RAL-TR-95-049

A set of Level 3 Basic Linear Algebra Subprograms for sparse matrices[1]

Iain S. Duff, Michele Marrone[2], Giuseppe Radicati[2], and Carlo Vittoli[2]

ABSTRACT
This paper proposes a set of Level 3 Basic Linear Algebra Subprograms and associated kernels for sparse matrices. We discuss the design, implementation and use of subprograms for the multiplication of a full matrix by a sparse one and for the solution of sparse triangular systems with one or more (full) right-hand sides. We include routines for checking the input data, generating a new sparse data structure from that input, and scaling a sparse matrix. The new data structure for the transformation can be specified by the user or can be chosen automatically by vendors to be efficient on their machines. We also include routines for permuting the columns of a sparse matrix and one for permuting the rows of a full matrix. A major aim is to establish standards to enable efficient, and portable, implementations of iterative algorithms for sparse matrices on high-performance computers. We have designed the routines so that the developer of mathematical software need not be concerned with the complexities of the various data structures used for sparse matrices. We have kept the interface and suite of codes as simple as possible while at the same time including sufficient functionality to cover most of the requirements of iterative solvers, and sufficient flexibility to cover most sparse matrix data structures. Fortran 77 code implementing this proposal is available by anonymous ftp.

Keywords: sparse matrices, sparse data structures, high-performance computing, programming standards, sparse BLAS, iterative solution.

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

---

[1] Current reports available by anonymous ftp from seamus.cc.rl.ac.uk (internet 130.246.8.32) in the directory "pub/reports". This report is in file dmrvRAL95049.ps.gz.

[2] IBM, Italy.

Computing and Information Systems Department
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

September 11, 1995.

Contents

1 Introduction

We are proposing a standard for a set of Level 3 Basic Linear Algebra Subprograms and associated kernels for sparse matrices. This set includes Level 3 sparse BLAS for sparse-full matrix multiplication and sparse triangular solutions. We also include routines for permuting full and sparse matrices and a routine for data checking, for changing the data format of a sparse matrix, and for performing a diagonal scaling on the input matrix. We anticipate that these will form the basis for writing mathematical software for implementing iterative methods on sparse matrices.

We are quite intentionally not overly ambitious in this proposal because we feel it is important to get a standard established in the very near future before different manufacturers develop their own different methods of implementing the functions we seek to standardize.

In developing programs for sparse matrices, the choice of the data structure used to represent the non-zero coefficients of the matrix plays a crucial role. There are several contrasting requirements that guide this choice. Ordinarily, the data structure is chosen to limit the number of zeros stored and to avoid unnecessary calculations with zero values during subsequent numerical calculations. Additionally, the data structure must allow the software developer to take advantage of any regularity present in the sparsity pattern. Finally, the data structure may be chosen so that the software can exploit hardware features, such as vector registers or parallel processing capabilities. As a consequence of these somewhat contrasting requirements, a great many different data structures are used for sparse matrices.

The primary user community that we are targeting consists of developers of library software although we recognize that the software could and should be used as building blocks by applications programmers. We expect a sophisticated user community but not necessarily one that is or need be familiar with details of sparse storage schemes. We have designed the routines so that the numerical library software developer need not be concerned with the complexities of the various data structures used for sparse matrices. We have established a single simple interface that will accommodate most of the data formats in use today and have provided the ability to transform between formats without the need to know their details explicitly. The application programmer would provide the input matrix in a supported format nearest to the natural one for the particular application and would normally request that a transformation be made to the format provided by the vendor for the target machine. We expect that this transformation will be provided by vendors of high-performance computers. The cost in moving or rearranging the data, and storage, is in many cases more than offset by the gain in performance. Sometimes such rearranging is done explicitly as in the routine sparse_matvec_setup in CMSSL (CMSSL 1992), and sometimes it is implicit as in the IBM Library ESSL (ESSL 1990). We envisage that a major benefit of the provision of this interface will be a much quicker implementation of new algorithmic ideas into complex applications packages.

Standard computational kernels have been proposed for basic linear algebra operations on full matrices. These Basic Linear Algebra Subprograms include routines for vector operations such as a scalar product (Level 1 BLAS, Lawson, Hanson, Kincaid & Krogh (1979)), for matrix-vector operations such as the product of a vector by a matrix

(Level 2 BLAS, Dongarra, Du Croz, Hammarling & Hanson (1988)) and for matrix-matrix operations such as the multiplication of full matrices (Level 3 BLAS, Dongarra, Du Croz, Duff & Hammarling (1990)). All these BLAS are now widely used in the development of software for linear algebra for dense problems. There are only a very limited number of different forms, each of which has a natural storage scheme: the matrices may be general, symmetric, triangular, or banded. It has been natural to develop different versions of an algorithm for the various forms. In the case of sparse matrices, this is not practical because of the variety of data structures. There are, however, instances when the full BLAS can be used on subproblems within an implementation of a code for sparse matrices (see for example Duff (1981), Amestoy, Daydé & Duff (1989), and Duff & Reid (1995)). Indeed, much of the power of frontal and multifrontal techniques for the solution of sparse equations comes from these kernels.

Dodson, Grimes & Lewis (1991) have proposed a standard for extending the Level 1 BLAS to the sparse case. Routines are included for gather and scatter, saxpy and sdot, and application of a Givens rotation. While they are a useful extension, they suffer from the same problem as the Level 1 BLAS in the full case, namely that the data-access requirements are of the same order as the arithmetic and so high efficiency is not obtained on most high performance computers.

Current sparse software packages do not offer enough flexibility. In NSPCG (Oppe, Joubert & Kincaid 1988), for example, the user is offered a choice of 5 basic storage formats and will choose the relevant subroutines to perform the basic linear algebra operations. The user is forced to adhere to this choice throughout the code. The user also has the ability to use other formats, but will then have to provide the code, following a given matrix-vector routine template. In our experience, the interface is not flexible enough to accommodate sufficient choices. Furthermore, if a format different from that provided by the library is used, there is less functionality (for example, red/black preconditioning cannot be applied). The issue of sparse BLAS for use with iterative methods is discussed at some length in Oppe & Kincaid (1990) but they consider only Level 1 and Level 2 BLAS and again use only very few prescribed storage modes for the sparse matrices. There are conversion routines in SPARSKIT (Saad 1994) but they are more at the level of the proposal by Carney, Heroux & Li (1993) and are again not designed to cater for any data structure.

Recently, there has been some concern over standardizing interfaces to iterative solvers for sparse systems (Ashby & Seager 1990) and it is primarily in this context that our current proposal is oriented. We feel that, particularly with the recent rapid growth in the use of high-performance computers, it is most timely to establish standards to achieve high performance without sacrifice of portability. Note that our proposal will fit equally well whether the iterative software performs a call to a matrix-vector multiply routine or whether reverse communication is used since in either case a call can be made to a given sparse matrix-full matrix multiplication routine; the call is made by the routine in the former case and by the user in the latter.

A preliminary draft of this proposal, entitled "User Level Sparse BLAS", was discussed in a workshop at the Copper Mountain Conference on Iterative methods in April 1992 in conjunction with a paper "A proposal for a sparse BLAS toolkit" by Michael Heroux. The Toolkit paper provides implementation level data dependent routines for Level 2 and Level 3 BLAS compatible with our proposal. Since 1992, our proposal has been discussed at several conferences and workshops so we now believe that the current proposal represents a fair reflection of the needs of the user community.

In this document, we have quite deliberately used Fortran 77 to describe our routines. This is partly for consistency with earlier BLAS proposals, but we also feel that it gives more information on the organization of the routines. Additionally, the numerical community, particularly in the United States, is still strongly wedded to Fortran 77. We recognize, however, that Fortran 90 will give a cleaner interface, and we illustrate this in Section 10. A sample code in Fortran 77 is available through anonymous ftp (to seamus.cc.rl.ac.uk in directory sparseblas) and shows an implementation of some of the paths.

2 General Overview

It is proposed to define standard interfaces for the following functions:

(1) a routine for performing the product of a sparse and a dense matrix,

(2) a routine for solving a sparse upper or lower triangular system of linear equations for a matrix of right-hand sides,

(3) a routine to check the input data, to transform from one sparse format to another, and to scale a sparse matrix, and

(4) a routine to permute the columns of a sparse matrix and a routine to permute the rows of a full matrix.

Note that our definition in (1) and (2) includes operations on vectors as a trivial subset. These may be coded separately at the machine dependent level.

The data preprocessing routine (3) is essential to this proposal. This routine is designed to be called before the body of the computation. The interface is designed to accept many different data formats and produce many others. In particular, it can interrogate the machine it is running on and transform the data into a format that is particularly suited for that machine. Readers interested in writing code for transforming from their own particular structures should consult the sparse toolkit proposal of Carney et al. (1993), which is primarily concerned with implementation issues.

Many algorithms require the permutation of matrices. Additionally, some efficient implementations of sparse matrix-vector products, and of the solution of sparse triangular systems on vector or parallel processors, require the vectors to be reordered. If high efficiency is required, it is necessary to avoid explicit vector permutations in the inner loops and, to enable this, routines have been added (4) to permute sparse matrices and full matrices appropriately. The permutation routines can also be called outside the body of the computation in order to increase efficiency by avoiding permutations within the main loop of the algorithm. This facility is discussed more in Section 6.

3 Scope of the Level 3 Sparse BLAS

In this section we present the mathematical definition of the scope of our Level 3 sparse BLAS.

If

- A and H are sparse matrices
- T is a triangular sparse matrix
- B and C are dense matrices
- D is a diagonal matrix
- P, P_R, and P_C are permutation matrices
- α and β are scalars,

then the operations proposed have the following forms:

- Matrix-matrix products
  - $C \leftarrow \alpha P_R A P_C B + \beta C$
  - $C \leftarrow \alpha P_R A^T P_C B + \beta C$
- Solving triangular systems of equations with multiple right-hand sides
  - $C \leftarrow \alpha D P_R T^{-1} P_C B + \beta C$
  - $C \leftarrow \alpha D P_R T^{-T} P_C B + \beta C$
  - $C \leftarrow \alpha P_R T^{-1} P_C D B + \beta C$
  - $C \leftarrow \alpha P_R T^{-T} P_C D B + \beta C$
- Data preprocessing including change of data structure
  - Checks on input data (optional)
  - $(H, P_R, P_C) \leftarrow DA$
- Permuting the columns of a sparse matrix
  - $A \leftarrow AP$
  - $A \leftarrow AP^T$
- Permuting the rows of a dense matrix
  - $C \leftarrow PC$
  - $C \leftarrow P^T C$

4 Naming conventions

The name of a Level 3 sparse BLAS routine follows the conventions of the Level 3 BLAS for dense matrices. The first character in the name denotes the Fortran data type of the matrix as follows:

- S REAL
- D DOUBLE PRECISION
- C COMPLEX
- Z DOUBLE COMPLEX

For the routines (1) to (3) in Section 2, characters two and three are 'CS' and denote that the input matrix has a *C*ompressed *S*tructure. For the routine in (4) for permuting dense systems, the characters GE (*GE*neral) are used for compatibility with earlier BLAS standards.

The fourth and fifth characters denote the operation as follows:

- MM Matrix-matrix product
- SM Solve a system of linear equations for a matrix of right-hand sides
- DP Data preprocessing routine
- CP Column Permutation
- RP Row Permutation

In the text of this paper we use precision independent names obtained by relacing the first character by an underscore. Thus, for example, _CSMM covers SCSMM, DCSMM, CCSMM, and ZCSMM.

5 Representation of sparse matrix

A sparse matrix A is represented by a character string and five arrays: a character array, a real array, and three integer arrays. All these arrays are included within the derived data type for sparse matrices when using Fortran 90, as we indicate in Section 10.

- FIDA: character*5
- DESCRA: character*1 array
- A: real array
- IA1 and IA2: integer arrays
- INFOA: integer array

FIDA is a character*5 variable that defines the format of the sparse matrix. DESCRA is a character*1 array of size 10 that describes the characteristics of the matrix. We give some examples of possible formats in the following but stress that this list is neither exhaustive nor necessarily supported in any particular implementation.

```
FIDA - the storage technique that is used
       CSC or Compressed Sparse Column
       CSR or Compressed Sparse Row
       COO or Coordinate format
       DIA or Diagonal format
       ELL or Ellpack_Itpack format
       JAD or Jagged_Diagonals
       BDI or Block Diagonal format
       BSC or Block Sparse Column format
       BSR or Block Sparse Row format
       SKY or Skyline format

DESCRA(1:1) - Matrix structure
              G or General
              S or Symmetric
              H or Hermitian
              T or Triangular
              A or Anti-symmetric (Skew_Symmetric/Hermitian)
              D or Diagonal

DESCRA(2:2) - Upper/Lower Indicator
              U or Upper
              L or Lower

DESCRA(3:3) - Diagonal
              U or Unit (diagonals not stored)
              N or Non_Unit

DESCRA(4:10) - Special information which may be used for a particular
storage representation
```

Sometimes it is necessary to store supplementary integer information on the sparse data format. One possibility is to use IA1 or IA2. However, this is often not acceptable because it would destroy compatibility with using them for some of the common formats. We therefore include a further integer array INFOA that can hold this information, examples being the number of nonzeros when using a coordinate scheme or the block size for block data formats. The INFOA array can also be used to convey further information to the preprocessing routine, for example an indication of the likely number of right-hand sides or the number of times the transformed data structure will be used so that the preprocessing routine may choose a better and more efficient data structure. INFOA can also be used to indicate that both the matrix and its transpose will be required. The format for the data structure output from the transformation routine is, of course, controlled by the implementor. Although we allow two integer arrays for output, in some implementations only one of these arrays might be used.

There is a potential problem with character string variables when calling Fortran subroutines from C. We feel strongly that the use of a character string is the best way of communicating data on the matrix format but, since we wish to avoid posing a difficult problem to the C programmer, we suggest the easier alternative of forbidding the use of the last character in argument FIDA so that the maximum string permitted is of length 4.

For example, if we were using the Harwell-Boeing format (Duff, Grimes & Lewis 1989), FIDA would be set to CSC, DESCRA(1) to G, A to the matrix values by columns, IA1 the corresponding row indices, and IA2 pointers to the position of the first entry of each column in the arrays A and IA1. INFOA could be used to convey information for the machine-specific transformation if it is required.

In the routine _CSDP, the user can request that the format of the sparse matrix is automatically transformed to one that is best on the target machine. This is done by specifying the characters `???` in FIDA for the output format. We expect the manufacturers to provide appropriate code in their implementations. A nice aspect of this is that the vendor could change the output format to best suit the hardware on which the code is actually being run. In this way, the user is isolated from changes in the hardware. Because of this principal feature of _CSDP, auxiliary data might need to be passed to _CSDP using the INFOA array. This environment has much in common with the currently favoured object-oriented (o-o) approach. For example, the best data structure for _CSMM or _CSSM may be very dependent on the number of right-hand sides or the number of columns in the full matrix (`K`, say). The entries of the sparse matrix are used `K` times, therefore as `K` increases the cost of indirectly addressing them reduces. When `K` is very large, a dense vector model of computation gives high performance (Agarwal, Gustavson & Zubair 1992). Similarly, it can be helpful to provide _CSDP with information on the number of subsequent calls that will be made using the output structure. If the computation is to be repeated a large number of times, it is more worthwhile to spend extra time in the data preprocessing phase in order to obtain a particularly efficient structure for subsequent computation. INFOA can also be used to inform the implementor that operations with both the matrix and its transpose will be later required, thus enabling further efficiency in data preprocessing by allowing both structures to be computed at the same time. A vendor chosen code would be output by _CSDP into the descriptor associated with the output matrix to describe the data structure used. This code would be used in calls to subsequent subroutines.

It is important to stress that the above formats are provided by way of guidance only. We do not claim that they would all be fully supported and, in addition, users may wish to add their own structures. For example, see results of Erhel (1990), or the Stripped Jagged Diagonals scheme (Paolini & Radicati di Brozolo 1989) that has proven efficient on an IBM 3090/VF computer. A good example of the influence of data structures on machine performance is given by Agarwal et al. (1992).

We feel that the six proposed parameters are sufficient to accommodate all of the more widely used sparse matrix storage techniques and we have allowed for future expansion by declaring DESCRA as a character*1 array of length 10 and INFOA as an INTEGER array of length 10. Although in much of our software development for this paper, we have primarily considered matrices represented in the CSR format, we have worked on implementations for several other storage techniques.

We have given no details of the storage for each of the structures mentioned above. The reader can find these in the paper by Carney et al. (1993).

6 Permutations

In order to avoid permutations on each vector algebra operation, we allow permutations on the data structures outside the loop of the iterative algorithm. We believe that this can be accomplished using only a column permutation of a sparse matrix and a row permutation of a dense matrix and we thus introduce subroutines to permute the columns of a sparse matrix or the rows of a full matrix explicitly. These subroutines are discussed in Section 8.4. Here we indicate how they may be used when efficient implementation is required.

Let us assume that all that is required in the inner loop of the iterative method is:

do i = 1, ...	
$y \leftarrow Ax$	Multiply by a sparse matrix
$x \leftarrow x + ay$	
end do	

and that the data conversion routine had converted the sparse matrix A to a matrix H with the same column ordering but a different row ordering. The code would then become:

$(P_C, H) \leftarrow A$	Change data structure and generate permutations
do i = 1, ...	
$y \leftarrow Hx$	Multiply by a sparse matrix
$y \leftarrow P_C y$	Multiply (left) by a permutation matrix
$x \leftarrow x + ay$	
end do	

and the additional permutation $P_C y$ (necessary to return y to the original ordering) in each iteration could significantly affect efficiency. By introducing explicit permutation subroutines, we can avoid this as follows:

$(P_C, H) \leftarrow A$	Change data structure
$x \leftarrow P_C^T x$	Multiply (left) by a permutation
$H \leftarrow H P_C$	Multiply sparse matrix by a permutation
do i = 1, ...	
$y \leftarrow Hx$	Multiply by a sparse matrix
$x \leftarrow x + ay$	
end do	
$x \leftarrow P_C x$	Multiply (left) by a permutation to restore to the original ordering

7 Argument conventions

We follow a convention for the argument lists similar to that for the Level 3 BLAS for dense matrices with

- Arguments specifying options
- Arguments defining the sizes of the matrices
- Input scalar
- Description of input matrices
- Input scalar (associated with input-output matrix)
- Description of the input-output matrix
- Work array
- Error flag

7.1 Arguments specifying options

TRANS is a character*1 argument and is used by the routines as shown in the following table.

Value	Meaning of TRANS
'N'	Operate with or generate the matrix
'T'	Operate with or generate the transpose of the matrix
'C'	Operate with or generate the conjugate of the matrix
'H'	Operate with or generate the conjugate transpose of the matrix

CHECK is a character*1 argument specifying an option in the data preprocessing routine _CSDP. Values for CHECK and their meanings are given in the following table.

Value	Meaning of CHECK
'C'	Perform checks on data and exit
'Y'	Perform checks on data and transform format
'N'	Do not perform data checks but transform format

In any call, if an actual argument has a character value other than those in the tables, an immediate error return is made and the data is unchanged.

7.2 Arguments defining the sizes of the matrices

As in the full BLAS, the number of rows and columns in the output array are given by M and N respectively. For _CSMM, the number of columns of the input sparse matrix and rows in the input full matrix are given by K. For _CSSM, the input sparse matrix is triangular of order M and there are N right-hand sides. If TRANS is set to 'T', 'C', or 'H' in the _CSMM routine, M and K are the number of rows and columns of the transpose of the sparse input matrix. It is permissible to call the routines with M or $N <= 0$, in which case the routines exit immediately without referencing their matrix arguments. If M and $N > 0$, but $K <= 0$, the operation performed by _CSMM reduces to $C \leftarrow \beta C$.

7.3 Input scalars

The scalars always have the dummy argument names ALPHA and BETA. ALPHA for the scalar α associated with the input matrix, and BETA for the scalar β associated with the input-output matrix.

7.4 Description of input matrices

We discussed the arguments for representing sparse matrices in Section 5. The sparse matrix for the matrix-matrix multiplication and the column permutation routines is held in arrays A, IA1, IA2, FIDA, DESCRA, INFOA. as indicated in Section 5. The triangular sparse matrix (for _CSSM) is held in arrays T, IT1, IT2, FIDT, DESCRT, and INFOT, while the output sparse matrix from _CSDP is held in H, IH1, IH2, FIDH, DESCRH, and INFOH. For the output sparse matrix only, lengths for H, IH1, and IH2 are held in LH, LIH1, and LIH2 respectively (see Sections 8.3 and 9.1). We also have a data format for a diagonal matrix. We hold this matrix in two arguments, a real array D holding the values of the entries on the diagonal, and a character*1 argument (UNITD) which indicates whether the diagonal matrix is unit (in which case D is not accessed) or whether the matrix is used for row scaling, column scaling, or both.

Value	Meaning of UNITD
'U'	Unit matrix
'L'	Row scaling
'R'	Column scaling
'B'	Row and column scaling with $D^{\frac{1}{2}}$

Permutation matrices are used both as input and output to the routines. Permutations for input matrices are held in PR and PC, and those for the output sparse matrix in P1 and P2.

A permutation matrix P is represented by an integer array stored as a vector whose entry i is equal to the position of the only nonzero entry in row i; in the following example the permutation matrix:

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

is represented by:

$$\begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix}.$$

If no permutation is needed (that is, the permutation is the identity), the value of the first entry of the permutation array can be set to 0.

The description of the input dense matrix consists of a permutation array and the array name B followed by LDB, the leading dimension of the array as declared in the calling (sub)program.

7.5 Description of the input-output matrix

The description of the input-output dense matrix consists of the array name C followed by the leading dimension LDC.

7.6 Work array

In many instances, efficiency is promoted if extra work space is available. We accommodate this in each case by a real array WORK. The length of array WORK is stored on entry in the variable LWORK. The minimum length required by the subroutine is returned in WORK(1).

7.7 Error flag

In contrast to the full Level 3 BLAS, we provide an error flag, IERROR, in every routine. Error handling is outlined in Section 9. The data checking option of the data preprocessing routine _CSDP does fairly extensive checking but the other routines, since they will normally be called afterwards and in the main loops of the code, do only minimal checking. We feel it is important to introduce this parameter since we have a more complex (and hence more error prone) situation than the full case and there has also been some criticism of the full case for not including such an argument.

8 Specifications of the Level 3 sparse BLAS

Type and dimension for variables occurring in the subroutine specifications are as follows

```
INTEGER       IERROR, LDB, LDC, LWORK, LIH1, LIH2, LH, M, N, INFOA(10),
              INFOH(10)
```

```
INTEGER        IA1(*), IA2(*), IT1(*), IT2(*), IH1(LIH1), IH2(LIH2),
               PR(M), PC(*), P1(M), P2(K)
CHARACTER*1    CHECK, TRANS, UNITD
CHARACTER*5    FIDA, FIDT, FIDH
CHARACTER*1    DESCRA(10), DESCRT(10), DESCRH(10)
```

For routines whose first letter is an S:

```
REAL           ALPHA, BETA
REAL           A(*), B(LDB,N), C(LDC,N), D(M), H(LH), T(*), WORK(LWORK)
```

For routines whose first letter is a D:

```
DOUBLE PRECISION  ALPHA, BETA
DOUBLE PRECISION  A(*), B(LDB,N), C(LDC,N), D(M), H(LH), T(*),
                  WORK(LWORK)
```

For routines whose first letter is a C:

```
COMPLEX        ALPHA, BETA
COMPLEX        A(*), B(LDB,N), C(LDC,N), D(M), H(LH), T(*), WORK(LWORK)
```

For routines whose first letter is a Z:

```
DOUBLE PRECISION COMPLEX  ALPHA, BETA
DOUBLE PRECISION COMPLEX  A(*), B(LDB,N), C(LDC,N), D(M), H(LH),
                          T(*), WORK(LWORK)
```

8.1 Sparse Matrix times Dense matrix

```
_CSMM (TRANS, M, N, K, ALPHA, PR, FIDA, DESCRA, A, IA1, IA2, INFOA, PC,
       B, LDB, BETA, C, LDC, WORK, LWORK, IERROR)
```

Operation (C is always mxn)

TRANS = 'N'	TRANS = 'T'
$C \leftarrow \alpha P_R\ A\ P_C\ B + \beta C$	$C \leftarrow \alpha P_R\ A^T\ P_C\ B + \beta C$

When the matrix is complex, the TRANS parameter can be 'N', 'T' for transpose, 'H' for conjugate transpose, or 'C' for conjugate.

8.2 Solution of triangular systems of equations

```
_CSSM(TRANS, M, N, ALPHA, UNITD, D, PR, FIDT, DESCRT, T, IT1, IT2, INFOT,
      PC, B, LDB, BETA, C, LDC, WORK, LWORK, IERROR)
```

	TRANS = 'N'	TRANS = 'T'
UNITD = 'U'	$C \leftarrow \alpha\ P_R\ T^{-1}\ P_C\ \ B + \beta C$	$C \leftarrow \alpha\ P_R\ T^{-T}\ P_C\ \ B + \beta C$
UNITD = 'L'	$C \leftarrow \alpha P_R\ D\ T^{-1}\ P_C\ \ B + \beta C$	$C \leftarrow \alpha P_R\ D\ T^{-T}\ P_C\ \ B + \beta C$
UNITD = 'R'	$C \leftarrow \alpha P_R\ T^{-1}\ \ D\ P_C\ \ B + \beta C$	$C \leftarrow \alpha P_R\ T^{-T}\ \ D\ P_C\ \ B + \beta C$
UNITD = 'B'	$C \leftarrow \alpha P_R D^{\frac{1}{2}} T^{-1} D^{\frac{1}{2}} P_C B + \beta C$	$C \leftarrow \alpha P_R D^{\frac{1}{2}} T^{-T} D^{\frac{1}{2}} P_C B + \beta C$

As we see in the second and third columns, when TRANS has the value 'T', the only change to the assignments in column 2 is that T^{-T} is used in place of T^{-1}. When the matrix is complex, the TRANS parameter may also have the value 'H' for conjugate transpose, T^{-H} for T^{-1}, or 'C' for conjugate, $\overline{T}^{-1}$.

8.3 Specification of data preprocessing routine

The purpose of the data preprocessing routine is to convert the user's data structures for holding the sparse matrix into structures better suited for the subsequent operations on the target architecture. The matrix can also be scaled. There is also a capability to check the input data for inconsistencies or errors.

The matrix being input is held in arrays FIDA, DESCRA, A, IA1, IA2, and INFOA. Information for scaling is held in UNITD and D and the output matrix is held in FIDH, DESCRH, H, IH1, IH2, and INFOH.

The data preprocessing routine invoked by the following call:

```
_CSDP(CHECK, TRANS, M, N, UNITD, D, FIDA,DESCRA,A,IA1,IA2, INFOA, P1,
      FIDH, DESCRH, H, IH1, IH2, INFOH, P2, LH, LIH1, LIH2,
      WORK, LWORK, IERROR)
```

If CHECK is set to 'Y' or 'C', the above routine will check the input data to see if indices are within range, and, in the case of triangular matrices, to see if they are indeed triangular and have a nonzero diagonal. An error return is also invoked if the data structures requested are not in the implementation. If CHECK is equal to 'C' then the routine immediately returns after checking the input data. If CHECK is equal to 'Y' or 'N', the subroutine transforms the sparse matrix from the data structure FIDA, DESCRA, A, IA1, IA2, and INFOA to the data structure FIDH, DESCRH, H, IH1, IH2, and INFOH, optionally (depending on value of UNITD) scaling the matrix by the diagonal matrix D. The input-output relation is $DA = P_1 H P_2$; row and column permutations used in the conversion are provided as output in P1 and P2, respectively. When a call to _CSDP is followed by a call to _CSMM, P1 and P2 can be used as PR and PC respectively in the second call. When a call to _CSDP is followed by a call to _CSSM, P1 and P2 can be used as PC and PR, due to the inversion of matrix T in the _CSSM definition.

Because the storage for the transformed matrix may differ from the original, we input the dimensions of H, IH1, and IH2 in LH, LIH1, and LIH2, respectively. If these are insufficient for the data format requested, an error return is invoked. The number of locations required in H, IH1, and IH2 is returned in H(1), IH1(1), and IH2(1) respectively, as described in Section 9.1.

TRANS might be used if the matrix A is stored but the user wishes a more efficient way for performing the matrix-matrix multiplication $A^T B$. Then TRANS would be set to 'T' in the _CSDP call but be set to 'N' in the call to _CSMM. Note that if _CSDP is called with TRANS 'N' followed by _CSMM with TRANS set to 'T', the same function will be performed but _CSMM would possibly be using an inappropriate data structure for multiplication by A^T. It may be that some vendors would wish to discourage this by not supporting calls to _CSMM or _CSSM with TRANS equal to 'T', returning with an error condition if such a call is attempted.

It is not expected that transformations will be provided between all possible data structures. Note that when _CSDP is used to generate a vendor chosen data structure, FIDH will be set to ??? on input and will be reset to a vendor chosen identifier on output. It is this reset value which should be used in subsequent calls.

8.4 Routines to permute matrices

As discussed in Section 6, our standard provides routines for permuting the columns of a sparse matrix and the rows of a dense matrix. These are provided to allow permutations on the data structures outside the loop of the iterative algorithm as illustrated in Section 6. Because the permutation of the dense matrix corresponds to a full Level 3 BLAS, we use the appropriate full Level 3 BLAS nomenclature.

We thus have the following calls, _CSCP performs the column permutation of a sparse matrix and _GERP the row permutation of a dense one.

```
_CSCP(TRANS, M, N, FIDA, DESCRA, IA1, IA2, INFOA, P, WORK, LWORK, IERROR)
```

and

```
_GERP(TRANS, M, N, P, B, LDB, WORK,LWORK, IERROR)
```

Calling _CSMM with the two permutation matrices P_R and P_C is the same as calling _CSMM with I (identity matrix) and P_C followed by a call to _GERP with P_R.

9 Error management

As we said in Section 7.7, we provide the argument IERROR to return information on errors found in the routines. There are two levels of error. Those with IERROR negative signify a terminal error so that the normal recourse is to halt the computation. A positive value of IERROR is used to warn the user about a potentially abnormal situation. This follows the system used in the Harwell Subroutine Library (Anon 1993), where negative flags are associated with fatal errors and positive flags with warnings.

Following the style of LAPACK (Anderson, Bai, Bischof, Demmel, Dongarra, DuCroz, Greenbaum, Hammarling, McKenney, Ostrouchov & Sorensen 1992), we use negative values of IERROR when an invalid argument is passed to a routine. If the i-th argument has an invalid value, IERROR is set to $-i$ and the error handler routine XERBLA is called. If, for instance, a user calls DCSMM with TRANS set to 'M', XERBLA writes to standard output the message:

```
 ** On entry to DCSMM  parameter number  1 had an illegal value
```

and stops the execution. The user, however, can remove the STOP in XERBLA to allow the routine to return to the calling BLAS which, in turn, will return immediately to the calling program. In this case, the user can check for an error by testing the value of IERROR.

An example where a warning might be returned is a call to _CSMM with M set to 0. As we pointed out in Section 7.2, in this case the routine exits immediately. We do not consider this to be an error, because this feature might be useful for particular applications, but a warning should be issued anyway, since this condition might be caused by a programming error.

9.1 Memory errors

When a routine that needs a certain amount of memory to work is called with an insufficient value in arguments LWORK, LH, LIH1, or LIH2, the routine sets IERROR to the proper (negative) value, calls XERBLA and returns. In some cases, it is desirable to know the minimum amount of memory required for the correct execution, *before* the actual computation is done. This can be done by calling the routine with arguments LWORK, LH, LIH1, LIH2 set to 1. If the minimum values of these arguments can be supplied, they will be returned in the first location of the corresponding arrays, else the standard error handling takes place. Note that, if the call to _CSDP is successful, the minimum value of LWORK is returned in WORK(1), but this cannot be done for LH, LIH1, LIH2, since H, IH1 and IH2 contain the converted output matrix. So a call with LH, LIH1, LIH2 set to 1 is the only guaranteed way for _CSDP to return the minimum values for these arguments.

10 Fortran 90 interface

The use of Fortran 77 as a description language is ideal for illustrating the closeness of our proposal to previous BLAS standards, and gives a tight definition for our proposal which will, we believe, be of assistance to anybody implementing it. However, it is clear that a cleaner interface can be provided by Fortran 90, particularly in the description of the sparse matrix data structures.

We show below the interface block for a Fortran 90 module for implementing the sparse

BLAS. At present, although this block is available on anonymous ftp, the whole suite of Fortran 90 subprograms is not yet ready. We have, however, tested the Fortran 90 version of _CSMM.

```
MODULE TYPESP
   TYPE SPMAT
      INTEGER M,K
      CHARACTER*5 FIDA
      CHARACTER*1 DESCRA(10)
      INTEGER     INFOA(10)
      DOUBLE PRECISION,POINTER :: ASPK(:)
      INTEGER,POINTER :: IA1(:),IA2(:),PL(:),PR(:)
   END TYPE SPMAT
END MODULE TYPESP
MODULE SPARSE_BLAS3
   INTERFACE PREPARE
      FUNCTION PREPARE(A,TRANS,CHECK,UNITD,D,ERROR)
         USE TYPESP
         TYPE (SPMAT) PREPARE, A
         CHARACTER TRANS, CHECK, UNITD
         DOUBLE PRECISION D(A%M)
         INTEGER ERROR
      END FUNCTION PREPARE
   END INTERFACE
   INTERFACE MULTIPLY
      SUBROUTINE MULTIPLY(ALPHA,A,TRANS,B,BETA,C,ERROR)
         USE TYPESP
         TYPE (SPMAT) A
         DOUBLE PRECISION C(:,:)
         DOUBLE PRECISION B(A%K,SIZE(C,2)), ALPHA, BETA
         CHARACTER TRANS
         INTEGER ERROR
      END SUBROUTINE MULTIPLY
   END INTERFACE
   INTERFACE TRISOLVE
      SUBROUTINE TRISOLVE(ALPHA,A,TRANS,UNITD,D,B,BETA,C,ERROR)
         USE TYPESP
         TYPE (SPMAT) A
         DOUBLE PRECISION C(:,:)
         DOUBLE PRECISION D(A%M), B(A%K,SIZE(C,2)), ALPHA, BETA
         CHARACTER TRANS, UNITD
         INTEGER ERROR
      END SUBROUTINE TRISOLVE
   END INTERFACE
   INTERFACE SPARSEPERM
      SUBROUTINE SPARSEPERM(A,TRANS,PERM,ERROR)
         USE TYPESP
         TYPE (SPMAT) A
```

```
         CHARACTER TRANS
         INTEGER PERM(A%K), ERROR
      END SUBROUTINE SPARSEPERM
   END INTERFACE
   INTERFACE FULLPERM
      SUBROUTINE FULLPERM(B,TRANS,PERM,ERROR)
         DOUBLE PRECISION B(:,:)
         CHARACTER TRANS
         INTEGER PERM(SIZE(B,1)), ERROR
      END SUBROUTINE FULLPERM
   END INTERFACE
END MODULE SPARSE_BLAS3
```

11 Discussion of the design

As we said in the introduction, establishing a standard for Level 3 sparse BLAS involves many compromises. We have tried to be as frugal as practical in introducing new routines and, although the calling sequences in Fortran 77 are long, we have kept them as short as possible for the functionality and flexibility we think necessary. In this section, we describe some of the compromises explicitly. Also, we stress that the standard is designed to allow future expansion and so the fact that some feature is omitted in the current proposal does not preclude its existence in the future.

- Naming conventions

  Our naming convention follows that of the earlier BLAS. For the matrix-matrix and triangular solve, we have used the characters MM and SM respectively as in the case of full BLAS. Since the routine for creating a new data format, checking the data, and scaling the matrix would normally be called prior to the body of the main code, we have called this a "data preprocessing" routine and have used the characters DP.

- Representation of sparse format

  The format for the sparse matrix is held in the real array A, two integer arrays IA1 and IA2, a character string FIDA and a character array DESCRA. We have incorporated UPLO and DIAG, as used in the full BLAS, within the DESCRA array. Because it is sometimes necessary to hold auxiliary information, for example the number of entries for the coordinate scheme or the number of right-hand sides to help in the automatic selection of optimal data formats, we do this by including another integer array INFOA. We prefer this to two other solutions of including the integer information in IA1 or IA2 (or in a combined IA1/IA2 array), or providing an auxiliary subroutine PUTCHAR that stores an integer value in DESCRA. Although it would have perhaps been cleaner to have only one integer array instead of three (IA1, IA2, and INFOA), we have made a concession to some of the more commonly used formats (CSC, CSR and the coordinate scheme, COO, for example) by including the two arrays. We feel it is important and user-friendly to provide this backward compatibility, but it is certainly an area that caused much comment (both for and against) in our earlier draft. The format of the output arrays from _CSDP (H, IH1, IH2, INFOH) are of course at the discretion of the implementor so it is possible that all integer output is included in only one array. Although we gave some examples for possible data formats in Section 11, we must stress that they are not meant to be exhaustive, nor would we expect every implementation to support all those we mention. What is important, however, is that our interface is flexible enough so that any sensible input data format can be included within our framework.

- Transformation between data formats

  In most applications, the ??? output will be used. We feel, however, that it is worth allowing the greater flexibility of permitting the user to transform between two data structures of his or her own choosing, even if such a code is not implemented by the vendors.

- Use of character variables

  There is a problem with the use of character string variables when calling Fortran subroutines from C programs. Since we envisage the use of the kernels from C we need to take some action. We could place the burden on the C programmer or could avoid the use of character strings altogether. However, we do not like either of these options and have chosen to avoid storage of information in the last character of the character variable. In this case, we only allow a maximum of four characters to be stored in a character*5 variable.

- Finite-element matrices

  We have tried not to prejudice the way the sparse matrix is stored. Indeed we debated about including a routine to assemble a finite-element problem. The comments which we had on our earlier draft indicate that it would not be possible to offer full support for finite-element applications without significantly altering the proposal and making it much less suitable for its main purpose. There was some support for a routine for assembling a matrix, but that would be at quite a different level from our current set of routines and so we do not propose to include this. Such a routine has been included in Release 12 of HSL as subroutine MC37. We should point out that the Harwell-Boeing format (Duff et al. 1989) does allow for the storage of finite-element matrices and so they can be held in the format proposed in this paper.

- Scaling matrices

  We have not included a diagonal scaling matrix in the _CSMM routines but feel the appropriate place for such a scaling is at the preprocessing stage to avoid extra overhead in the inner loops. We have, however, allowed the user to use a diagonal scaling matrix in the case of triangular solves. This could, for example, permit the diagonal of the triangular matrix (or its inverse) to be stored separately for efficiency and flexibility.

- Use of TRANS

  In most cases, _CSMM and _CSSM will be run with TRANS equal to 'N', since the matrix will have been transposed already, if required, by _CSDP. Indeed, some vendors may disable a call to _CSMM or _CSSM with TRANS not equal to 'N'. At the moment, however, we have kept the parameter TRANS because, for example, the user may be developing code on a machine without vendor implementations and may find it easier to avoid explicit transposition in _CSDP. Note that, in the _CSDP routine, the implementor could choose to generate both structures for the matrix and its transpose. Information on whether both are required is passed in the INFOA array. We preferred this to allowing TRANS in _CSDP to have the value 'B' (for both) since we feel this is more an implementation issue akin to stipulating the number of subsequent right-hand sides.

- Triangular matrix-matrix multiplication

  There was some interest in a triangular matrix-matrix multiplication routine. We feel that this is more suitably included as a subcase of the _CSMM routine and so do not have this as a special case.

- Omission of TRANSB

  The full Level 3 BLAS allows transposition of the matrix B. However, we do not see the usefulness of this in the present context and do not include this parameter in our Level 3 sparse BLAS.

- Level 2 sparse BLAS

  We have intentionally not provided explicit Level 2 BLAS routines _CSMV and _CSSV since we feel their functionality is easily incorporated within _CSMM and _CSSM respectively. We do not believe efficiency need be compromised because special action could be taken by the vendor when the number of columns in `B` is equal to 1.

- Permutations of matrices

  We have restricted the explicit permutation calls to only two in the belief that it is unnecessary also to include row permutations of sparse matrices or column permutations of full ones. Although this is unæsthetically unsymmetric, we want to keep the demands on implementors to a minimum. In the permutation routines, we allow both operations by a permutation and its transpose although it is trivial to generate one from the other.

12 Acknowledgments

Many of our colleagues made very helpful comments on an earlier draft of this proposal. We would like to thank all those who participated in the various discussions at iterative conferences and individuals who have commented on the draft: Ramesh Agarwal, Richard Brankin, Michel Daydé, Fred Gustavson, Nick Gould, Michael Heroux, Gérard Meurant, Marco Perezzani, Alexander Peters, John Reid, Willi Schönauer, Ray Tuminaro, and Henk van der Vorst.

References

Agarwal, R. C., Gustavson, F. G. & Zubair, M. (1992), A high performance algorithm using pre-processing for the sparse matrix-vector multiplication, *in* ACM/IEEE, ed., 'Proceedings of Supercomputing '92, Minneapolis, MN. Nov 16-20, 1992.', IEEE Computer Society Press, Los Alamitos, California, pp. 32–41.

Amestoy, P. R., Daydé, M. & Duff, I. S. (1989), Use of level 3 BLAS in the solution of full and sparse linear equations, *in* J.-L. Delhaye & E. Gelenbe, eds, 'High Performance Computing: Proceedings of the International Symposium on High Performance Computing, Montpellier, France, 22–24 March, 1989', North Holland, Amsterdam, pp. 19–31.

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S. & Sorensen, D. (1992), *LAPACK Users' Guide.*, SIAM Press.

Anon (1993), *Harwell Subroutine Library. A Catalogue of Subroutines (Release 11)*, Theoretical Studies Department, AEA Industrial Technology.

Ashby, S. F. & Seager, M. K. (1990), A proposed standard for iterative solvers, Technical Report 102860, LLNL, Livermore, CA, USA.

Carney, S., Heroux, M. A. & Li, G. (1993), A proposal for a sparse BLAS toolkit, Technical Report TR/PA/92/90 (Revised), CERFACS, Toulouse, France.

CMSSL (1992), CMSSL for CM Fortran. version 3.0, Technical report, Thinking Machines Corporation.

Dodson, D. S., Grimes, R. G. & Lewis, J. G. (1991), 'Sparse extensions to the Fortran Basic Linear Algebra Subprograms', *ACM Transactions on Mathematical Software* **17**, 253–263.

Dongarra, J. J., Du Croz, J., Duff, I. S. & Hammarling, S. (1990), 'A set of Level 3 Basic Linear Algebra Subprograms.', *ACM Transactions on Mathematical Software* **16**, 1–17.

Dongarra, J. J., Du Croz, J. J., Hammarling, S. & Hanson, R. J. (1988), 'An extented set of Fortran Basic Linear Algebra Subprograms', *ACM Trans. Math. Softw.* **14**, 1–17.

Duff, I. S. (1981), Full matrix techniques in sparse Gaussian elimination, *in* G. Watson, ed., 'Numerical Analysis Proceedings, Dundee 1981', Lecture Notes in Mathematics 912, Springer-Verlag, Berlin, pp. 71–84.

Duff, I. S. & Reid, J. K. (1995), Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems, Technical Report RAL-TR-95-040, Rutherford Appleton Laboratory.

Duff, I. S., Grimes, R. G. & Lewis, J. G. (1989), 'Sparse matrix test problems', *ACM Transactions on Mathematical Software* **15**(1), 1–14.

Erhel, J. (1990), 'Sparse matrix multiplication on vector computers', *Int Journal of High Speed Computing* **2**, 101–116.

ESSL (1990), IBM Engineering and Scientific Subroutine Library. Guide and Reference, Technical report, IBM Corporation.

Lawson, C. L., Hanson, R. J., Kincaid, D. R. & Krogh, F. T. (1979), 'Basic linear algebra subprograms for Fortran usage', *ACM Trans. Math. Softw.* **5**, 308–323.

Oppe, T. C. & Kincaid, D. R. (1990), Are there iterative BLAS?, Technical Report CNA-240, Center for Numerical Analysis, The University of Texas at Austin.

Oppe, T. C., Joubert, W. & Kincaid, D. R. (1988), NSPCG User's guide. A package for solving large linear systems by various iterative methods, Technical Report CNA-216, Center for Numerical Analysis, The University of Texas at Austin.

Paolini, G. V. & Radicati di Brozolo, G. (1989), 'Data structures to vectorize CG algorithms for general sparsity patterns', *BIT* **29**, 703–718.

Saad, Y. (1994), SPARSKIT: a basic tool kit for sparse matrix computations. VERSION 2, Technical report, Computer Science Department, University of Minnesota.