparse BLAS matrix handle has been completely constructed (something that can be tested by checking the property `blas_valid_handle`), it is possible to use the matrix handle to perform operations. At this time the four operations shown in Tables 3.2 and 3.3 are supported.

In addition to performing operations with a Sparse BLAS matrix, it is possible to query its properties through its handle. Table 5.5 lists the properties that can be obtained by calling the get properties routine.

5.3 Destroying a Sparse BLAS matrix

When a Sparse BLAS matrix is no longer needed, it may be destroyed (and all associated storage returned to the operating system) by passing the matrix handle

to the Sparse BLAS destructor routine.

5.4 Power method example

We present an example showing how the Sparse and dense BLAS can be used to implement the power method for finding the dominant eigenvalue and corresponding eigenvector of a matrix [14]. Additional examples showing how to construct and use a Sparse BLAS matrix with each of the language interfaces (Fortran 77, C, and Fortran 95) can be found in Section 7. The matrix used for all examples is the following 4×4 sparse matrix

$$A = \begin{pmatrix} 1.1 & 0 & 0 & 0 \\ 0 & 2.2 & 0 & 2.4 \\ 0 & 0 & 3.3 & 0 \\ 4.1 & 0 & 0 & 4.4 \end{pmatrix}. \tag{5.4}$$

Code for the power method example for each of the supported programming languages is given in the appendix.

6 Error handling and matrix properties

The Sparse BLAS have an error return scheme consistent with the error-handling framework described in Section 1.8 of the BLAS Standard [1]. In particular, most housekeeping routines related to managing sparse matrix handles utilize return codes to signify the success or failure of the operation. Furthermore, the Level 2 and Level 3 computational kernels employ a similar mechanism. Typically, these kernels return a value of 0 (zero) if the operation completed successfully, for example

```
if (BLAS_duscr_insert_entry(A, 2.1, 2, 3) != 0)
    /* matrix entry was not successfully included */ ;
```

The handle-management routines use return codes rather than aborting the program, since these requests (such as constructing matrices, inserting elements, setting matrix properties) usually make use of queries about available resources, such as computer memory, and their failure needs to be recoverable.

Query routines interrogate the status or properties of a handle. These properties are listed in Table 5.5. A few query routines return integer values, denoting the various matrix size properties, for example

In ANSI C,

```
M = BLAS_usgp(A, blas_num_rows);
N = BLAS_usgp(A, blas_num_cols);
nz = BLAS_usgp(A, blas_num_nonzeros);
```

or, in Fortran 77,

```
CALL BLAS_USGP(A, BLAS_NUM_ROWS, M)
CALL BLAS_USGP(A, BLAS_NUM_COLS, N)
CALL BLAS_USGP(A, BLAS_NUM_NONZEROS, NZ)
```

or, in Fortran 95,

```
call usgp(a, blas_num_rows, m)
call usgp(a, blas_num_cols, n)
call usgp(a, blas_num_nonzeros, nz)
```

The majority of matrix properties, however, are Boolean values, such as whether a matrix contains complex values, or whether it is symmetric or triangular. In such cases, the values returned are either 1 (true) or 0 (false). These can be used to form *guarded* statements, which can often reduce potential errors of calling Sparse BLAS kernels with inappropriate arguments. For example, to avoid calling a triangular solve operation on a non-triangular matrix in ANSI C, one can write,

```
if (BLAS_usgp(A, blas_lower_triangular))
    BLAS_ussv(blas_no_transp, alpha, A, x, incx);
```

or, in Fortran 95,

```
call usgp(A,blas_lower_triangular,m)
if (m.eq.1) call ussv(A, x, istat, transt, alpha)
```

7 Interface issues

The Sparse BLAS supports Fortran 95, Fortran 77, and C and interfaces for the purpose of avoiding the necessity of interlanguage calls. In addition to specific function interfaces for each language, header files (modules for Fortran 95) of pre-defined named constants and (where appropriate) function prototypes are also provided. The purpose of the pre-defined named constants is to make code more readable and portable. All constants are compatible with constants used by the dense BLAS. In the remainder of this section, we discuss some details specific to each language interface, and then give a simple example.

The functionality of the Sparse BLAS is nearly identical for each language binding, except for one specific case: the Level 1 (vector) operations in C allow the use for both 0-based and 1-based offsets. This feature is not available in the Fortran bindings.[2]

[2]This decision was made by the BLAS Technical Forum for consistency with pre-existing Fortran Level 1 Sparse BLAS.

7.1 Interface issues for Fortran 95

Predefined constants for the Sparse BLAS are included in the module `blas_sparse_namedconstants` that is included in the module `blas_sparse`. These include the sparse matrix properties constants defined in Tables 5.4 and 5.5. In addition, the module `blas_sparse` provides explicit interfaces to all routines.

The interface example below illustrates multiplying the 4×4 matrix from Section 5.4 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.

A separate implementation report [13] describes the details of a reference implementation of the Sparse BLAS in Fortran 95.

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

      PROGRAM F95_EX

      USE blas_sparse

      IMPLICIT NONE
      INTEGER, 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/)
      x=1.0D0
      y=0.0D0

      n = nmax
!
!     ---------------------------------
!     Step 1:  Create Sparse BLAS Handle
!     ---------------------------------
!
!     Note that the matrix handle is just the integer variable a
!
      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)
!
!     ------------------------
!     Step 6:  Output solution
!     ------------------------
!
      WRITE(6,'(5D16.8)') y

      END
```

7.2 Interface issues for Fortran 77

Although Fortran 77 is no longer a current 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 (Appendix B of [1]) 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.

Naming conventions for the this interface prefix `BLAS_x` to the root name, where `x` can be one of (`F`, `D`, `C`, `Z`) representing real single-precision, real double-precision, complex single-precision, and complex double-precision, respectively. For example, the complex single-precision version of a sparse dot product is `BLAS_CUSDOT`. Some routines for supporting generic handles are not floating-point type specific and do not require an `x` specifier in the prefix. The routine for retrieving handle properties, `BLAS_USGP`, is one such example. (These are clearly listed in the standard documentation [1].)

Predefined constants for the Sparse BLAS are included in the header file `blas_sparse_namedconstants.h`. These include the sparse matrix properties constants defined in Tables 5.4 and 5.5.

The following program illustrates the use of Fortran 77 code on the 4×4 matrix from Section 5.4. We note that some non-standard features are present in this example: the use of `IMPLICIT NONE`, the `INCLUDE` statement, and long identifiers for variables and subroutine names. However, these extensions are permitted by all the Fortran 77 compilers of which we have experience.

```
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)
C
C     ------------------------
C     Step 6:  Output solution
C     ------------------------
C
      WRITE(6,*)(Y(I),I=1,4)

      END
```

7.3 Interface issues for ANSI C

Predefined constants for the Sparse BLAS are included in the header file `blas_enum.h`, which is included as part of `blas_sparse.h`. These include the sparse matrix properties constants defined in Tables 5.4 and 5.5.

In a manner similar to the Fortran 77 interface, the naming conventions in C prefix `BLAS_x` to the root name, where `x` can be one of (`f`, `d`, `c`, `z`) representing real single-precision, real double-precision, complex single-precision, and complex double-precision, respectively. Some routines for supporting generic handles are not floating-point type specific and do not require an `x` specifier in the prefix, and these are clearly listed in the standard documentation [1].

Level 2 and Level 3 routines that utilize matrix handles assume an index base of 0, consistent with C conventions for array indexing. This value can be overridden by specifying `blas_one_base` at the time of creation of the matrix handle.

C sparse matrix handles are integers, but are `typedef` to `blas_sparse_matrix` for clarity.

The following program illustrates the use of C code on the 4×4 matrix from Section 5.4.

```
    /* 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,   4,   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;
    double alpha = 1.0;
    int i;

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

    A = BLAS_duscr_begin( n, n );
```

```
    /*-----------------------------------*/
    /*  Step 2: insert entries one-by-one */
    /*-----------------------------------*/

    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);

    /*--------------------------*/
    /* Step 6:  Output Solution  */
    /*--------------------------*/

    for (i=0; i<n; i++) printf("%12.4g ",y[i]);
    printf("\n");
    return 0;
}
```

8 Conclusions

The Sparse BLAS specification allows for portable, high-performance sparse matrix - vector kernels that can be application or hardware specific. By not specifying the internal format of the sparse matrices, library developers and application programmers are able to avoid the complicated issues of sparse matrix data structures. In addition, this enables the implementor of a Sparse BLAS library to choose whatever data structure is appropriate for efficient implementation for the target application and machine.

This is managed through the use of a handle to the matrix rather than specifying the specific components of the underlying data structure. We have described this handle interface in some detail and given examples of how it can be used in practice.

Like the previous BLAS specifications, we hope that this initiative promotes the exchange of portable numerical codes, while permitting programmers, commercial library developers, and computer vendors to design optimized and efficient code.

Acknowledgements

We would like to thank John Reid and Jennifer Scott of the Rutherford Appleton Laboratory for their helpful comments on an early draft of this text.

The Sparse BLAS Standard which this paper describes was not solely the work of the authors, but was part of a large collaborative effort of the BLAS Technical Forum (BLAST) and we would like take this opportunity to thank all of its members who actively participated in this process. In particular, several individuals whose ideas and efforts had significant impact included Guangye Li and Karin Remington, who independently developed early implementations of the Sparse BLAS kernels in C and Fortran 77 for public distribution; Galen Wilkerson, who provided an early test suite for verification and validation of the computational routines; Christof Vömel (assisted by Marcelin Youan), who developed the Fortran 95 reference implementation of the Sparse BLAS; Anthony Skjellum and Andrew Lumsdaine, who championed early on for a generic handle-based interface to the Sparse BLAS; Susan Blackford, who developed the C and f77 Sparse BLAS header files, and tirelessly edited the standard document through multiple authors and numerous revisions.

References

[1] BLAS Technical Forum Standard. *The International Journal of High Performance Computing Applications*, 15(3–4), 2001.

[2] E. Anderson, Z. Bai, C. H. Bischof, J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. C.

Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, 2nd edition, 1995. (Also available in Japanese, published by Maruzen, Tokyo, translated by Dr Oguni).

[3] R. E. Bank and C. C. Douglas. Sparse matrix multiplication package (SMMP). *Advances in Computational Mathematics*, 1:127–137, 1993.

[4] G. Blelloch, M. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical Report CMU-CS-93-173, Carnegie Mellon University, Pittsburgh, Pennsylvania, Aug 1993.

[5] Sandra Carney, Michael A. Heroux, Guangye Li, and Kesheng Wu. A revised proposal for a sparse BLAS toolkit. Technical Report 94-034, Army High Performance Computing Research Center, June 1994. Updated version at Web address `http://www.cray.com/products/applications/support/scal/spblastk.ps`.

[6] E. Chow and M. Heroux. An object-oriented framework for block preconditioning. *ACM Transactions on Mathematical Software*, 24(2):159–183, June 1998.

[7] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16:1–28, 1990. (Algorithm 679).

[8] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 14:1–32, 399, 1988. (Algorithm 656).

[9] I. S. Duff. Full matrix techniques in sparse Gaussian elimination. In G. A. Watson, editor, *Numerical Analysis Proceedings, Dundee 1981*, Lecture Notes in Mathematics 912, pages 71–84, Berlin, 1981. Springer-Verlag.

[10] I. S. Duff, M. Marrone, G. Radicati, and C. Vittoli. Level 3 Basic Linear Algebra Subprograms for sparse matrices: a user level interface. *ACM Trans. Math. Softw.*, 23(3):379–401, 1997.

[11] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 9:302–325, 1983.

[12] I. S. Duff and J. A. Scott. The design of a new frontal code for solving sparse unsymmetric systems. *ACM Trans. Math. Softw.*, 22(1):30–45, 1996.

[13] I. S. Duff and C. Vömel. The implementation of the Sparse BLAS in Fortran 95. Technical Report TR/PA/01/xx, CERFACS, Toulouse, France, 2000. Submitted to *ACM Trans. Math. Softw.*

[14] G. Golub and C. Van Loan. *Matrix Computations*. Johns-Hopkins, Baltimore, Third edition, 1996.

[15] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Performance modeling and tuning of an unstructured mesh CFD application. Technical Report ANL/MCS-P833-0700, Argonne National Laboratory, Illinois, July 2000.

[16] J. Hu. *Cache Based Multigrid on Unstructured Grids in Two and Three Dimensions*. PhD thesis, University of Kentucky, Lexington, Kentucky, 2000.

[17] K. A. Remington and R. Pozo. NIST Sparse BLAS User's Guide. Internal Report NISTIR 6744, National Institute of Standards and Technology, Gaithersburg, MD, USA, May 2001.

[18] C. Tong and R. Tuminaro. ML 2.0 smoothed aggregation user's guide. Technical Report SAND2001-8028, Sandia National Laboratories, December 2000.

Appendix

In this appendix we present codes in Fortran 77, C, and Fortran 95 that implement the example on the use of the Power method described in Section 5.4. All codes have been compiled and executed. The dominant eigenvalue is 4.4.

```
------------------- File: power_method_driver.f -------------------------

C     Fortran 77 example:  Build sparse matrix and call power method


      PROGRAM MAIN
      IMPLICIT NONE
C     Header file of prototypes and named constants
      INCLUDE "blas_namedconstants.h"
      INTEGER NMAX, NNZ
      PARAMETER (NMAX = 4, NNZ = 6)
      INTEGER I, N, ISTAT
      INTEGER INDX(NNZ), JNDX(NNZ)
      DOUBLE PRECISION VAL(NNZ), X(NMAX), Y(NMAX)
      INTEGER A
      INTEGER NITERS
      DOUBLE PRECISION Q(NMAX), WORK(NMAX), LAMBDA
C
C     ---------------------------------------------
C     Define Matrix, LHS and RHS in Coordinate format
C     ---------------------------------------------
C
      DATA VAL   / 1.1D0, 2.2D0, 2.4D0, 3.3D0, 4.1D0, 4.4D0/
      DATA INDX  /   1,   2,   2,   3,   4,   4/
      DATA JNDX  /   1,   2,   4,   3,   1,   4/
C
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 all at once
C     ----------------------------------
C
        CALL BLAS_DUSCR_INSERT_ENTRIES(A, NNZ, VAL, INDX, JNDX, ISTAT)
C
C     ------------------------------------------------
C     Step 3:  Complete construction of sparse matrix
C     ------------------------------------------------
C
      CALL BLAS_USCR_END(A, ISTAT)

C
C     ------------------------------------------------
C     Step 4:  Call Power Method Routine
C     ------------------------------------------------
C

C     Q      - eigenvector approximation.
C     LAMBDA - eigenvalue approximation.

      NITERS = 100

      CALL POWER_METHOD(A, Q, LAMBDA, NITERS, WORK, ISTAT)
      IF (ISTAT.NE.0) PRINT*,'Error in POWER_METHOD = ',ISTAT

      PRINT*,'Number of iterations = ',NITERS
      PRINT*,'Approximate dominant eigenvalue = ', LAMBDA

C
C     ------------------------------------------------
C     Step 5:  Release Matrix Handle
C     ------------------------------------------------
C
      CALL BLAS_USDS(A);

      END

------------------- File: power_method.f --------------------------------

C     Apply power method to given matrix.
```

```
      SUBROUTINE POWER_METHOD(A, Q, LAMBDA, NITERS, Z, ISTAT)
      IMPLICIT NONE
      INCLUDE "blas_namedconstants.h"
      INTEGER N
      INTEGER A
      DOUBLE PRECISION Q(*)
      DOUBLE PRECISION LAMBDA
      INTEGER NITERS
      DOUBLE PRECISION Z(*), NORMZ
      INTEGER ISTAT
      INTEGER I, ITER
      INTEGER BLAS_USGP
      EXTERNAL BLAS_USGP

C     Get row dimension of A (assume square)
      N = BLAS_USGP(A, BLAS_NUM_ROWS)

C     Fill Z.
C     Note:  We should fill Z with random numbers, but no standard
C            random function is available for Fortran 77.
      DO 10 I = 1, N
         Z(I) = I
 10   CONTINUE

      DO 20 ITER = 1, NITERS
C        Compute 2-norm of Z
         CALL BLAS_DNORM(BLAS_TWO_NORM, N, Z, 1, NORMZ)
C        Copy Z to Q
         CALL BLAS_DCOPY(N, Z, 1, Q, 1)
C        Normalize Q
         CALL BLAS_DRSCALE(N, NORMZ, Q, 1)
C        Compute new Z
         CALL BLAS_DUSMV(BLAS_NO_TRANS, 1.0, A, Q, 1, Z, 1, ISTAT)
C        Test error flag
         IF (ISTAT.NE.0) RETURN
C        New LAMBDA
         CALL BLAS_DDOT(BLAS_NO_CONJ, N, 1.0, Q, 1, 0.0, Z, 1, LAMBDA)
 20   CONTINUE

      RETURN

      END
```

```
------------------- File: power_method_driver.c -------------------------

/* C example:  Build sparse matrix and call power method */

#include "blas_sparse.h" /* Header file of prototypes and named constants */

int main()
{
    const int N = 4;
    const int nnz = 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,   4,   2,   0,   3};

    blas_sparse_matrix A;
    int istat = 0;
    int niters;
    double * q, lambda;

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

    A = BLAS_duscr_begin( N, N );


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

    BLAS_duscr_insert_entries(A, nnz, val, indx, jndx);

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

    BLAS_uscr_end(A);


    /*-------------------------------------------------*/
    /* Step 4:  Call Power Method Routine              */
```

```
    /*-----------------------------------------------*/

    /*.
      q      - eigenvector approximation (allocated by power_method).
      lambda - eigenvalue approximation.
    */

    niters = 100;

    istat = power_method( A, q, &lambda, niters);

    printf("Approximate dominant eigenvalue after %d iterations = %10.4g\n",
           niters, lambda);

    /*-----------------------------------------------*/
    /* Step 5:  Release Matrix Handle, Eigenvector  */
    /*-----------------------------------------------*/

    free ((void *) q);
    BLAS_usds(A);

     return istat;
}
------------------- File: power_method.c --------------------------------

/* Apply power method to given matrix. */
#include <stdlib.h> /* Needed for malloc */
#include "blas_sparse.h"/* Header file of prototypes and named constants */
#include "blas_dense.h"/* Header file of prototypes and named constants */

int power_method(blas_sparse_matrix A, double * q, double * lambda, int niters)
{
    int istat = 0;
    int i, n, iter;
    double * z, normz;

    n = BLAS_usgp(A, blas_num_rows); /* Get row dimension of A (assume square) */

    q = (double *) malloc(sizeof(double)*n); /* Allocate eigenvector, work vectors
    z = (double *) malloc(sizeof(double)*n);

    /* Fill z with random numbers using rand() function */
```

```
    for (i=0; i < n; i++) z[i] = ((double) rand())/ ((double) RAND_MAX);


  for (iter = 0; iter < niters; iter++)
    {
      BLAS_dnorm(blas_two_norm, n, z, 1, &normz); /* Compute 2-norm of z */
      BLAS_dcopy(n, z, 1, q, 1); /* Copy z to q */
      BLAS_drscale(n, normz, q, 1); /* Normalize q */
      istat = BLAS_dusmv( blas_no_trans, 1.0, A, q, 1, z, 1 ); /* Compute new z */
      if (istat!=0) return(istat); /* Test error flag */
      BLAS_ddot(blas_no_conj, n, 1.0, q, 1, 0.0, z, 1, lambda); /* new lambda */
    }

  free((void *) z); /* release work vector */

  return istat;
}


------------------- File: power_method_driver.f95 --------------------------

!     Fortran 95 example:  Build sparse matrix and call power method


      PROGRAM MAIN
!     Header file of prototypes and named constants
      USE blas_sparse
      INTEGER, PARAMETER :: nmax = 4, nnz = 6
      INTEGER i, n, istat
      INTEGER, DIMENSION(:), ALLOCATABLE :: indx,jndx
      DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: val,q,work
      INTEGER a,niters
      DOUBLE PRECISION lambda

      ALLOCATE (val(nnz),indx(nnz),jndx(nnz))
      ALLOCATE (q(nmax),work(nmax))
!
!     ----------------------------------
!     Define matrix, in coordinate format
!     ----------------------------------
!
      val  = (/ 1.10D0, 2.20D0, 2.40D0, 3.30D0, 4.10D0, 4.40D0/)
      indx = (/   1,   2,   2,   3,   4,   4/)
```

```
      jndx = (/   1,   2,   4,   3,   1,   4/)
!
!
      n = nmax
!
!     ---------------------------------
!     Step 1:  Create Sparse BLAS handle
!     ---------------------------------
!
      CALL duscr_begin( n, n, a, istat)

!
!     ---------------------------------
!     Step 2:  Insert entries all at once
!     ---------------------------------
!
        CALL uscr_insert_entries(a, val, indx, jndx, istat)
!
!     ------------------------------------------------
!     Step 3:  Complete construction of sparse matrix
!     ------------------------------------------------
!
      CALL uscr_end(a, istat)

!
!     ------------------------------------------------
!     Step 4:  Call Power Method Routine
!     ------------------------------------------------
!

!     q      - eigenvector approximation.
!     lambda - eigenvalue approximation.

      niters = 100

      CALL POWER_METHOD(a, q, lambda, n, niters, work, istat)
      IF (istat.NE.0) THEN
        WRITE(*,*) 'Error in POWER_METHOD = ',istat
      ELSE
        WRITE(*,*) 'Number of iterations = ',niters
        WRITE(*,*) 'Approximate dominant eigenvalue = ', lambda
      ENDIF
```

```
!
!     ------------------------------------------
!     Step 5:  Release Matrix Handle
!     ------------------------------------------
!
      CALL usds(a,istat)

      END PROGRAM MAIN


------------------- File: power_method.f95 ---------------------------------


!     Apply power method to given matrix.

      SUBROUTINE POWER_METHOD(a, q, lambda, n, niters, z, istat)
      USE blas_sparse
      INTEGER, INTENT(IN) :: a,n,niters
      DOUBLE PRECISION, DIMENSION(:), INTENT(OUT) :: q(n),z(n)
      DOUBLE PRECISION, INTENT(OUT) :: lambda
      INTEGER, INTENT(OUT) :: istat
      INTEGER i, iter
      DOUBLE PRECISION normz, dnrm2
      REAL y
      EXTERNAL dnrm2
      INTRINSIC RANDOM_NUMBER,DOT_PRODUCT

!     Fill Z with random numbers
      DO I = 1, n
         CALL RANDOM_NUMBER(HARVEST=y)
         z(I) = DBLE(y)
      END DO

      DO iter = 1, niters
!        Compute 2-norm of z
         normz = DNRM2(n, z, 1)
!     Normalize z
         CALL DSCAL(n, 1.0D0/normz, z, 1)
!        Copy Z to Q
         q = z
! Set z to 0
```

```
         z = 0.0D0
!        Compute new z
         CALL usmv(a, q, z, istat)
!        Test error flag
         IF (istat.NE.0) RETURN
!        New LAMBDA
         LAMBDA = DOT_PRODUCT(q,z)
      END DO

      RETURN

      END
```