ALGORITHM 679
A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs

JACK J. DONGARRA
University of Tennessee and Oak Ridge National Laboratory
JEREMY DU CROZ and SVEN HAMMARLING
Numerical Algorithms Group Ltd.
and
IAIN DUFF
Harwell Laboratory

This paper describes a model implementation and test software for the Level 3 Basic Linear Algebra Subprograms (Level 3 BLAS). The Level 3 BLAS are targeted at matrix–matrix 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 3 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 the Level 3 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 [4] we have defined the specification of a set of Level 3 Basic Linear Algebra Subprograms for selected matrix–matrix operations. They provide a standard framework for developing modular, portable, and efficient Fortran 77 code for

The research of J. J. Dongarra 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.

Authors' addresses: J. J. Dongarra, Computer Science Department, University of Tennessee, Knoxville, TN 37996-1301; J. Du Croz and S. Hammarling, Numerical Algorithms Group Ltd., Wilkinson House, Jordan Hill Road, Oxford OX2 8DR, England; I. Duff, Computer Science and Systems Division, Harwell Laboratory, Oxfordshire OX11 0RA, England.

block algorithms for many computational problems in linear algebra. We also anticipate that they will be useful building blocks in other areas of numerical software. Our hope is that specialized implementations of the Level 3 BLAS will be developed for high-performance computers, and thus programs that call the Level 3 BLAS can be efficient across a wide range of machines.

To support and encourage the use of the Level 3 BLAS, this paper describes two software components:

(1) A model implementation of the subprograms in Fortran 77. This enables the Level 3 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 (see Section 4).

Section 3 contains some advice on developing specialized implementations of the subprograms.

2. THE MODEL IMPLEMENTATION

2.1 Programming Considerations

There are many mathematically equivalent ways to implement the Level 3 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, whenever possible. 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) Provision is made to skip the innermost loop if relevant elements of the matrices are zero. This can yield a considerable gain in efficiency if there are zero entries in the input matrices, for example:

```
      IF( B(L,J).NE.ZERO )THEN
         TEMP = ALPHA*(B(L,J)
         DO 50, I = 1, M
            C(I,J) = C(I,J) + TEMP*A(I,L)
   50    CONTINUE
      END IF
```

2.2 Efficiency

The model implementation has not been designed to be particularly efficient on any specific machine, although the organization of the code is such that it should perform tolerably well on conventional scalar machines with a good optimizing compiler and possibly on some single-processor vector machines with a good optimizing compiler. Our main aim is a clear and straightforward implementation of the algorithms in the same style as the model implementation of the Level 2 BLAS [6]. Section 3 discusses how to implement the Level 3 BLAS efficiently.

2.3 Language Standards

The model implementation of the Level 3 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;

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 uppercase 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 obstacle to portability on some systems (as is discussed in Section 7 [5]), but we have avoided most of the problems by using the auxiliary LOGICAL function LSAME described in the Appendix.

2.4 Auxiliary Subprograms

Two auxiliary subprograms are called by the Level 3 BLAS: an error-handling routine XERBLA and the character-comparison routine LSAME. They are identical to the subprograms of the same names in the model implementation of the Level 2 BLAS. 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 the Level 3 BLAS. The following broad possibilities should be considered:

(a) Rewriting the algorithms in Fortran so that the structure of the inner loops is better adapted to the architecture of the machine.

(b) Using calls to Level 2 BLAS.

(c) Using machine-specific extensions to Fortran, such as array-syntax, compiler-directives, or calls to library routines.

(d) Coding the routines in assembly language.

Approaches (b), (c), and (d) should be considered as extensions of (a), not as alternatives. Implementers should not consider translating the model implementation into calls to Level 2 BLAS, extended Fortran, or assembly language without first considering whether the structure of the model implementation is well adapted to their machine.

We wish to draw the attention of implementers to the following possible approaches to restructuring the code:

(1) *Permuting the Nesting of Loops.* Each matrix–matrix operation performed by the Level 3 BLAS involves triply nested loops. The code may be reorganized in six different ways by permuting the order of nesting of the loop indices, as described in [3]. Different organizations are likely to offer advantages on different machines in terms of maximizing the use of vector registers, and minimizing the use of cache memory or exploitation of loop-based parallelism, for example.

(2) *Introducing a Block Structure.* Additional loops may be added to the code in order to break the computation into operations on submatrices of suitable size. As in (1), this may be important either to minimize memory traffic (if the submatrices being operated on can all be stored in high-speed cache or local memory) or to exploit parallelism (if operations on separate submatrices can be performed in parallel).

(3) *Using Alternative Segments of Code for Different Shapes and Sizes of Matrices.* It is essential that implementers should not confine their efforts to achieving high performance only when all the matrices are large and square. In the intended applications of the Level 3 BLAS (as illustrated in Section 9 of [4], for example), the matrices are often long and thin, and good performance in this case is equally important. It is likely that some implementations need to switch between different segments of code, according to the absolute or relative values of the matrix dimensions m, n, and k.

(4) *Using Work Arrays to Allow Contiguous Storage.* On some machines performance is significantly degraded if vectors are not stored contiguously. In order to allow contiguous storage it may be necessary to copy a row or rows of a matrix into a temporary work vector.

As a general guideline, efficient implementations are likely to be achieved by reducing the ratio of memory traffic to arithmetic operations, making full use of vector operations (if available), and exploiting parallelism (if available).

However, note that in some environments exploitation of fine-grain parallelism within the Level 3 BLAS may interfere with possible coarse-grain parallelization at higher levels in the program.

On many high-performance computers, achieving an optimal implementation of the Level 3 BLAS is likely to be a more difficult task than achieving the same for the Level 2 BLAS. However, we believe that it will be well worth the effort since it will enable a wide range of higher level algorithms to be transported onto those machines without serious loss of efficiency [1].

Timing programs for all the Level 3 BLAS are available from netlib [2] or from the authors.

As an illustration of what can be achieved we show in Table I the speed of implementations of particular Level 1, 2, and 3 BLAS routines or routines of similar functionality, on three different machines. The routines are for Level 1: BLAS, SDOT/DDOT, and SAXPY/DAXPY; for Level 2: BLAS, SGEMV/DGEMV (or equivalent for matrix–vector multiply); for Level 3:

Table I. Performance in Mflops of BLAS

				Machine		
				CRAY-2	IBM 3090/VF	Alliant FX/8
Peak Performance		Memory reference	Flops	488	108	88
Level 1	SDOT/DDOT	$2n$	$2n$	151	32	14
	SAXPY/DAXPY	$3n$	$2n$	121	26	14
Level 2	SGEMV/DGEMV	n^2	$2n^2$	350	60	26
Level 3	SGEMM/DGEMM	$3n^2$	$2n^3$	437	80	43

Note: Memory references and flops are for matrices and vectors of order n. The Mflop rates are the maximum rates for large n.

BLAS, SGEMM/DGEMM (or equivalent for matrix–matrix multiply). The speeds quoted are the maximum speeds observed for tuned machine-specific implementations of the routines. We give expressions for the amount of memory traffic and the number of floating-point operations (flops) for each calculation.

We show the results of timing the routines on three computers: the CRAY-2 (with vector register and local memory), the IBM 3090/VF (with cache and vector registers), and the Alliant FX/8 (with parallel processors, cache, and vector registers). We also show the manufacturer's peak performance figures for each machine. Apart from the Alliant figures (limited by the speed of the memory-to-cache transfer), the Level 3 routines attain a performance remarkably close to the peak, and the clear advantage of using higher level BLAS is seen on all machines. These figures were obtained on lightly loaded machines. Because the Level 3 BLAS make fewer demands on memory, the comparison is likely to be even more favorable on more heavily loaded machines.

Specialized implementations should, where possible, use a 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 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 subprograms 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, n, and k; the options SIDE, UPLO, TRANS, and DIAG; and the scalars α and β.

All relevant combinations of the options SIDE, 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 for m, n, and k, a set S_α of values for α, and a set S_β of values for β.

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

```
for $m \in S_n$
  for $n \in S_n$
    for $k \in S_n$
      for all relevant values of SIDE, UPLO, TRANS, and DIAG
        for $\alpha \in S_\alpha$
          for $\beta \in S_\beta$.
```

(Of course, arguments not relevant to the routine are omitted from the loop structure.)

Obviously, the sets S_n, S_α, and S_β should be as small as possible; otherwise, a very large number of problems are generated, and the test program takes 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 $m, n, k = 0$ or 1, $\alpha = 0$, $\alpha = 1$, $\beta = 0$, $\beta = 1$. Note that we cannot be sure what cases are regarded as special or extreme in any specialized implementation.

A data file, which specifies sets of parameters suitable for many machines, is supplied with the test program, but installers and implementers 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 matrices A, B, and C are generated using a simple, portable congruential number generator. Values for the matrix elements are uniformly distributed over $(-0.5, 0.5)$. Care is taken to ensure that the data have full working accuracy. Some of the matrix elements are set to zero so that special code (see Section 2) can be tested. When DIAG = 'N', 1.0 is added to the diagonal elements of triangular matrices to ensure that they are reasonably well conditioned. For each call of a routine the argument LDA is set to $min(\text{LDAMIN} + 1, n_{\max})$, where LDAMIN is the minimum permitted value of LDA for that call, and $n_{\max}$ is the maximum value permitted by the array dimensions in the program; LDB and LDC are set in the same way.

Elements in the arrays that are not to be referenced by the subprogram (e.g., the subdiagonal elements when UPLO = 'U') are set to a "rogue" value (-10^{10}) to increase the likelihood that a reference to them is 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 whether it has been changed by the subprogram. If any argument, other than the specified elements of the result 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 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 sense.

In the matrix multiply routines, each element of the result vector is defined by an expression of the form

$$c_{ij} \leftarrow \alpha \sum_{l=1}^{k} a_{il} b_{lj} + \beta c_{ij}.$$

The absolute error in the computed inner product and subsequent arithmetic is bounded by

$$|\hat{c}_{ij} - c_{ij}| \le \epsilon n t_{ij}$$

where

$$t_{ij} = |\beta| |c_{ij}| + |\alpha| \sum_{l=1}^{k} |a_{il}| |b_{lj}|$$

For each element c_{ij} of the result, the program computes the test ratio:

$$\frac{|\hat{c}_{ij} - c_{ij}|}{\epsilon t_{ij}}.$$

Theoretically, this test ratio could be as large as n, but in practice we have never observed such growth, and so the ratio 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 9.7). The precise value is not critical. Errors in the routines are most likely to be errors in array indexing, which 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 $\epsilon^{-1/2}$ as a fatal error.

Similar tests are performed for the other Level 3 BLAS routines, in each case separating suspect results from obviously inaccurate ones.

Note that the test programs make no attempt to generate data that would reveal potential numerical instability in an implementation of the Level 3 BLAS. (This would be a vain task without detailed knowledge of the algorithm.) Passing the tests should not be taken to mean that the requirement for numerical stability (see Section 7 of [4]) has been satisfied.

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 3 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 STRSM 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 traceback mechanisms.

The logical function LSAME is used to perform all character comparison in the Level 3 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 1. The first 14 records are read using list-directed output, and the last 6 records are read using the format (A6, L2).

Change the first and third records of the data file, if necessary, to ensure that the file name is legal on your system. No other changes to the data files should be necessary before an initial run of the test program, but some changes may be needed to ensure that the tests are sufficiently thorough (see A3, A4, and A5).

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.

```
Record no.  Record contents

 1          'SBLAT3.SUMM'     NAME OF SUMMARY OUTPUT FILE
 2          6                 UNIT NUMBER OF SUMMARY FILE
 3          'SBLAT3.SNAP'     NAME OF SNAPSHOT OUTPUT FILE
 4          -1                UNIT NUMBER OF SNAPSHOT FILE (NOT USED IF .LT. 0)
 5          F        LOGICAL FLAG, T TO REWIND SNAPSHOT FILE AFTER EACH RECORD.
 6          F        LOGICAL FLAG, T TO STOP ON FAILURES.
 7          T        LOGICAL FLAG, 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          3                 NUMBER OF VALUES OF ALPHA
12          0.0 1.0 0.7       VALUES OF ALPHA
13          3                 NUMBER OF VALUES OF BETA
14          0.0 1.0 1.3       VALUES OF BETA
15           SGEMM  T PUT F FOR NO TEST. SAME COLUMNS.
16           SSYMM  T PUT F FOR NO TEST. SAME COLUMNS.
17           STRMM  T PUT F FOR NO TEST. SAME COLUMNS.
18           STRSM  T PUT F FOR NO TEST. SAME COLUMNS.
19           SSYRK  T PUT F FOR NO TEST. SAME COLUMNS.
20           SSYR2K T PUT F FOR NO TEST. SAME COLUMNS.
```

Figure 1

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

Note: the test program includes a special version of the auxiliary subprogram XERBLA. This is for use with the test program only. It is needed to check that XERBLA is called by the Level 3 BLAS if and only if an error exit is intended. When the Level 3 BLAS are linked into the test program, they must be linked to this special version of XERBLA. If the model implementation of XERBLA is used for testing, the test program stops prematurely 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.

Table II. Execution Time in Seconds of Test Programs on Different Machines

	S	C	D	Z
SUN 3/260	264	500	728	1,423
VAX 8800 VMS	34	60	40	102
IBM 3090/VF	8	15	8	15
CRAY-2	9	14	121	—

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 postmortem 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 that the implementation has been thoroughly tested. For example, if the technique of loop unrolling is applied, make sure that sufficient values of N are used to test all the clean-up 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 following symbolic constants in the test program:

Name	Meaning	Value
NIDMAX	Maximum number of values of M, N, and K	9
NALMAX	Maximum number of values of ALPHA	7
NBEMAX	Maximum number of values of BETA	7
NMAX	Maximum value of M, N, K	65

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 also writes to a separate file a one-line record giving details of the arguments in each call of a Level 3 BLAS subprogram, for example

25: STRSM ('L,' 'U,' 'T,' 'U,' 10, 5, −1.0, *A*, 11, *B*, 5)

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

As a cumulative "history" file, this record enables the user to monitor which tests are passed successfully before a failure occurs. Moreover, if an exception occurs in the Level 3 BLAS routine (e.g., array bound error or division by zero), the last record written to the file should give details of the call that caused the exception. However, on some systems the output buffers are not emptied when a program is terminated abnormally. If the logical flag in line 5 of the data file is set to T, then the program rewinds 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 3 BLAS routine.

REFERENCES

1. Demmel, J. W., Dongarra, J. J., Du Croz, J., Greenbaum, A., Hammarling, S., and Sorensen, D. Prospectus for the development of a linear algebra library for high-performance computers. Argonne National Laboratory Report, ANL-MCS-TM-97, Argonne, Ill., Sept. 1987.
2. Dongarra, J. J., and Grosse, E. Distribution of mathematical software via electronic mail. *Commun. ACM 30*, 5 (May 1987), 403–407.
3. Dongarra, J. J., Gustavson, F., and Karp, A. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Rev. 26*, 1 (Jan. 1984), 91–112.
4. Dongarra, J. J., Du Croz, J., Duff, I., and Hammarling, S. A set of level 3 basic linear algebra subprograms. This issue, pp. 1–17.
5. Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math Softw. 14*, 1 (Mar. 1988), 1–17.
6. Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. Algorithm 656: An extended set of basic linear algebra subprograms: Model implementation and test programs. *ACM Trans. Math. Softw. 14*, 1 (Mar. 1988), 18–32.

Received October 1988; revised February 1989; accepted March 1989