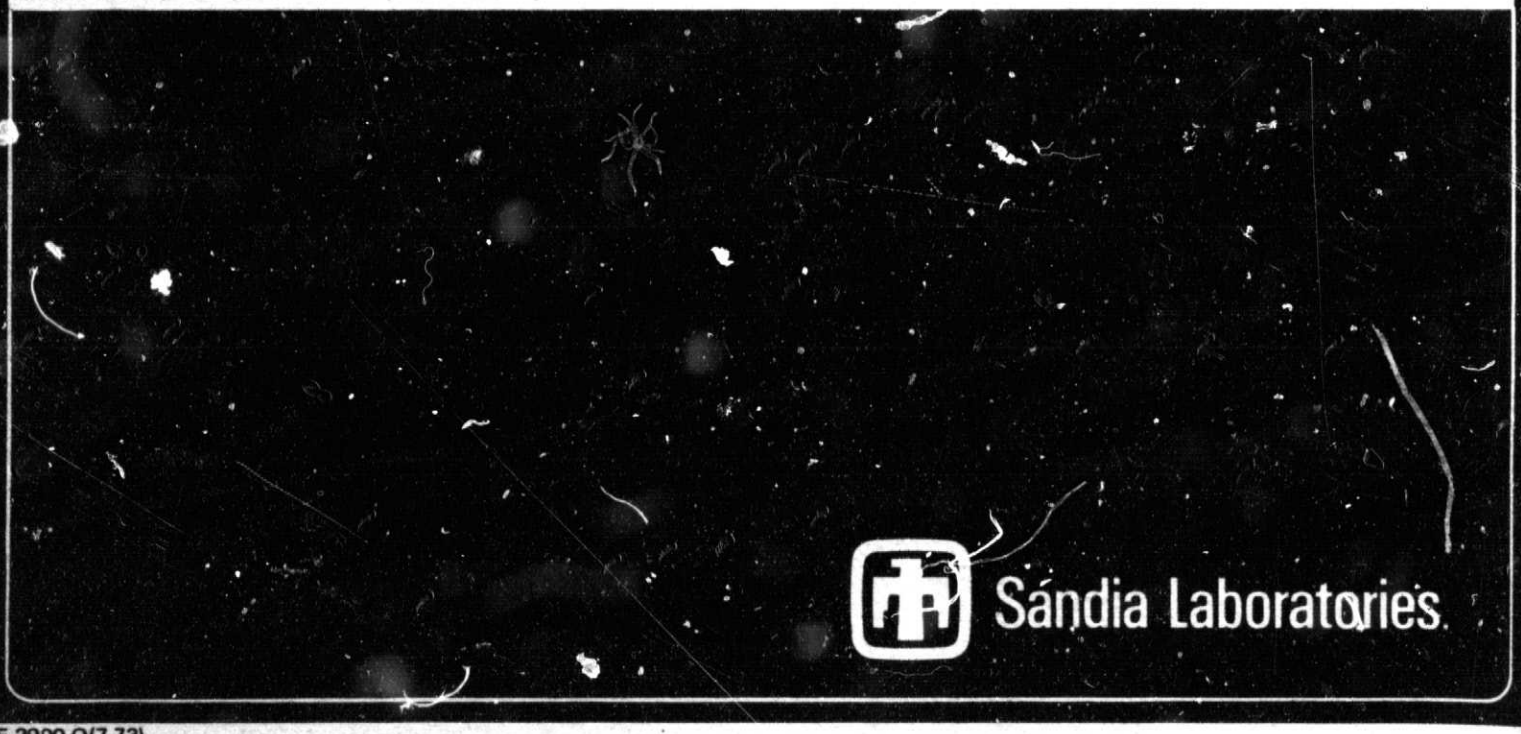
SAND77-0895
Unlimited Release

Basic Linear Algebra Subprograms for FORTRAN Usage

Chuck L. Lawson, Richard J. Hanson, David R. Kincaid, Fred T. Krogh



SF 2900 Q(7-73)

SAND77-0898
Unlimited Release
Printed October 1977

BASIC LINEAR ALGEBRA SUBPROGRAMS FOR FORTRAN USAGE

C. L. Lawson, Jet Propulsion Laboratory
R. J. Hanson, Sandia Laboratories, Albuquerque
D. R. Kincaid, University of Texas, Austin
F. T. Krogh, Jet Propulsion Laboratory

Abstract

A package of 38 low-level subprograms for many of the basic operations of numerical linear algebra is presented. The package is intended to be used with FORTRAN. The operations in the package are dot products, elementary vector operations, Givens transformations, vector copy and swap, vector norms, vector scaling, and the indices of components of largest magnitude.

The subprograms and a test driver are available in portable FORTRAN. Versions of the subprograms are also provided in assembly language for the IBM 360/67, the CDC 6600 and CDC 7600, and the Univac 1108.

Printed in the United States of America

Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, Virginia 22161
Price: $4.~~00~~ 5.00, Microfiche $3.00

Table of Contents

BASIC LINEAR ALGEBRA SUBPROGRAMS FOR FORTRAN USAGE

C. L. Lawson, Jet Propulsion Laboratory
R. J. Hanson, Sandia Laboratories, Albuquerque
D. R. Kincaid, University of Texas, Austin
F. T. Krogh, Jet Propulsion Laboratory

1. Introduction

This paper describes a package, called the BLAS, of thirty-eight FORTRAN-callable subprograms for basic operations of numerical linear algebra. This paper and the associated package of subprograms and testing programs are the result of a collaborative voluntary project of the ACM-SIGNUM committee on basic linear algebra subprograms. This project was carried out during the period 1973-1977.

The initial version of the subprogram specifications appeared in Ref. [1]. Following distribution of Ref. [1] to persons active in the development of numerical linear algebra software, open meetings of the project were held at the Purdue Mathematical Software II Conference, May, 1974, Ref. [2], and at the National Computer Conference, Anaheim, May, 1975. Extensive modifications of the specifications were made following the Purdue meeting which was attended by thirty people. A few additional changes resulted from the Anaheim meeting. Most of the further Fortran code changes resulted from an effort to improve the design and to make them more robust.

2. Reasons for Developing the Package

Designers of computer programs involving linear algebraic operations have frequently chosen to implement certain low-level operations such as the dot product as separate subprograms. This may be observed both in many published codes and in codes written for specific applications at many computer installations. Following are some of the reasons for taking this approach:

(1) It can serve as a conceptual aid in both the design and coding stages of a programming effort to regard an operation such as the dot product as a basic building block. This is consistent with the ideas of structured programming which encourage modularizing common code sequences.

(2) It improves the self-documenting quality of code to identify an operation such as the dot product by a unique mnemonic name.

(3) Since a significant amount of the execution time in complicated linear algebraic programs may be spent in a few low-level operations, a reduction of execution time spent in these operations may be reflected in cost savings in the running of programs. Assembly language coded subprograms for these operations provide such savings on some computers.

(4) The programming of some of these low-level operations involves algorithmic and implementation subtleties that are likely to be ignored in the typical applications programming environment. For example the subprograms provided for the modified Givens transformation incorporate control of the scaling terms which otherwise can drift monotonically toward underflow.

If there could be general agreement on standard names and parameter lists for some of these basic operations it would add the additional benefit of portability with efficiency on the assumption that the assembly language subprograms were generally available. Such standard subprograms would provide building blocks with which designers of portable subprograms for higher level linear algebraic operations such as solving linear algebraic equations,

eigenvalue problems, etc., could achieve additional efficiency. The package of subprograms described in this paper is proposed to serve this purpose.

3. Scope of the Package

Specifications will be given for thirty-eight FORTRAN-callable subprograms covering the operations of dot product, vector plus a scalar times a vector, Givens transformation, modified Givens transformation, copy, swap, Euclidean norm, sum of magnitudes, multiplying a scalar times a vector, and locating an element of largest magnitude. Since we are thinking of these subprograms as being used in an ANS FORTRAN context we provide for the cases of single precision, double precision, and (single precision) complex data.

In Table 1 a concise summary of the operations provided and the conventions adopted for naming the subprograms is given. Each type of operation is identified by a root name. The root name is prefixed by one or more of the letters I, S, D, C, or Q to denote operations on integer, single precision, double precision, (single precision) complex, or extended precision data types, respectively. For subprograms involving a mixture of data types the type of the output quantity is indicated by the left-most prefix letter. Suffix letters are used on four of the dot product subprograms to distinguish variants of the basic operation.

If one were to extend this package to include double precision complex type data (COMPLEX*16 in IBM FORTRAN) we suggest that the prefix Z be used in the names of the new subprograms. For example, subprograms CZDOTC and CZDOTU for the dot product of (single precision) complex vectors, with double precision accumulation, have been written for the CDC 6600. These may be obtained directly from Kincaid.

Table 1

Summary of Functions and Names
Of the Basic Linear Algebra Subprograms

Function	Prefix and Suffix of Name								Root of Name
Dot Product	SDS-	DS-	DQ-I	DQ-A	C-U	C-C	D-	S-	-DOT-
Constant Times a Vector Plus a Vector						C-	D-	S-	-AXPY
Set-up Givens Rotation							D-	S-	-ROTG
Apply Rotation							D-	S-	-ROT
Set-up Modified Givens Rotation							D-	S-	-ROTMG
Apply Modified Rotation							D-	S-	-ROTM
Copy x into y						C-	D-	S-	-COPY
Swap x and y						C-	D-	S-	-SWAP
2-Norm (Euclidean Length)						SC-	D-	S-	-NRM2
Sum of Absolute Values*						SC-	D-	S-	-ASUM
Constant Times a Vector					CS-	C-	D-	S-	-SCAL
Index of Element Having Max Absolute Value*						IC-	ID-	IS-	-AMAX

*For complex components $z_j = x_j + iy_j$ these subprograms compute $|x_j| + |y_j|$ instead of $(x_j^2 + y_j^2)^{1/2}$.

Section 5 lists all of the subprogram names and their parameter lists, and defines the operations performed by each subprogram.

The criterion for including an operation in the package was that it should involve just one level of looping and occur in the usual algorithms of numerical linear algebra such as Gaussian elimination or the various elimination methods using orthogonal transformations.

This orientation affected the specifications of SCASUM and ICAMAX particularly. Although SASUM and DASUM compute ℓ_1 norms we assumed that the usage of either of these subprograms in numerical linear algebra software would be for the purpose of computing a vector norm that was less expensive to compute than the ℓ_2 norm. Thus for the complex version, SCASUM, instead of specifying the ℓ_1 norm which would be

$$w = \sum_i \left\{ [\mathrm{Re}(x_i)]^2 + [\mathrm{Im}(x_i)]^2 \right\}^{1/2}$$

we specified the less expensive norm,

$$w = \sum_i \left\{ |\mathrm{Re}(x_i)| + |\mathrm{Im}(x_i)| \right\} .$$

Similarly, whereas ISAMAX and IDAMAX may be regarded as determining the ℓ_∞ norm of a vector, we do not regard this as the essential property to be carried over to the complex case. Thus ICAMAX is specified to find an index j such that

$$|\mathrm{Re}(x_j)| + |\mathrm{Im}(x_j)| = \max_i \left\{ |\mathrm{Re}(x_i)| + |\mathrm{Im}(x_i)| \right\}$$

rather than finding an index j such that

$$[\mathrm{Re}(x_j)]^2 + [\mathrm{Im}(x_j)]^2 = \max_i \left\{ [\mathrm{Re}(x_i)]^2 + [\mathrm{Im}(x_i)]^2 \right\}$$

In both the computation of the ℓ_2 norm and the Givens transformation

a naive computation of the squares of the given data would restrict the exponent range of acceptable data. This package avoids this restriction by making use of ideas described by Cody, Ref. [11], and Blue, Ref. [12]. Additionally, in the case of the Givens transformations, an idea of Stewart, Ref. [13], permits the storage of all the transformations of a matrix decomposition in the memory space occupied by the elements zeroed by the transformation.

The modified Givens transformation is a relatively new innovation among numerical linear algebra algorithms, Refs. [3], [4], and [5]. The significant features are the reduction of the number of multiplications, the elimination of square root operations, and the capability of removing rows of data in least squares problems. The details of this algorithm as implemented in this package are given in the Appendix.

4. Programming Conventions

Vector arguments are permitted to have a storage spacing between elements. This spacing is specified by an increment parameter. For example, suppose a vector x having components x_i, i = 1,...,N is stored in a DOUBLE PRECISION array DX( ) with increment parameter INCX. If INCX $\geq$ 0 then x_i is stored in DX(1+(i-1)*INCX). If INCX $<$ 0 then x_i is stored in DX(1+(N-i)*|INCX|). This method of indexing when INCX $<$ 0 avoids negative indices in the array DX( ) and thus permits the subprograms to be written in FORTRAN. Only positive values of INCX are allowed for operations 26-38 that each have a single vector argument.

It is intended that the loops in all subprograms process the elements of vector arguments in order of increasing vector component indices, i.e., in the order x_i, i = 1,...,N. This implies processing in reverse storage order when INCX $<$ 0. If these subprograms are implemented on a computer having

parallel processing capability, it is recommended that this order of processing be adhered to as nearly as is reasonable.

5. Specification of the BLA Subprograms

Type and dimension information for variables occurring in the subprogram specifications are as follows:

$$mx = \max(1, N*|INCX|)$$
$$my = \max(1, N*|INCY|)$$

```
INTEGER     N,INCX,INCY,IMAX
REAL        SC(mx),SY(my),SA,SB,SC,SS
REAL        SD1,SD2,SB1,SB2,SPARAM(5),SW,QC(10)
DOUBLE PRECISION   DX(mx),DY(my),DA,DB,DC,DS
DOUBLE PRECISION   DD1,DD2,DB1,DB2,DPARAM(5),DW
COMPLEX     CX(mx),CY(my),CA,CW
```

Type declarations for function names are as follows:

```
INTEGER     ISAMAX,IDAMAX,ICAMAX
REAL        SDOT,SDSDOT,SNRM2,SCNRM2,SASUM,SCASUM
DOUBLE PRECISION  DSDOT,DDOT,DQDOTI,DQDOTA,DNRM2,DASUM
COMPLEX     CDOTC CDOTU
```

Dot Product Subprograms

1. `SW = SDOT(N,SX,INCX,SY,INCY)` $\qquad w := \sum_{i=1}^{N} x_i y_i$

2. `DW = DSDOT(N,SX,INCX,SY,INCY)` $\qquad w := \sum_{i=1}^{N} x_i y_i$

   Double precision accumulation is used within the subprogram DSDOT.

3. `SW = SDSDOT(N,SB,SX,INCX,SY,INCY)` $\qquad w := b + \sum_{i=1}^{N} x_i y_i$

   Accumulation of the inner product and addition of b is in double precision. Conversion of the final result to single precision is done the same as the intrinsic function SNGL( ).

4. DW = DDOT(N,DX,INCX,DY,INCY) $\quad w := \sum_{i=1}^{N} x_i y_i$

5. DW = DQDOTI(N,DB,QC,DX,INCX,DY,INCY) $w := c := b + \sum_{i=1}^{N} x_i y_i$

The input data, b, x, and y, are converted internally to extended precision. The result is stored in extended precision form in QC( ) and returned in double precision form as the value of the function DQDOTI.

6. DW = DQDOTA(N,DB,QC,DX,INCX,DY,INCY) $\quad w := c := b + c + \sum_{i=1}^{N} x_i y_i$

The input value of c in QC( ) is extended precision. The value c must have resulted from a previous execution of DQDOTI or DQDOTA since no other way is provided for defining an extended precision number. The computation is done in extended precision arithmetic and the result is stored in extended precision form in QC( ) and is returned in double precision form as the function value DQDOTA.

7. CW = CDOTC(N,CX,INCX,CY,INCY) $\quad w := \sum_{i=1}^{N} \bar{x}_i y_i$

The suffix C on CDOTC indicates that the complex conjugates of the components x_i are used.

8. CW = CDOTU(N,CX,INCX,CY,INCY) $\quad w := \sum_{i=1}^{N} x_i y_i$

The suffix U on CDOTU indicates that the vector components x_i are used unconjugated.

In the preceding eight subprograms the value of $\sum_{i=1}^{N}$ will be set to zero if $N \leq 0$.

Elementary Vector Operation $\quad y := ax + y$

9. CALL SAXPY(N,SA,SX,INCX,SY,INCY)

10. CALL DAXPY(N,DA,DX,INCX,DY,INCY)

11. CALL CAXPY(N,CA,CX,INCX,CY,INCY)

If a = 0 or if N ≤ 0 these subroutines return immediately.

Construct Givens Plane Rotation

12. CALL SROTG(SA,SB,SC,SS)

13. CALL DROTG(DA,DB,DC,DS)

Given a and b each of these subroutines computes

$$\sigma = \begin{cases} \text{sgn}(a) & \text{if } |a| > |b| \\ \text{sgn}(b) & \text{if } |b| \geq |a| \end{cases}$$

$$r = \sigma(a^2+b^2)^{1/2}$$

$$c = \begin{cases} a/r & \text{if } r \neq 0 \\ 1 & \text{if } r = 0 \end{cases}$$

and

$$s = \begin{cases} b/r & \text{if } r \neq 0 \\ 0 & \text{if } r = 0 \end{cases}$$

The numbers c, s, and r then satisfy the matrix equation

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \cdot \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} .$$

The introduction of σ is not essential to the computation of a Givens rotation matrix but its use permits later stable reconstruction of c and s from just one stored number, an idea due to Stewart, Ref. [13]. For this purpose the subroutine also computes

$$z = \begin{cases} s & \text{if } |a| > |b| \text{ or if } a = b = 0 \\ 1/c & \text{if } |a| \leq |b| \neq 0 \text{ and } c \neq 0 \\ 1 & \text{if } |a| \leq |b| \neq 0 \text{ and } c = 0 \end{cases}$$

The subroutines return r overwriting a, and z overwriting b, as well as returning c and s.

If the user later wishes to reconstruct c and s from z it can be done as follows

If $z = 1$ set $c = 0$ and $s = 1$

If $|z| < 1$ set $c = (1-z^2)^{1/2}$ and $s = z$

If $|z| > 1$ set $c = 1/z$ and $s = (1-c^2)^{1/2}$

Apply a Plane Rotation

14. CALL SROT(N,SX,INCX,SY,INCY,SC,SS)

15. CALL DROT(N,DX,INCX,DY,INCY,DC,DS)

Each of these subroutines computes

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} := \begin{bmatrix} c & s \\ -s & \end{bmatrix} \cdot \begin{bmatrix} x_i \\ y_i \end{bmatrix} \quad \text{for} \quad i = 1,\ldots,N$$

If $N \leq 0$ or if $c = 1$ and $s = 0$ the subroutines return immediately.

Construct a Modified Givens Transformation

16. CALL SROTMG(SD1,SD2,SB1,SB2,SPARAM)

17. CALL DROTMG(DD1,DD2,DB1,DB2,DPARAM)

The input quantities d_1, d_2, b_1, and b_2 define a 2-vector $[a_1,a_2]^T$ in partitioned form as

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} d_1^{1/2} & 0 \\ 0 & d_2^{1/2} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

The subroutine determines the modified Givens rotation matrix H, as

defined in Eqs. (A.6) - (A.7) of Appendix 1 that transforms b_2, and thus a_2, to zero. A representation of this matrix is stored in the array SPARAM( ) or DPARAM( ) as follows. Locations in PARAM not listed are left unchanged.

PARAM(1) = 1	PARAM(1) = 0	PARAM(1) = -1
Case of Eq. (A.7)	Case of Eq. (A.6)	Case of rescaling
$h_{12} = 1$ $h_{21} = -1$	$h_{11} = h_{22} = 1$	PARAM(2) = h_{11}
PARAM(2) = h_{11}	PARAM(3) = h_{21}	PARAM(3) = h_{21}
PARAM(5) = h_{22}	PARAM(4) = h_{12}	PARAM(4) = h_{12}
		PARAM(5) = h_{22}

In addition PARAM(1) = -2 indicates H = I.

The values of d_1, d_2, and b_1 are changed to represent the effect of the transformation. The quantity b_2 which would be zeroed by the transformation is left unchanged in storage.

The input value of d_1 should be nonnegative, but d_2 can be negative for the purpose of removing data from a least squares problem. Further details can be found in Appendix 1.

Apply a Modified Givens Transformation

18. CALL SROTM(N,SX,INCX,SY,INCY,SPARAM)

19. CALL DROTM(N,DX,INCX,DY,INCY,DPARAM)

Let H denote the modified Givens transformation defined by the parameter array SPARAM( ) or DPARAM( ). The subroutines compute

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} := H \begin{bmatrix} x_i \\ y_i \end{bmatrix} \quad \text{for} \quad i = 1,\ldots,N$$

If $N \leq 0$ or if H is an identity matrix the subroutines return immediately. See Appendix 1 for further details.

Copy a Vector x to y y := x

20. CALL SCOPY(N,SX,INCX,SY,INCY)

21. CALL DCOPY(N,DX,INCX,DY,INCY)

22. CALL CCOPY(N,CX,INCX,CY,INCY)

Return immediately if N ≤ 0.

Interchange Vectors x and y x :=: y

23. CALL SSWAP(N,SX,INCX,SY,INCY)

24. CALL DSWAP(N,DX,INCX,DY,INCY)

25. CALL CSWAP(N,CX,INCX,CY,INCY)

Return immediately if N ≤ 0.

Euclidean Length or ℓ_2 Norm of a Vector $w := \left[\sum_{i=1}^{N} |x_i|^2\right]^{1/2}$

26. SW=SNRM2(N,SX,INCX)

27. DW=DNRM2(N,DX,INCX)

28. SW=SCNRM2(N,CX,INCX)

If N ≤ 0 the result is set to zero.

Sum of Magnitudes of Vector Components

29. SW=SASUM(N,SX,INCX)

30. DW=DASUM(N,DX,INCX)

31. SW=SCASUM(N,CX,INCX)

The functions SASUM and DASUM compute $w := \sum_{i=1}^{N} |x_i|$. The function SCASUM computes

$$w := \sum_{i=1}^{N} \left\{ |\text{Real}(x_i)| + |\text{Imag}(x_i)| \right\}$$

These functions return immediately with the result set to zero if $N \leq 0$.

Vector Scaling x := ax

32. CALL SSCAL(N,SA,SX,INCX)

33. CALL DSCAL(N,DA,DX,INCX)

34. CALL CSCAL(N,CA,CX,INCX)

35. CALL CSSCAL(N,SA,CX,INCX)

Return immediately if $N \leq 0$.

Find Largest Component of a Vector

36. IMAX=ISAMAX(N,SX,INCX)

37. IMAX=IDAMAX(N,DX,INCX)

38. IMAX=ICAMAX(N,CX,INCX)

The functions ISAMAX and IDAMAX determine the smallest index i such that $|x_i| = \max\{|x_j| : j = 1,\ldots,N\}$.

The function ICAMAX determines the smallest index i such that $|x_i| = \max\{|\text{Real}(x_j)| + |\text{Imag}(x_j)| : j = 1,\ldots,N\}$.

These functions set the result to zero and return immediately if $N \leq 0$.

6. Implementation

In addition to the FORTRAN versions, all of the subprograms except DQDOTI and DQDOTA are also supplied in assembler language for the Univac 1108, the IBM 300/67, and the CDC 6600 and 7600. The FORTRAN versions of DQDOTI and DQDOTA use part of Brent's multiple precision package, Ref. [14]. Assembler language modules for these two subprograms are given only for the Univac 1108.

Only four of the assembly routines for the CDC 6600 and 7600 take advantage of the pipeline architecture of these machines. The four routines

SDOT( ), SAXPY( ), SROT( ), and SROTM( ) are those typically used in the innermost loop of computations. Some timing results are given in section 8.

The subprograms SMCHCN and DMCHCN provide three machine dependent parameters that are used by the five routines SROTG, DROTG, SNRM2, DNRM2 and SCNRM2. These parameters are: SMALL ≡ smallest positive floating point number, BIG ≡ biggest positive floating point number, and EPS ≡ relative arithmetic precision. They are computed by use of subprograms SMCHAR and DMCHAR. These two subprograms were provided by W. J. Cody. An individual computer installation may wish to remove Cody's routines and simply have the subprograms SMCHCN and DMCHCN return the appropriate constants. The test driver prints these numbers so that their values will be known by the user installation.

7. Relation to the ANS FORTRAN Standard

As of this writing (May, 1977) the present American National Standard FORTRAN is the 1966 standard, Ref. [6-8], that we will refer to as 1966 FORTRAN. A draft proposed revision to this standard is currently identified as FORTRAN 77, Ref. [9], presently in the final editing phase.

The calling sequences of the BLA subprograms would require that the subprograms contain declarations of the form

```
REAL  SX(MAXO(1,N*IABS(INCX))
```

to precisely specify the array lengths. Neither 1966 FORTRAN nor FORTRAN 77 permits such a statement. A statement of the form

```
REAL  SX(1)
```

is permitted by major FORTRAN compilers to cover cases in which it is inconvenient to specify an exact dimension. This latter form is used in the BLA subprograms even though it does not conform to 1966 FORTRAN. FORTRAN 77

allows the form

```
REAL  SX(*)
```

for this situation. Thus the BLAS can be made to conform to FORTRAN 77 by changing "1's" to "*'s" in the subprogram array declarations.

8. Testing

A Master Test Package has been written in FORTRAN and is included with the submitted code. This package consists of a main program and a set of subprograms containing built-in test data and correct answers. It executes a fixed set of test cases exercising all thirty-eight subprograms or optionally any selected subset of these.

The test driver also calls subroutines SMCHCN and DMCHCN and prints the values of machine dependent values determined by these subroutines.

We have attempted to design the test cases and the Master Test Program to be usable on a wide variety of non-decimal machines having FORTRAN systems.

The Master Test Package has successfully executed, testing the FORTRAN coded version of the Basic Linear Algebra Subprograms, on Univac 1108, IBM 360/67, Burroughs 6700, CDC 6600, and CDC 7600 computers. These tests have also been run successfully testing the respective assembler packages on the Univac 1108, IBM 360/67, CDC 6600 and CDC 7600 computers.

The following method of comparing true and computed numbers is used in the Master Test Package. Let z denote a pre-stored true result and let $\bar{z}$ denote the corresponding computed result to be tested. The numbers σ and ϕ are prestored constants that will be discussed below. The test program computes

$$
\begin{aligned}
d &= fl(\bar{z}-z) \\
g &= fl\left(|\sigma| + |fl(\phi*d)|\right) \\
h &= |\sigma| \\
\tau &= fl(g-h)
\end{aligned}
$$

where fl denotes machine floating point arithmetic of the current working precision, either single precision or double precision. It is further assumed that g and h are truncated to working precision before being used in the computation of τ.

The test is passed if $\tau = 0$ and fails if $\tau \neq 0$. Note that τ will be zero if $|d|$ is so small that adding $|fl(\phi*d)|$ to $|\sigma|$ gives a result that is not distinguished from $|\sigma|$ when truncated to working precision.

For example, suppose $\sigma = 1.$, $\phi = .5$, $d = 10^{-9}$; then the mathematical value of $\sigma + \phi*d$ is 1.0000000005, but the single precision computed value of g on the Univac 1108 will be 1. resulting in $\tau = 0$. Thus in this case d is small enough to pass the test.

The number σ is prestored along with the correct result z in the testing program. In general, σ has different values for different test cases.

The number ϕ is a "tuning" factor which has been determined empirically to make the test perform correctly on a variety of machines. Note that the stringency of the test is relaxed by decreasing the value of ϕ. This has been used to desensitize the testing to the effects of differences in the treatment of trailing digits in the floating point arithmetic of different machines.

There are four different values of ϕ prestored in the main program, TBLA, of the testing package. These values are called SFAC, SDFAC, DFAC, and DQFAC. These are used for testing operations which are respectively single precision, mixed single and double precision, double precision, and mixed double and extended precision.

It is intended that the test package be useful to anyone who undertakes the implementation of an assembly-coded version of this package. In working on a new machine, one may find it necessary to reduce the values of one or more of the numbers SFAC, SDFAC, DFAC, or DQFAC to obtain correct test

performance. The authors would appreciate hearing of any new assembly-coded versions of the packages and of any need to reduce the values of these tuning parameters.

9. Selected Timing Results for the IBM 360/67, CDC 6600 and Univac 1108

Timing of Dot Products and Elementary Vector Operations

The most obvious implementation of the dot product and elementary vector operations for vectors with unit storage increments are in-line FORTRAN loops 1 and 2:

```
      W = 0.
            DO 10 I = 1,N
10          W = W + X(I)* Y(I)
```

In-Line FORTRAN for Dot Products Loop 1

```
            DO 20 I = 1,N
20          Y(I) = A*X(I) + Y(I)
```

In-Line FORTRAN for Elementary Vector Operations Loop 2

The BLAS replacements for these in-line FORTRAN loops, using the same variable names and appropriate type statements, are

```
W = _DOT(N,X,1,Y,1)
```

BLAS Replacement for Loop 1

```
CALL _AXPY(N,A,X,1,Y,1)
```

BLAS Replacement for Loop 2

The "_" in front of the BLA subprogram names is due to the fact that both single and double precision versions are discussed here.

These subprograms, coded in assembly language, were timed and compared

with the time for the in-line loops. As was stated in section 2, one reason for development of the package was to make highly efficient code possible. This goal has been achieved for the CDC 6600 but not for the IBM 360/67. The IBM 360/67 FORTRAN H compiler, operating with Opt = 2, generates nearly perfect object code.

In Tables 2 and 3 are some sample times for the three machines comparing Loops 1 and 2 and their BLAS replacement. Interpretation of Tables 2 and 3, supported more fully in Appendix 2, are as follows:

- Because of linkage overhead, the BLA subprograms for the IBM 360/67 are always less efficient than the in-line loops. For vectors of large enough length the linkage overhead is relatively negligible.

- The dot product and elementary vector operation subprograms for the CDC 6600 are respectively 3.1 and 1.6 times more efficient than in-line code for vectors of large enough length.

- For the CDC 6600, dot products are considerably more efficient than elementary vector operations on vectors of the same length.

Vector Length, N	IBM 360/67 Double Precision		CDC 6600 Single Precision		Univac 1108 Single Precision	
	In-Line FORTRAN (H,Opt=2)	Assembler	In-Line FORTRAN (FTN,Opt=2)	Assembler	In-Line FORTRAN	Assembler
10	0.1438	0.1917	0.0360	0.0480	0.0756	0.0790
25	0.3436	0.3854	0.0750	0.0625	0.1836	0.1730
50	0.6719	0.7186	0.1400	0.0800	0.3598	0.3182
100	1.3750	1.3750	0.2800	0.1250	0.6986	0.6162

Time, in seconds, for 1000 executions of in-line FORTRAN Loop 1 and calls to the _DOT( ) function. Times for 5 runs were averaged.

Apply factors of 1.1 and 0.75 to IBM 360/67 times to get approximate respective times for nonequally spaced increments and single precision. No distinction for nonequal increments is necessary for the CDC 6600 and Univac 1108.

Table 2. _DOT( ) function and in-line Loop 1 timings

Vector Length, N	IBM 360/67 Double Precision		CDC 6600 Single Precision		Univac 1108 Single Precision	
	In-Line FORTRAN (H,Opt=2)	Assembler	In-Line FORTRAN (FTN,Opt=2)	Assembler	In-Line FORTRAN	Assembler
10	0.0590	0.2050	0.0500	0.0650	0.0740	0.0886
25	0.3930	0.4375	0.1125	0.1000	0.1806	0.1890
50	0.7950	0.8400	0.2100	0.1725	0.3544	0.3574
100	1.5500	1.6000	0.4200	0.3000	0.7292	0.7170

Time, in seconds, for 1000 executions of in-line FORTRAN Loop 2 and calls to the _AXPY( ) subprogram. Times for 5 runs were averaged.

Apply factor of 0.75 to get single precision IBM 360/67 times. Only vectors with unit increments were used in this timing.

Table 3. _AXPY( ) subprogram and in-line Loop 2 timings

Timing of Standard and Modified Givens Methods

Gentlemen's modification of the Givens transformation is discussed in the Appendix. This technique eliminates square roots and two of the four multiply operations when forming the product of the resulting matrix by a 2-vector.

The relative efficiency of Gentlemen's modification to the standard Givens transformation was compared. Both techniques were used to triangularize 2N by N matrices $A = \{a_{ij}\}$ where

$$a_{ij} = (i+j-1)^{-1}$$

In Table 4 there are some sample times which resulted from the triangularizations using both methods.

We are primarily interested in algorithm comparison here, so both methods were timed using their assembler versions to apply the matrix products.

A conclusion is that in the context of triangularizing matrices, the modified Givens transformation method is ultimately more efficient in computer time by factors varying between 1.4 and 1.6. This is fully supported in Appendix 2. The comparison is most favorable on the IBM 360/67 in double precision.

N	IBM 360/67 Double Precision		CDC 6600 Single Precision		Univac 1108 Single Precision	
	Standard Givens	Modified Givens	Standard Givens	Modified Givens	Standard Givens	Modified Givens
10	0.0800	0.0650	0.0200	0.0190	0.0335	0.0298
25	0.8789	0.6250	0.1719	0.1445	0.3633	0.3001

Time, in seconds, for the triangularization of 2N by N matrices using standard and modified Givens transformations. Times for 5 runs were averaged.

Table 4. Standard and modified Givens transformation in matrix triangularization

Acknowledgements

We are grateful for the contributions that numerous people have made to this project. The Master Test Package was programmed by Lawson, with a few modifications by Hanson. The FORTRAN versions of the BLA subprograms were written by Lawson, Krogh, Hanson, and J. Dongarra. The assembly-coded versions for the Univac 1108 were programmed by Krogh and S. Singletary Gold. The assembly-coded versions for the IBM 360/67 were programmed by Hanson and K. Haskell. The assembly-coded versions for the CDC 6600 were programmed by Kincaid, J. Sullivan and E. Williams. Four of these routines were recoded by Hanson and C. Moler. Test runs were made on a variety of machines by P. Fox and E. W. McMahon (Honeywell 6000), P. Knowlton (PDP 10), L. Fosdick (CDC 6600), C. Moler (IBM 360/67), K. Fong (CDC 7600), B. Garbow, J. Dongarra (IBM 370/195), W. Brainerd (Burroughs 6700), and others. Helpful suggestions, based on previous similar work of their own, were given by P. S. Jensen and C. Bailey.

J. Dongarra supplied versions of several FORTRAN implementations of

the subprograms. The choice of coding technique used by Dongarra is based on a set of tests that was carried out at over 40 different installations with various machines in operation. The choice of coding technique was made on the basis of superior timing performance at the largest number of these sites, Ref. [10].

Not everyone who contributed significantly to this project is mentioned. One person who spent a great deal of time during the final phase of the project was J. Wisniewski. His valuable help and contributions are much appreciated. The contributions of W. MacGregor and G. Terrell are also acknowledged.

References

1. R. J. Hanson, F. T. Krogh, and C. L. Lawson, "A Proposal for Standard Linear Algebra Subprograms," Jet Propulsion Laboratory, TM 33-660, November 1973, 14 pp.

2. C. L. Lawson, "Standardization of FORTRAN Callable Subprograms for Basic Linear Algebra," Paper presented at Mathematical Software II, Purdue University, May 1974, 8 pp.

3. W. M. Gentleman, "Least Squares Computations by Givens Transformations without Square Roots," J. Inst. Math. Appl., 12, 1973, pp. 329-336.

4. Sven Hammarling, "A Note on Modifications of the Givens Plane Rotation," J. Inst. Math. Appl., 13, 1974, No. 2, pp. 215-218.

5. C. L. Lawson and R. J. Hanson, Solving Least Squares Problems, Prentice-Hall, 1974.

6. American National Standards Institute, "American National Standard FORTRAN," 1966, New York.

7. ASA Committee X3 "FORTRAN vs. Basic FORTRAN," Comm. ACM, 7, No. 10, 1964, 591-625.

8. ANSI Subcommittee X3J3, "Clarification of FORTRAN Standards - Second Report," Comm. ACM, 14, No. 10, 1971, 628-642.

9. ANS Committee X3J3, Document X3J3/76.7 Fortran 77, March 18, 1977.

10. Dongarra, J. J., Fortran BLAS Timing. LINPACK Working Note #3. Argonne Natl. Lab. (Draft of March, 1977)

11. Cody, W. J., Software for the Elementary Functions. Mathematical Software, Edited by J. R. Rice. Academic Press, New York (1971).

12. Blue, J. L., A Portable Fortran Program to Find the Euclidean Norm of a Vector. Trans. Math. Software (to appear).

13. Stewart, G. W., The Economical Storage of Plane Rotations. Numer. Math., Vol. 25, No. 2, p. 137-139 (1976).

14. Brent, R., A Fortran Multiple Precision Arithmetic Package. To appear, TOMS.

Appendix 1

The Modified Givens Transformation

The Givens transformation which eliminates z_1, if $z_1 \neq 0$, is

$$GW = \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{bmatrix} w_1 \cdots w_N \\ z_1 \cdots z_N \end{bmatrix} \tag{A.1}$$

where $c = w_1/r$, $s = z_1/r$, $r = \pm \left(w_1^2+z_1^2\right)^{\frac{1}{2}}$. This requires $\sim 4N$ floating point multiplications, 2N floating point additions and one square root. Gentleman, Ref. [3], has reported on a modification to the Givens transformation which reduces this operation count. Gentleman's idea is presented here in a slightly different form than found in his paper.

Suppose that W in Eq. (A.1) is available in factored form

$$W = D^{\frac{1}{2}}X \equiv \begin{bmatrix} d_1^{\frac{1}{2}} & 0 \\ 0 & d_2^{\frac{1}{2}} \end{bmatrix} \begin{bmatrix} x_1 \cdots x_N \\ y_1 \cdots y_N \end{bmatrix} \tag{A.2}$$

Substituting $D^{\frac{1}{2}}X$ for W and refactoring $GD^{\frac{1}{2}}$ as $\tilde{D}^{\frac{1}{2}}H$ yields

$$(A.3) \qquad GW = GD^{\frac{1}{2}}X = \tilde{D}^{\frac{1}{2}}HX \equiv \begin{bmatrix} \tilde{d}_1^{\frac{1}{2}} & 0 \\ 0 & \tilde{d}_2^{\frac{1}{2}} \end{bmatrix} HX$$

The right-hand side of Eq. (A.3) yields an updated factored form for the matrix product GW. The crucial point is that the matrix H is selected so that two elements are exactly units. This eliminates 2N floating point multiplications when forming the matrix product HX. To preserve numerical stability two cases are considered:

For $|s| < |c|$

$$(A.4) \quad GD^{\frac{1}{2}} = \begin{bmatrix} d_1^{\frac{1}{2}}c & d_2^{\frac{1}{2}}s \\ -d_1^{\frac{1}{2}}s & d_2^{\frac{1}{2}}c \end{bmatrix} = \begin{bmatrix} d_1^{\frac{1}{2}}c & 0 \\ 0 & d_2^{\frac{1}{2}}c \end{bmatrix} \begin{bmatrix} 1 & t(d_2/d_1)^{\frac{1}{2}} \\ -(d_1/d_2)^{\frac{1}{2}}t & 1 \end{bmatrix} = \begin{bmatrix} \tilde{d}_1^{\frac{1}{2}} & 0 \\ 0 & \tilde{d}_2^{\frac{1}{2}} \end{bmatrix} \begin{bmatrix} 1 & d_2y_1/d_1x_1 \\ -y_1/x_1 & 1 \end{bmatrix} \equiv \tilde{D}^{\frac{1}{2}}H$$

where $t = s/c$.

For $|c| \leq |s|$, by similar manipulations,

$$(A.5) \qquad GD^{\frac{1}{2}} = \begin{bmatrix} \tilde{d}_1^{\frac{1}{2}} & 0 \\ 0 & \tilde{d}_2^{\frac{1}{2}} \end{bmatrix} \begin{bmatrix} d_1x_1/d_2y_1 & 1 \\ -1 & x_1/y_1 \end{bmatrix} \equiv \tilde{D}^{\frac{1}{2}}H$$

where $\tilde{d}_1^{\frac{1}{2}} = d_2^{\frac{1}{2}}s$, and $\tilde{d}_2^{\frac{1}{2}} = d_1^{\frac{1}{2}}s$. This factorization can be done for any plane rotation matrix.

Only the squares of the scale factors $d_i^{\frac{1}{2}}$ are involved in the non-unit elements of the matrix H defined in Eq. (A.4) - (A.5), which permits the Givens transformation Eq. (A.1) to be computed without square roots. Using the identity $c^2 = (1+t^2)^{-1}$ and Eq. (A.4) allows the squares of the scale

factors to be updated: $\tilde{d}_i = d_i(1+t^2)^{-1}$, $i = 1,2$. Letting $\tau = s/c$ in Eq. (A.5) we have $\tilde{d}_1 = d_2(1+\tau^2)^{-1}$ and $\tilde{d}_2 = d_1(1+\tau^2)^{-1}$. For $|c| > |s|$ or, equivalently, $|d_1x_1^2| > |d_2y_1^2|$

$$
\begin{aligned}
h_{11} &= 1\ , \quad h_{21} = -y_1/x_1 \\
h_{12} &= d_2y_1/d_1x_1\ , \quad h_{22} = 1 \\
u &= 1 - h_{21}h_{12} \\
d_1 &:= d_1/u \\
d_2 &:= d_2/u \\
x_1 &:= x_1u
\end{aligned}
\tag{A.6}
$$

For $|c| \le |s|$ or, equivalently, $|d_1x_1^2| \le |d_2y_1^2|$

$$
\begin{aligned}
h_{11} &= d_1x_1/d_2y_1\ , \quad h_{21} = 1 \\
h_{12} &= 1\ , \quad h_{22} = x_1/y_1 \\
u &= 1 + h_{11}h_{22} \\
v &= d_1/u \\
d_1 &:= d_2/u \\
d_2 &:= v \\
x_1 &:= y_1u
\end{aligned}
\tag{A.7}
$$

When using the modified Givens transformation in the context of "row accumulation," $d_i > 0$, $i = 1, 2$, the values of u in Eq. (A.6) - (A.7) will satisfy $1 \le u \le 2$. Thus the squares d_i, $i = 1,2$, decrease by as much as 1/2 at each updating step. If no rescaling action is taken, these scale

factors would ultimately underflow. The details concerning rescaling are implemented in the modified Givens subprograms.

Since only d_i, the squares of the weights, appear in the formulas of Eq. (A.6) - (A.7) it is possible to use the same formulas to remove a row from a least squares problem simply by setting $d_2 = -1$. Remarks about this row removal method are found in Ref. [5], Chapter 27.

When the modified Givens transformation is used in the context of the "row removal method" mentioned above, the values of u in Eq. (A.6) - (A.7) satisfy $0 \le u \le 1$. The case $u = 0$ is eliminated by restricting $d_1 \ge 0$. If $d_1 < 0$, we define H as the zero matrix, the updated $d_i = 0$, $i = 1,2$, and $x_1 = 0$. With this restriction, we have $0 < u \le 2$ in Eq. (A.6) - (A.7). Thus the change in the scale factors d_i, $i = 1,2$, is unbounded at each step. Either underflow or overflow can occur if no rescaling is performed.

The problem is rescaled by the modified Givens subprograms to keep within the conservative limits

$$\gamma^{-2} \le |d_i| \le \gamma^2 , \quad i = 1,2, \ \gamma = 4096 .$$

Note that when we rescale $d_i := d_i\gamma^2$, we must rescale $h_{ij} := h_{ij}\gamma^{-1}$, $j = 1,2$, and rescale $x_1 := x_1\gamma^{-1}$.

Appendix 2

Extended Timing Results for Some Operations

In Section 9 selected timing results were presented for the IBM 360/67 (double precision), the CDC 6600 (sirgle precision), and the Univac 1108 (single precision). Timing of dot products, elementary vector operations, and Givens transformations was presented. This was done mainly for the purpose of illustrating the relative efficiency of in-line FORTRAN vs. assembler, and the standard vs. the modified Givens transformation.

Tables 5-11, given below, give more of this data than found in Section 9. The exception to this is the Univac 1108 timing data which is totally presented in Section 9, so we did not reproduce it here.

Vector Length, N	IBM 360/67 Single Precision Equal Storage Increments		IBM 360/67 Single Precision Nonequal Storage Increments	
	In-Line FORTRAN (H,Opt=2)	Assembler	In-Line FORTRAN (H,Opt=2)	Assembler
10	0.1020	0.1470	0.1160	0.1660
25	0.2380	0.2840	0.2740	0.3100
50	0.4620	0.5110	0.5510	0.5720
100	0.9490	0.9970	1.1700	1.1000

Time, in seconds, for 1000 executions of in-line FORTRAN Loop 1, Section 9, and calls to the SDOT( ) function. Times for 5 runs were averaged.

Table 5. IBM 360/67 SDOT( ) function and single precision in-line Loop 1 timings

Vector Length, N	IBM 360/67 Double Precision Equal Storage Increments		IBM 360/67 Double Precision Nonequal Storage Increments	
	In-Line FORTRAN (H,Opt=2)	Assembler	In-Line FORTRAN (H,Opt=2)	Assembler
10	0.1430	0.1910	0.1590	0.1980
25	0.3430	0.3840	0.3840	0.4160
50	0.6770	0.7250	0.7800	0.8180
100	0.3900	1.3900	1.5400	1.5700

Time, in seconds, for 1000 executions of in-line FORTRAN Loop 1, Section 9, and calls to the DDOT( ) function. Times for 5 runs were averaged.

Table 6. IBM 360/67 DDOT( ) function and double precision in-line Loop 1 timings

Vector Length, N	CDC 6600 Single Precision Equal or Nonequal Storage Increments		CDC 7600 Single Precision Equal or Nonequal Storage Increments	
	In-Line FORTRAN (FTN,Opt=2)	Assembler	In-Line FORTRAN (FTN,Opt=2)	Assembler
10	0.0358	0.0480	0.0042	0.0092
25	0.0756	0.0638	0.0100	0.0110
50	0.1420	0.0808	0.0210	0.0162
100	0.2750	0.1230	0.0414	0.0254

Time, in seconds, for 1000 executions of in-line FORTRAN Loop 1, Section 9, and calls to the SDOT( ) function. Times for 5 runs were averaged.

Table 7. CDC 6600 and CDC 7600 SDOT( ) function and single precision in-line Loop 1 timings

Vector Length, N	IBM 360/67 Single Precision Equal Storage Increments		IBM 360/67 Double Precision Equal Storage Increments	
	In-Line FORTRAN (H,Opt=2)	Assembler	In-Line FORTRAN (H,Opt=2)	Assembler
10	0.1190	0.1700	0.1590	0.2040
25	0.2880	0.3610	0.3930	0.4390
50	0.5760	0.6300	0.7960	0.8420
100	1.1700	1.1900	1.5500	1.5900

Time, in seconds, for 1000 executions of in-line FORTRAN Loop 2, Section 9, and calls to the SAXPY( ) and DAXPY( ) subprograms. Times for 5 runs were averaged.

Table 8. IBM 360/67 SAXPY( ) and DAXPY( ) subprogram, and single and double precision in-line Loop 2 timings

Vector Length, N	CDC 6600 Single Precision Equal Storage Increments		CDC 7600 Single Precision Equal Storage Increments	
	In-Line FORTRAN (FTN,Opt=2)	Assembler	In-Line FORTRAN (FTN,Opt=2)	Assembler
10	0.0502	0.0640	0.0060	0.0114
25	0.1120	0.1020	0.0150	0.0162
50	0.2130	0.1710	0.0290	0.0252
100	0.4240	0.3020	0.0582	0.0420

Time, in seconds, for 1000 executions of in-line FORTRAN Loop 2, Section 9, and calls to the SAXPY( ) subprogram. Times for 5 runs were averaged.

Table 9. CDC 6600 and CDC 7600 SAXPY( ) subprogram and single precision in-line Loop 2 timings

N	IBM 360/67 Single Precision		IBM 360/67 Double Precision	
	Standard Givens	Modified Givens	Standard Givens	Modified Givens
10	0.0580	0.0484	0.0800	0.0650
25	0.5850	0.4635	0.8789	0.6250

Time, in seconds, for the triangularization of 2N by N matrices using standard and modified Givens transformations. Times for 5 runs were averaged.

Table 10. IBM 360/67 single and double precision standard and modified Givens transformation timing for matrix triangularization

N	CDC 6600 Single Precision		CDC 7600 Single Precision	
	Standard Givens	Modified Givens	Standard Givens	Modified Givens
10	0.0200	0.0190	0.0036	0.0035
25	0.1719	0.1445	0.0279	0.0250
50	0.9600	0.7550	0.1430	0.1265
100	5.8500	4.3500	0.8200	0.7100

Time, in seconds, for the triangularization of 2N by N matrices using standard and modified Givens transformations. Times for 5 runs were averaged.

Table 11. CDC 6600 and CDC 7600 single precision standard and modified Givens transformation timing for matrix triangularization

Appendix 3

Sample Usage of the BLAS in FORTRAN Programming

Our experience indicates that using the BLAS actually enhances the readability and reliability of codes in which they are utilized. Efficiency does not appreciably degrade with their usage, as indicated in Section 9, and for large-scale problems certain of the BLAS will markedly out-perform in-line FORTRAN code.

These remarks are based on usage of the BLAS in developing new software for the Sandia Math. Library, developing new ordinary differential equation solving codes, conversations with members of the LINPACK working group participating in the project of Ref. [10], and experience with applications programmers at Sandia Laboratories and Jet Propulsion Laboratory.

Typical usage of the BLAS in FORTRAN programs is now illustrated with nine examples using the single precision versions of the operations.

Some rules, based upon the FORTRAN language, that a programmer may find useful to recall are these:

- Suppose a two-dimensional FORTRAN array A(MDA,NDA) is used to hold an M by N matrix $A = \{a_{IJ}\}$. If A(I,J) := a_{IJ}, then the Ith row vector of A and the Jth column vector of A respectively start at A(I,1) and A(1,J). The relations MDA $\geq$ M and NDA $\geq$ N must hold for the matrix to fit into this array.

- The storage increment between elements of row vectors of A, e.g. A(1,1) and A(1,2), is MDA, the first dimensioning parameter of the array A(*,*).

- The storage increment between elements of column vectors of A, e.g. A(1,1) and A(2,1), is 1. This is due to the fact that the FORTRAN language stores A(*,*) by columns:

  A(1,1),A(2,1),...,A(MDA,1),A(1,2),...,A(MDA,2),...,A(MDA,NDA)

  The value of NDA is used by the FORTRAN compiler only to allocate MDA*NDA words of memory in the program.

Example 1

Given M by K and K by N matrices A and B, compute the M by N product matrix C = AB.

The coding technique for this computation is based on the fact that each element c_{IJ} of C is the dot product of row I of A and column J of B.

```
      DIMENSION A(20,20),B(15,10),C(20,15)
C
      MDA=20
      MDB=15
      MDC=20
C
      M=10
      K=15
      N=10
C
C     FORM THE DOT PRODUCT OF ROW I OF A WITH COLUMN J OF B.  EACH OF THESE
C     VECTORS IS OF LENGTH K.  THE VALUE OF MDA IS THE STORAGE INCREMENT
C     BETWEEN ELEMENTS OF ROW VECTORS OF A.
C
                  DO 10 I=1,M
                  DO 10 J=1,N
10                C(I,J)=SDOT(K,A(I,1),MDA,B(1,J),1)
```

Example 2

Solve an N by N upper triangular nonsingular system of algebraic equations, $A\underset{\sim}{x} = \underset{\sim}{b}$. The method used is based on the observation that if we compute the component $x_N = b_N/a_{NN}$, then we have a new problem in N - 1 unknowns, still upper triangular, with the new right-side vector $(b_1-a_{1N}x_N,\ldots,b_{N-1}-a_{N-1,N}x_N)^T$. In this example the solution vector, $\underset{\sim}{x}$, overwrites the vector $\underset{\sim}{b}$ in the array B(*).

```
                  DO 20 II=1,N
                  I=N+1-II
                  B(I)=B(I)/A(I,I)
20                CALL SAXPY (I-1,-B(I),A(1,I),1,B,1)
```

Example 3

Scale the columns (each assumed to be nonzero) of an M by N matrix C so that each column has unit length.

```
                    DO 30 J=1,N
                    T=1.E0/SNRM2(M,C(1,J),1)
30                  CALL SSCAL(M,T,C(1,J),1)
```

Example 4

Row-equilibrate an N by N matrix A. (Divide each non-zero row vector of A by the entry in that row of maximum magnitude). Here MDA is the first dimensioning parameter of the array A(*,*).

```
                    DO 40 I=1,N
                    JMAX=ISAMAX(N,A(I,1),MDA)
                    T=A(I,JMAX)
                    IF(T.EQ.0.E0) GO TO 40
                    CALL SSCAL(N,1.E0/T,A(I,1),MDA)
40                  CONTINUE
```

When using ISAMAX( ) to choose row pivots in Gaussian elimination, for example, the major loop contains a statement of the form

```
IMAX=ISAMAX(N-J+1,A(J,J),1)+J-1
```

At that point IMAX corresponds to the row that will be interchanged with row J. Thus the offset value J - 1 must be added to the computed value of ISAMAX( ) to get the actual row number to interchange.

Example 5

Set an N by N matrix A to the N by N identity matrix. Then set B = A. Notice that a storage increment value of 0 for the first vector

argument of SCOPY( ) is used. This "broadcasts" the values of 0.E0 and 1.E0 into the second vector argument.

Here MDA is the first dimensioning parameter of the array A(*,*).

```
                DO 50 J=1,N
50              CALL SCOPY(N,0.E0,0,A(1,J),1)

                CALL SCOPY(N,1.E0,0,A,MDA+1)

                DO 60 J=1,N
60              CALL SCOPY(N,A(1,J),1,B(1,J),1)
```

Example 6

Interchange or swap the columns of an M by N matrix C. The column to be interchanged with column J is in a type INTEGER array IP(*), and has the value IP(J).

```
                DO 70 J=1,N
                L=IP(J)
                IF(J.NE.L) CALL SSWAP(M,C(1,J),1,C(1,L),1)
70              CONTINUE
```

Example 7

a) Extract the first number and "pop" a list of N single precision numbers: $x_0 := x_1$, $x_i := x_{i+1}$, $i = 1,\ldots,N-1$, $N := N-1$

b) "Push-down" a list of N single precision numbers and insert a new number x_0 at the top of the list: $x_{i+1} := x_i$, $i = N,\ldots,1$; $x_1 := x_0$, $N := N + 1$.

For these illustrations the vector $\underset{\sim}{x} = (x_1,\ldots,x_N)^T$ is in the FORTRAN array X(*).

Notice the usage of the negative increments (-1) for the push-down example of b). This causes the assignment

X(N+1)=X(N),X(N)=X(N-1),...,X(2)=X(1)

to be implemented in this order.

a) Extract and "pop"

```
N=N-1
XO=X(1)
CALL SCOPY(N,X(2),1,X(1),1)
```

b) "Push-down" and insert

```
CALL SCOPY(N,X(1),-1,X(2),-1)
N=N+1
X(1)=XO
```

Example 8

In this example we want to transpose an N by N matrix A in-place, (in-situ). Here MDA is the first dimensioning parameter of the array A(*,*).

```
      DO 80 J=1,N
80    CALL SSWAP(N-J,A(J,J+1),MDA,A(J+1,J),1)
```

Example 9

In this more complicated example we swap in-place (in-situ) the components of the vector

$$(x_1,\ldots,x_K,x_{K+1},\ldots,x_N)^T$$

so they become

$$(x_{K+1},\ldots,x_N,x_1,\ldots,x_K)^T$$

making repeated use of the "Pop" or "Push-down" operations.

```
      NMK=N-K
      IF(.NOT.(K.GT.0.AND.NMK.GT.0)) GO TO 120

         IF(.NOT.(K.LT.NMK)) GO TO 100
            DO 90 I=1,K
            T=X(1)

            CALL SCOPY(N-1,X(2),1,X(1),1)
90          X(N)=T
         GO TO 120
```

```
100          CONTINUE
                DO 110 I=1,NMK
                T=X(N)
                CALL SCOPY(N-1,X(1),-1,X(2),-1)
110             X(1)=T

120       CONTINUE
```