ALGORITHM 656
An Extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs

JACK J. DONGARRA
Argonne National Laboratory
JEREMY DU CROZ and SVEN HAMMARLING
Numerical Algorithms Group, Ltd.
and
RICHARD J. HANSON
Sandia National Laboratory

This paper describes a model implementation and test software for the Level 2 Basic Linear Algebra Subprograms (Level 2 BLAS). Level 2 BLAS are targeted at matrix-vector operations with the aim of providing more efficient, but portable, implementations of algorithms on high-performance computers. The model implementation provides a portable set of FORTRAN 77 Level 2 BLAS for machines where specialized implementations do not exist or are not required. The test software aims to verify that specialized implementations meet the specification of Level 2 BLAS and that implementations are correctly installed.

Categories and Subject Descriptors: F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*computations on matrices*; G.1.0 [**Numerical Analysis**]: General—*numerical algorithms*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*linear systems (direct and iterative methods)*; G.4 [**Mathematics of Computing**]: Mathematical Software—*certification and testing; efficiency; portability; reliability and robustness; verification*

General Terms: Algorithms, Measurement, Performance, Reliability, Verification

Additional Key Words and Phrases: Extended BLAS, utilities

1. SCOPE OF THE ALGORITHM

In [7] we have defined the specification of a set of Basic Linear Subprograms for selected matrix-vector operations usually referred to as "Level 2 BLAS" or "Extended BLAS." They provide a standard framework to develop modular, portable, and efficient FORTRAN 77 code for many computational problems in

J. J. Dongarra's research was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38; R. J. Hanson's research was supported by the U.S. Department of Energy.

Authors' addresses: J. J. Dongarra, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439; J. Du Croz and S. Hammarling, Numerical Algorithms Group, Ltd., NAG Central Office, Mayfield House, 256 Banbury Road, Oxford OX2 7DE, England; R. J. Hanson, Applied Dynamics International Corp., 3800 Stone School Road, Ann Arbor, MI 48108.

linear algebra. Our hope is that specialized implementations of Level 2 BLAS will be developed for many machines, especially for vector processors and other high-performance computers. Thus, programs that call Level 2 BLAS can be efficient across a wide range of machines.

To support and encourage the use of Level 2 BLAS, this algorithm contains two components of software:

(1) A model implementation of the subprograms in FORTRAN 77. This enables Level 2 BLAS to be used on any machine, regardless of whether a specialized implementation exists. It is described in Section 2.

(2) Test programs, designed to ensure that implementations conform to the specification and have been correctly installed. The programs are described in Section 4.

Section 3 contains some advice on developing specialized implementations of the subprograms. Installation notes for the software are given in the Appendix.

2. THE MODEL IMPLEMENTATION

2.1 Programming Considerations

There are many mathematically equivalent ways to implement Level 2 BLAS, even in standard FORTRAN 77, as discussed in Section 3. The choice of method for the model implementation has been guided by the following considerations:

(1) The elements of the array A are accessed sequentially, column by column. On vector-processing machines, this allows the columns of the array to be recognized as contiguous vectors (by the FORTRAN compiler). On virtual-memory machines, it keeps page swaps to a minimum.

(2) Special code is included for the commonly occurring cases when the increment parameters (*INCX* or *INCY* or both), which are used in the inner loops, are equal to 1. This code can use a simpler, and hence more efficient, FORTRAN indexing scheme and also allows contiguous vectors to be recognized by the FORTRAN compiler; for example,

```
      DO 50, I = 1, M
         Y(I) = Y(I) + TEMP*A(I, J)
50    CONTINUE
```

instead of the following code:

```
      IY = KY
      DO 50, I = 1, M
         Y(IY) = Y(IY) + TEMP*A(I, J)
         IY = IY + INCY
50    CONTINUE
```

(3) Provision is made to skip the innermost loop if relevant elements of the vectors X or Y are zero. This can yield a considerable gain in efficiency if

the vectors are sparse:

```
         IF (X(JX).NE.ZERO) THEN
            TEMP = ALPHA*X(JX)
            DO 50, I = 1, M
               Y(I) = Y(I) + TEMP*A(I, J)
      50    CONTINUE
         END IF
```

2.2 Efficiency

The model implementation is likely to achieve considerable efficiency on scalar-processing machines with a good optimizing compiler, and even moderate efficiency on vector-processing machines with a good vectorizing compiler.

For illustration, Table I gives speeds obtained for some representative operations on a CRAY-1S, using automatic vectorization (no compiler directives to ignore data dependencies were needed). The speeds are given in megaflops and were measured with $m = n = 256$, $INCX = INCY = 1$, $UPLO$ = 'U', $DIAG$ = 'N', and no zero elements in the data. Speeds with $UPLO$ = 'L' were approximately the same as those with $UPLO$ = 'U', speeds of the SV routines were approximately the same as those of the corresponding MV routines, and speeds of the routines using packed storage were approximately the same as those of the corresponding routines using two-dimensional array storage. Without automatic vectorization (i.e., running in scalar mode), the speeds were between 5 and 7 Mflops for REAL data, and between 10 and 14 Mflops for COMPLEX data.

Table I does not include measurements on the routines for banded matrices. In the model implementation of these routines, the vector lengths are at most equal to the bandwidth, and hence, for narrow bandwidths the routines run at roughly scalar speeds. However, see Section 3.3 concerning an alternative implementation of some of these routines.

2.3 Language Standards

The model implementation of Level 2 BLAS is written entirely in portable standard FORTRAN 77 with two exceptions:

(1) For the routines that require a DOUBLE-PRECISION COMPLEX data type (names beginning with Z), we have used the following extensions to standard FORTRAN:

—COMPLEX*16 type specification statements;

—DCONJG and DCMPLX intrinsic functions whose argument and result are both of type COMPLEX*16;

—a DBLE intrinsic function with a COMPLEX*16 argument and DOUBLE PRECISION result, delivering the real part of the argument; and

—COMPLEX*16 constants formed from a pair of double-precision constants in parentheses.

(2) For the arguments of type CHARACTER that specify options, we wish to allow either upper- or lowercase characters to be supplied. Lowercase characters are not part of the standard FORTRAN character set, but their use is so widespread that it would be unfriendly not to allow them. This can be an

Table I. Speed of Level 2 BLAS on CRAY-1S in Mflops

Real			Complex		
Routine	*TRANS*	Speed	Routine	*TRANS*	Speed
SGEMV	'*N*'	39	CGEMV	'*N*'	63
	'*T*'	31		'*T*'	11
SSYMV		31	CHEMV		13
STRMV	'*N*'	33	CTRMV	'*N*'	56
	'*T*'	20		'*T*'	11
SGER		39	CGERU		63
SSYR		36	CHER		56
SSYR2		47	CHER2		46

obstacle to portability on some systems (as discussed in Section 7 of [7]), but we have avoided most of the problems by use of the auxiliary LOGICAL function LSAME described below.

2.4 Auxiliary Subprograms

Two auxiliary subprograms are called by Level 2 BLAS: an error-handling routine XERBLA, called by all routines, and the character-comparison routine LSAME, called by all except the _GER_ routines. Both these subprograms may be selectively modified by installers of the package as described in the Appendix. No changes need be made to the rest of the model code.

3. NOTES ON IMPLEMENTATION

Here we offer some advice to anyone planning to develop a specialized, machine-specific implementation of Level 2 BLAS. The following broad possibilities should be considered:

(1) rewriting the algorithms in FORTRAN so that the structure of the inner loops is better adapted to the architecture of the machine;
(2) using machine-specific extensions to FORTRAN, such as array syntax, compiler directives, or calls to library routines; and
(3) coding the routines in assembly language.

Approaches (2) and (3) should be considered as extensions of (1), not as alternatives. Implementors should not consider translating the model implementation into extended FORTRAN or assembly language without first considering whether the structure of the model implementation is well adapted to their machine.

Each matrix-vector operation performed by Level 2 BLAS involves doubly nested loops. By interchanging the inner and outer loop indexes, we obtain two variant ways of organizing the computation: In one variant the matrix is accessed by columns; in the other by rows. For the MV and SV routines, the inner loop is equivalent, in one variant, to an inner-product vector operation; and in the other variant, to a vector operation of the form $y \leftarrow \alpha x + y$, which we shall refer to as an AXPY. (For the rank-1 and rank-2 update routines, the inner loop is always equivalent to an AXPY operation.) The choice of method will be governed principally by the relative efficiency of performing inner products or AXPY operations, and by the cost of accessing a two-dimensional array by rows instead of by columns. We discuss the options in more detail for individual routines below. To be specific, we discuss the REAL SINGLE-PRECISION routines.

3.1 Routines Using Full Matrix Storage

We first consider those routines that require the matrix to be stored conventionally in a two-dimensional array: Columns of the matrix are stored in columns of the array and constitute contiguous vectors; and rows of the matrix are stored in rows of the array and constitute vectors with constant stride whose elements are not contiguous. Accessing vectors with noncontiguous elements may require expensive gather or scatter operations (e.g., as on the CDC Cyber 205), or may cost no more than accessing contiguous vectors except when memory-bank conflicts occur (as on the CRAY-1S).

3.1.1 *SGEMV.* The operations performed by this routine have been discussed by Dongarra, Gustavson, and Karp [5], and Daly and Du Croz [2]. The operation $y \leftarrow \alpha Ax + \beta y$ (*TRANS* = '*N*') can be performed either by AXPY operations with vector length m, accessing A by columns, or by inner products with vector length n, accessing A by rows. The operation $y \leftarrow \alpha A^T x + \beta y$ (*TRANS* = '*T*' or '*C*') can be performed either by inner products with vector length m, accessing A by columns, or by AXPY operations with vector length n, accessing A by rows. Here m and n are the numbers of rows and columns of A, respectively.

In both cases the AXPY form has the property that a sequence of AXPY operations are used to update a single left-hand-side vector. On a machine with vector registers, this left-hand-side vector (or segments of it) should ideally be held in a vector register throughout the iterations of the outer loop in order to reduce the number of memory references (see [5]). If the routines are being implemented in FORTRAN and the compiler cannot recognize and take advantage of this property, then the technique to unrolling the outer loops may be applied [1, 3].

Note that the relative advantage of the AXPY or inner-product forms depends on the values of m and n, and an optimal implementation may need to switch between the two forms accordingly.

On parallel machines the cleanest way to achieve concurrency is to compute segments of the result vector in separate processors; this is discussed further by Dongarra and Sorensen [4].

3.1.2 *SSYMV.* As in SGEMV each operation can be performed either by AXPY operations or by inner products. However, in each case the matrix must be accessed partly by columns and partly by rows (because only half the matrix is stored). The model implementation uses a mixed form in which each iteration of the outer loop contains one AXPY operation and one inner-product operation, both involving the same column of the matrix. On many machines this mixed form can halve the number of memory references to elements of A; however, if, say, inner products are markedly slower than AXPY operations, then they will govern the speed of the mixed form, and a pure AXPY form may be preferable. The remarks made about the AXPY forms of SGEMV on vector-register machines apply here also, although the details are more complicated because the lengths of the left-hand-side vectors are not constant, but increase or decrease by 1 on each iteration of the outer loop. Unrolling the outer loops to a depth of 2 can be handled neatly as described by Dongarra, Kaufman, and Hammarling [6]. In a pure AXPY form or pure inner-product form, it may be preferable to make two separate passes through the outer loop, one in which the matrix is accessed by rows and one in which it is accessed by columns.

3.1.3 *STRMV and STRSV.* Again, as in SGEMV, each operation can be performed either by AXPY operations or by inner products, with the matrix being accessed by rows or columns depending on the value of *TRANS*. Those forms of the code that are not used in the model implementation can easily be derived from those that are. For example, to derive an AXPY form of the code for $x \leftarrow L^T x$, simply take the code for $x \leftarrow Ux$, and replace $A(I, J)$ by $A(J, I)$. The remarks in Section 3.1.1 about implementing the AXPY forms on a machine with vector registers apply here also, although as in SSYMV the vector lengths are not constant throughout the outer loop.

The iterations of the outer loop must be performed in a particular order: forward from 1 to n for the operations $x \leftarrow Ux$, $x \leftarrow L^T x$, $x \leftarrow L^{-1}x$, and $x \leftarrow (U^T)^{-1}x$; and backward from n to 1 for the others. In STRMV this constraint is needed simply to allow the result vector to overwrite the input vector. In STRSV the recursive nature of the computation is more fundamental: Each element of the result vector depends on previously computed elements.

3.1.4 *Rank-1 and Rank-2 Update Routines.* Each column of the matrix can be updated by an AXPY operation or in the case of the R2 routines by a double AXPY operation. Moreover, on a parallel machine each column of the matrix can be updated concurrently. Interchanging the loop indexes merely results in AXPY operations on rows of the matrix.

3.2 Routines Using Packed Storage

With the specified storage scheme for packed matrices, columns of the matrix are stored as contiguous vectors within the packed array. Rows of the matrix do not constitute vectors with constant stride. Hence, those forms in which the matrix is accessed by columns are likely to be the only forms worth considering.

3.3 Routines for Banded Matrices

The same choice between inner-product and AXPY forms is available as for operations on full matrices. With the specified storage scheme for banded matrices, columns of a matrix are stored in columns of the array and constitute contiguous vectors; and rows of the matrix are stored in reverse diagonals of the array and constitute vectors with constant stride. Whether the matrix is accessed by rows or by columns, the vector length is at most $kl + ku + 1$ for SGBMV, and at most $k + 1$ for the other banded routines; hence, with typical bandwidths, speeds on vector processors may be slow.

For SGBMV and SSBMV, however, a third alternative can be used in which the matrix is accessed by diagonals, and hence, the array is accessed by rows. For SGBMV the essential features of the code (when $INCX = INCY = 1$) are shown in Figure 1. In this form the inner loop is equivalent to a vector operation of the form $V \leftarrow V + \alpha^* V^* V$ (V a vector, α a scalar). The vector lengths are close to n, and for large n good speeds can be obtained that are more or less independent of

```
      IF( LSAME( TRANS, 'N' ) )THEN
C
C        Form  y := alpha*A*x + y.
C
         DO 60, I = 1, KL + KU + 1
            L = KU + 1 - I
            DO 50, J = MAX( 1, 1 + L ), MIN( N, M + L )
               Y( J - L ) = Y( J - L ) + ALPHA*X( J )*A( I, J )
   50       CONTINUE
   60    CONTINUE
      ELSE
C
C        Form  y := alpha*A'*x + y.
C
         DO 100, I = 1, KL + KU + 1
            L = KU + 1 - I
            DO 90, J = MAX( 1, 1 + L ), MIN( N, M + L )
               Y( J ) = Y( J ) + ALPHA*X( J - L )*A( I, J )
   90       CONTINUE
  100    CONTINUE
      END IF
```

Fig. 1. Essential features of the code (*INCX* = *INCY* = 1) for SGBMV.

the bandwidth, for example, 30 Mflops for REAL data and 40 Mflops for COMPLEX data on a CRAY-1S. The same organization can be used for STBMV provided that a temporary work vector can be created to hold the result, but cannot be used for STBSV because of the recursive nature of the computations.

3.4 Other Remarks

The model implementation includes separate segments of code for cases when *INCX* and/or *INCY* = 1: On many machines this is unnecessary.

Specialized implementations should, where possible, use straightforward comparison of characters, rather than the routine LSAME used by the model implementation.

4. THE TEST PROGRAMS

A separate test program exists for each of the four data types (REAL, COMPLEX, DOUBLE PRECISION, and COMPLEX*16). All test programs conform to the same pattern with only the minimum necessary changes, so we shall talk generically about "the test program" in the singular.

The program has been designed not merely to check whether the model implementation has been correctly installed, but also to serve as a validation tool, and even as a modest debugging aid, for any specialized implementation.

The program has the following features:

—The parameters of the test problems and the names of the subprogram to be tested are specified by means of a data file, which can easily be modified for debugging.

—The data for the test problem are generated internally, and the results are checked internally.

—The program checks that no arguments are changed by the routines except the designated output vector or matrix.

—All error exits (caused by illegal parameter values) are tested.

—The program generates a concise summary report on the tests and optionally can generate a "history" or "snapshot" file as an additional debugging aid.

4.1 Parameters of the Test Problems

Each test problem (i.e., each call of a subprogram to be tested) depends on a choice of values for the following parameters (where relevant to the particular subprograms):

—the dimensions m and n;

—the bandwidth arguments k, kl, and ku;

—the options *UPLO*, *TRANS*, and *DIAG*;

—the increments *INCX* and *INCY*; and

—the scalars α and β.

All relevant combinations of the options *UPLO*, *TRANS*, and *DIAG* are tested. The values of the other arguments are defined by a data file. Specifically, the program reads in a set S_n of values of n, a set S_k of values of k, a set S_{inc} of values for *INCX* and *INCY*, a set S_α of values for α, and a set S_β of values for β.

For subprograms that require a second dimension m, two values of m are generated for each value of n, namely, $m = \max(n - [n/2] - 1, 0)$ and $m = \min(n + [n/2] + 1, n_{\max})$, where $n_{\max}$ is the maximum value permitted by the array dimensions in the program. If two bandwidth arguments kl and ku are required, they are generated from k by $kl = \max(k - 1, 0)$ and ku and k.

The test problems are then generated in a nested loop structure:

```
for $n \in S_n$
  for $k \in S_k$
    for all relevant values of UPLO, TRANS, and DIAG
      for $INCX \in S_{inc}$
        for $INCY \in S_{inc}$
          for $\alpha \in S_\alpha$
            for $\beta \in S_\beta$.
```

(Of course, arguments not relevant to the routine are omitted from the loop structure.) If $m = 0$ or $n = 0$ (a null problem), only one test with these values of m and n is generated.

Obviously, the sets S_n, S_k, etc., should be as small as possible; otherwise, a very large number of problems will be generated, and the test program will take a forbiddingly long time to run. On the other hand, for a comprehensive test it is essential to exercise all segments of the code and all special or extreme cases such as $n = 0$, $n = 1$, $k = 0$, $k = n - 1$, $INCX = 1$, $INCY = 1$, $\alpha = 0$, $\alpha = 1$, $\beta = 0$, and $\beta = 1$. Note that we cannot be sure what cases will be regarded as special or extreme in any specialized implementation.

A data file that specifies sets of parameters suitable for many machines is supplied with the test program, but installers and implementors must be alert to the possible need to extend or modify them (see the Appendix).

4.2 Data for the Test Problems

Data for the elements of the matrix A and the vectors x and y are generated using a simple portable congruential number generator. Values for the elements of A are uniformly distributed over $(-0.5, 0.5)$, and for the elements of x and y over $(0, 1)$. Care is taken to ensure that the data have full working accuracy. Some of the vectors have selected elements set to 0 so that special code for this case (see Section 2) can be tested. When *DIAG* = 'N', 1.0 is added to the diagonal elements of triangular matrices to ensure they are reasonably well conditioned.

Data for each test problem are first stored in a conventional two-dimensional array for the matrix A and in contiguous one-dimensional arrays for the vectors x and y. The matrix is stored as a full square or rectangular matrix, with all zero elements and unit diagonal elements stored explicitly. This form is used to compute the correct result, using the same simple code in each case.

The data are then copied into the arrays that will be passed to the subprogram being tested, taking into account the storage scheme required for the matrix, and of the values of *INCX* and *INCY*. The argument *LDA* is chosen to be 1 more than its minimum permitted value; that is, $LDA = m + 1$ for the GE routines; $n + 1$ for the SY, HE, and TR routines; $kl + ku + 2$ for the GB routines; and $k + 2$ for the SB, HB, and TB routines. (If this value exceeds $n_{\max}$, *LDA* is set equal to $n_{\max}$.)

Elements in these arrays that are not to be referenced by the subprogram (e.g., the subdiagonal elements of A when *UPLO* = 'U', or intervening elements of X when $INCX > 1$) are set to a "rogue" value (-10^{10}) to increase the likelihood that a reference to them will be detected. If a fatal error is reported and an element of the computed result is of order 10^{10}, then the routine has almost certainly referenced the wrong element of an array.

4.3 Checking the Results

After each call of a subprogram being tested, its operation is checked in two ways: First, each of its arguments, including all elements of the array arguments, is checked to see if it has been changed by the subprogram. If any argument, other than the specified elements of the result vector or matrix, has been changed, a fatal error is reported. (This check includes the supposedly unreferenced elements of the arrays, which were initialized with a rogue value.)

Second, the result vector or matrix computed by the subprogram is compared with the result computed by simple FORTRAN code. We do not expect exact agreement, because the two results are not necessarily computed by the same sequences of floating-point operations. We do, however, expect the differences to be insignificant to working precision in the following precise sense: In the MV routines, each element of the result vector is defined by an expression of the form

$$y_i \leftarrow \alpha a_{i\cdot}^T x + \beta y_i,$$

where $a_{i\cdot}^T$ denotes the ith row of A. (For the triangular matrix routines _TRMV, _TBMV, and _TPMV, we have $\alpha = 1$ and $\beta = 0$.) This expression may be regarded as a simple inner product $y_i \leftarrow u^T v$ by writing $u^T = (\alpha a_{i\cdot}^T, y_i)$, $v^T = (x^T, \beta)$. The absolute error in the computed inner product y_i is bounded by

$$|\hat{y}_i - y_i| \leq n\varepsilon |u|^T |v|,$$

where ε is the relative machine precision, and $|u|^T$ denotes the vector $(|u_1|, |u_2|, \ldots, |u_n|)^T$ (see [8, p. 36]). In our tests we have also allowed for errors introduced in the multiplication by α. On the other hand, the above bound is usually a substantial overestimate. We use the following semiempirical approach: For each element y of the result, the program computes the test ratio

$$\frac{|\hat{y}_i - y_i|}{\varepsilon |u|^T |v|},$$

with u and v defined as above. This is compared with a constant threshold value, which is defined in the data file. Test ratios greater than the threshold are flagged as "suspect." On the basis of experience, a threshold value of 16 is recommended (the largest value observed on a variety of machines has been 11.5). The precise value is not critical. Errors in the routines are most likely to be errors in array indexing, which will almost certainly lead to a totally wrong result. A more subtle potential error is the use of a single-precision variable in a double-precision computation. This is likely to lead to a loss of half the machine precision. The test program regards a test ratio greater than $\varepsilon^{-1/2}$ as a fatal error.

The R and R2 routines are checked in a similar way. Each element of the result is regarded as an inner product of length 2 or 3:

$$a_{ij} \leftarrow (a_{ij}, \alpha x_i)^T \begin{pmatrix} 1 \\ y_j \end{pmatrix}$$

or

$$a_{ij} \leftarrow (a_{ij}, \alpha x_i, \alpha y_i)^T \begin{pmatrix} 1 \\ y_j \\ x_j \end{pmatrix}.$$

The SV routines are checked as follows: If $y = T^{-1}x$ is the exact result and $\hat{y}$ is the computed result, then the test program computes $\hat{x} = T\hat{y}$ and compares it with x, as above. Thus, the test ratio is

$$\frac{|\hat{x}_i - x_i|}{\varepsilon |t_{i\cdot}|^T |x_i|},$$

where $t_{i\cdot}$ denotes the ith row of T. Theoretically, the test ratio should involve the condition number of T with respect to inversion, but the test program generates well-conditioned triangular matrices, and in practice the test ratios observed for these routines are no larger than for the others.

APPENDIX: Installation Notes

A1. Installing the Model Implementation

The subprograms fall into four sets according to the data type of the matrices and vectors: REAL, COMPLEX, DOUBLE PRECISION, and COMPLEX*16 (subprogram names beginning with S, C, D, and Z, respectively). Choose which set or sets are to be installed.

Examine the auxiliary subprograms XERBLA and LSAME (which are independent of the data type), and consider whether they need to be modified.

The subprogram XERBLA is called when one of the Level 2 BLAS detects an illegal value of one of its arguments. The version supplied with the model implementation writes a message to the standard output channel, for example,

```
** On entry to STRSV parameter number 3 had an illegal value.
```

and then executes a STOP statement. Installers may wish to redirect the error message to a different output channel, or to replace the STOP statement by a call to system-specific exception-handling or trace-back mechanisms.

The logical function LSAME is used to perform all character comparisons in Level 2 BLAS in a case-insensitive manner. For example, the expression

LSAME(*UPLO*, 'U')

is equivalent to

(*UPLO* .EQ. 'U').OR.(*UPLO* .EQ. 'u').

The supplied version works correctly on all systems that use the ASCII code for internal representation of characters. For systems that use the EBCDIC code, one constant must be changed. For CDC systems with 6–12-bit representation, alternative code is provided in comments. Any of the versions work correctly on all systems if only uppercase characters are passed as arguments.

Compile the chosen sets of subprograms, together with LSAME and XERBLA, and create an object library.

A2. Testing the Model Implementation

Select the test program or programs corresponding to the data types handled by the subprograms that have been installed.

An annotated example of a data file for the program can be obtained by editing the comments at the start of the main program. This defines the names and unit numbers of the output files, various parameters of the tests, and the names of those subprograms that are to be tested. The data file for the REAL routines is illustrated in Figure 2. The first 18 records are read using list-directed input; the last 16 are read using the format (A6, L2).

Change the first record of the data file, if necessary, to ensure that the file name is legal on your system. No other changes to the data file should be

```
Record no.   Record contents

  1    'SBLAT2.SUMM'      NAME OF SUMMARY OUTPUT FILE
  2    6         UNIT NUMBER OF SUMMARY FILE
  3    'SBLAT2.SNAP'      NAME OF SNAPSHOT OUTPUT FILE
  4    -1        UNIT NUMBER OF SNAPSHOT FILE (NOT USED IF .LT. 0)
  5    F      LOGICAL, T TO REWIND SNAPSHOT FILE AFTER EACH RECORD.
  6    F      LOGICAL, T TO STOP ON FAILURES.
  7    T      LOGICAL, T TO TEST ERROR EXITS.
  8    16.0  THRESHOLD VALUE OF TEST RATIO
  9    6                  NUMBER OF VALUES OF N
 10    0 1 2 3 5 9        VALUES OF N
 11    4                  NUMBER OF VALUES OF K
 12    0 1 2 4            VALUES OF K
 13    4                  NUMBER OF VALUES OF INCX AND INCY
 14    1 2 -1 -2          VALUES OF INCX AND INCY
 15    3                  NUMBER OF VALUES OF ALPHA
 16    0.0 1.0 0.7        VALUES OF ALPHA
 17    3                  NUMBER OF VALUES OF BETA
 18    0.0 1.0 0.9        VALUES OF BETA
 19    SGEMV  T PUT F FOR NO TEST. SAME COLUMNS.
 20    SGBMV  T PUT F FOR NO TEST. SAME COLUMNS.
 21    SSYMV  T PUT F FOR NO TEST. SAME COLUMNS.
 22    SSBMV  T PUT F FOR NO TEST. SAME COLUMNS.
 23    SSPMV  T PUT F FOR NO TEST. SAME COLUMNS.
 24    STRMV  T PUT F FOR NO TEST. SAME COLUMNS.
 25    STBMV  T PUT F FOR NO TEST. SAME COLUMNS.
 26    STPMV  T PUT F FOR NO TEST. SAME COLUMNS.
 27    STRSV  T PUT F FOR NO TEST. SAME COLUMNS.
 28    STBSV  T PUT F FOR NO TEST. SAME COLUMNS.
 29    STPSV  T PUT F FOR NO TEST. SAME COLUMNS.
 30    SGER   T PUT F FOR NO TEST. SAME COLUMNS.
 31    SSYR   T PUT F FOR NO TEST. SAME COLUMNS.
 32    SSPR   T PUT F FOR NO TEST. SAME COLUMNS.
 33    SSYR2  T PUT F FOR NO TEST. SAME COLUMNS.
 34    SSPR2  T PUT F FOR NO TEST. SAME COLUMNS.
```

Fig. 2. Data file for REAL routines.

Table II. Test Program Timing (in minutes)

	S	C	D	Z
CRAY-1S	0.2	0.2	0.5	—
DEC VAX-11/750	3.0	4.5	4.0	8.0
NSC 32032 PC[a]	8.0	10.0	17.0	20.0
Compaq Deskpro 286[b]	12.0	44.0	24.0	50.0

[a] DSI-32 coprocessor 10 MHz; Green Hills FORTRAN; compiler options -01 -02 -X71; RAM disk for I/O.
[b] Lahey F77L V.2.10; 80287, 5 MHz; 80286, 8 MHz.

necessary before an initial run of the test program, but some changes may be needed to ensure that the tests are sufficiently thorough (see below).

The data file is read from unit NIN, which is set to 5 in a PARAMETER statement in the main program: Change this if necessary.

Compile the test program, link in the required subprograms, and run the program.

Note that the test programs include an alternative version of the auxiliary subprogram XERBLA that is needed to test the error exits from Level 2 BLAS. On some systems special action must be taken to ensure this version is linked into the test program. If the model implementation of XERBLA is used, the test program will stop after writing an error message from XERBLA.

Table II gives the approximate times taken to run the test programs, using the supplied data file and the model implementation of the subprograms, on various machines.

If the tests using the supplied data file are completed successfully, consider whether the tests have been sufficiently thorough. For example, on a machine with vector registers, at least one value of N greater than the length of a vector register should be used; otherwise, important parts of the compiled code may not be exercised by the tests.

The tests may fail, with either "suspect results" or "fatal errors." Suspect results, with a test ratio slightly greater than the threshold, are probably caused by anomalies in floating-point arithmetic on the machine; if this explanation is considered to be sufficient, increase the value of the threshold specified in the data file. Fatal errors most probably indicate a compilation error or corruption of the source text. An error detected by the system, for example, an array subscript out of bounds or use of an unassigned variable, is almost certainly due to the same causes. If the system does not provide adequate post-mortem information about the error, the snapshot file can give a little help (see Section A5).

A3. Testing a Specialized Implementation

Proceed initially as described in Section A2. If the implementation does not use an error-handling subprogram XERBLA, compatible with the model implementation, then the data file must be modified to suppress the testing of error exits.

Consider very carefully what changes need to be made to the data file, to ensure the implementation has been thoroughly tested. For example, if the technique of loop unrolling is applied, make sure sufficient values of N are used

Table III. Symbolic Constants in the Test Program

Name	Meaning	Value
NIDMAX	Maximum number of values of *N*	9
NKB-MAX	Maximum number of values of *K*	7
NINMAX	Maximum number of values of *INCX*, *INCY*	7
NALMAX	Maximum number of values of *ALPHA*	7
NBEMAX	Maximum number of values of *BETA*	7
NMAX	Maximum value of *N*	65
INCMAX	Maximum value of *ABS*(*INCX*), *ABS*(*INCY*)	2

to test all the cleanup code; if *ALPHA* .EQ. −1.0 is treated as a special case, add −1.0 to the values of *ALPHA*.

A4. Changing the Parameters of the Tests

The values supplied in the data file must satisfy certain restrictions, defined by the symbolic constants in the test program shown in Table III. If necessary, modify the PARAMETER statements that define these symbolic constants.

A5. The History or Snapshot File

The main output file from the test program contains a concise report on the success or failure of the tests of each routine and the reasons for failure if it occurs. Optionally, the program writes to a separate file a one-line record giving details of the arguments in each call of a Level 2 BLAS subprogram; for example,

25: STRSV('*U*', '*T*', '*U*', 3, *A*, 4, *X*, −1).

(The number 25 indicates that this is the 25th call of STRSV.) The record is written immediately before the routine is called.

As a cumulative "history" file, this enables the user to monitor which tests are passed successfuly before a failure occurs. Moreover, if an exception occurs in the Level 2 BLAS routine (e.g., an array bound error or division by zero), the last record written to the file should give details of the call that caused the exception. On some systems, however, the output buffers are not emptied when a program is terminated abnormally. Therefore, the program has an option to rewind the file after each record is written in order to force emptying of the buffer: In this mode the file presents a one-line "snapshot" of the current or most recent call to a Level 2 BLAS routine.

REFERENCES

1. Cowell, W. R., and Thompson, C. P. Transforming Fortran DO loops to improve performance on vector architectures. Rep. ANL-85-63, Argonne National Laboratory, Argonne, Ill., 1985.
2. Daly, C., and Du Croz, J. J. Performance of a subroutine library on vector-processing machines. *Comput. Phys. Commun. 37* (1985), 181–186.
3. Dongarra, J. J., and Eisenstat, S. C. Squeezing the most out of an algorithm in CRAY FORTRAN. *ACM Trans. Math. Softw. 10*, 3 (Sept. 1984), 219–230.
4. Dongarra, J. J., and Sorensen, D. C. Linear algebra on high-performance computers. In *Proceedings of Parallel Computing 85*, U. Schendel, Ed., North-Holland, Amsterdam, 1986, pp. 3–32.

5. DONGARRA, J. J., GUSTAVSON, F. G., AND KARP, A. Implementing linear algorithms for dense matrices on a vector pipeline machine. *SIAM Rev. 26* (1984), 91–112.

6. DONGARRA, J. J., KAUFMAN, L., AND HAMMARLING, S. Squeezing the most out of eigenvalue solvers on high-performance computers. *Linear Algebra Appl. 77* (1986), 113–136.

7. DONGARRA, J. J., DU CROZ, J. J., HAMMARLING, S., AND HANSON, R. J. An extended set of Fortran basic linear algebra subprograms. Rep. ANL-MCS-TM-41 (rev. 3), Argonne National Laboratory, Argonne, Ill., 1986.

8. GOLUB, G. H., AND VAN LOAN, C. F. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, Md., 1983.

Received January 1987; accepted October 1987