Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard

Basic Linear Algebra Subprograms Technical (BLAST) Forum

August 21, 2001

Contents

Acknowledgments

The technical development was carried out by subgroups, whose work was reviewed by the full committee. During the period of development of the Basic Linear Algebra Subprograms (BLAS) Technical Forum Standard, many people served in positions of responsibility and are listed below.

- Jack Dongarra and Sven Hammarling, Conveners and Meeting Chairs
- Susan Blackford and Andrew Lumsdaine, Minutes
- Susan Blackford, Editor

The primary chapter authors are the following:

- Susan Blackford, Jack Dongarra, and Sven Hammarling, Chapter 1
- Susan Blackford, Jack Dongarra, and Sven Hammarling, Linda Kaufman, Zohair Maany, Antoine Petitet, Chapter 2
- Iain Duff, Mike Heroux, Roldan Pozo, Karin Remington, Chapter 3
- Jim Demmel, Greg Henry, Velvel Kahan, Xiaoye Li, Chapter 4
- Clint Whaley, C Interface to the Legacy BLAS
- Jack Dongarra, Fred Krogh, Journal of Development – Environmental routines
- Antoine Petitet, Journal of Development – Distributed-Memory Dense BLAS
- Sven Hammarling, Zohair Maany, Journal of Development – Fortran95 Thin BLAS
- George Corliss, Chenyi Hu, Baker Kearfoot, Bill Walster, J. Wolff v. Gudenberg, Journal of Development – Interval BLAS

We would like to thank the individuals from the following organizations who have written the reference implementations: University of California, Berkeley, University of Houston, Downtown, University of Notre Dame, University of Tennessee, HP/Convex, NAG, NIST, and CERFACS.

Specifically, we thank the following students at the University of California, Berkeley, for their work on the reference implementations and proofreading of various versions of the document: Ben Wanzo, Berkat Tung, Weihua Shen, Anil Kapur, Michael Martin, Jimmy Iskandar, Yozo Hida, Teresa Tung, Yulin Li.

We would like to thank the following vendors and ISPs: Cray, Digital/Compaq, HP/Convex, IBM, Intel, NEC, SGI, Tera, NAG, and VNI.

We thank Paul McMahan of the University of Tennessee for preparing the commenting and voting pages on the BLAS Technical Forum webpage.

We would like to thank the members of the global community who have posted comments, suggestions, and proposals to the email reflector and the BLAS Technical Forum webpage.

And lastly, we would like to thank the attendees of the BLAS Technical Forum meetings:

Andy Anda, Ed Anderson, Zhaojun Bai, David Bailey, Satish Balay, Puri Bangalore, Claus Bendtsen, Jesse Bennett, Mike Berry, Jeff Bilmes, Susan Blackford, Phil Bording, Clay Breshears, Sandra Carney, Mimi Celis, Andrew Chapman, Samar Choudhary, Edmond Chow, Almadena Chtchelkanova, Andrew Cleary, Isom Crawford, Michel Daydé, John Dempsey, Theresa Do, Dave Dodson, Jack Dongarra, Craig Douglas, Paul Dressel, Jeremy Du Croz, Iain Duff, Carter Edwards, Salvatore Filippone, Rob Gjertsen, Roger Golliver, Cormac Garvey, Ian Gladwell, Bruce Greer, Bill Gropp, John Gunnels, Fred Gustavson, Sven Hammarling, Richard Hanson, Hidehiko Hasegawa, Satomi Hasegawa, Greg Henry, Mike Heroux, Jeff Horner, Gary Howell, Mary Beth Hribar, Chenyi Hu, Steve Huss-Lederman, Melody Ivory, Naoki Iwata, Bo Kågström, Velvel Kahan, Chandrika Kamath, Linda Kaufman, David Kincaid, Jim Koehler, Vipin Kumar, Rich Lee, Steve Lee, Guangye Li, Jin Li, Sherry Li, Hsin-Ying Lin, John Liu, Andew Lumsdaine, Dave Mackay, Kristin Marshoffe, Kristi Maschhoff, Brian McCandless, Joan McComb, Noel Nachtigal, Jim Nagy, Esmond Ng, Tom Oppe, Antoine Petitet, Roldan Pozo, Avi Purkayastha, Padma Raghavan, Karin Remington, Yousef Saad, Majed Sidani, Jeremy Siek, Tony Skjellum, Barry Smith, Ken Stanley, Pete Stewart, Shane Story, Chuck Swanson, Françoise Tisseur, Anne Trefethen, Anna Tsao, Robert van de Geijn, Phuong Vu, Kevin Wadleigh, David Walker, Bob Ward, Jerzy Waśniewski, Clint Whaley, Yuan-Jye Jason Wu, Chao Yang, and Guodong Zhang.

Suggestions for Reading

This document is divided into chapters, appendices, a journal of development, and an index of routine names. It is large, and it is not necessary for a user to read it in its entirety. A user may choose to not read certain chapters or sections within this document, depending upon his/her areas of interest. **Chapters 2–4** contain a functionality discussion and language bindings for dense and band, sparse, and mixed and extended precision BLAS, respectively. Thus, these chapters may be read independently, referring to **Chapter 1** and the **Appendix** for notation and implementation details common to all chapters. Refer to section 1.3 for a more detailed description of the organization of this document.

Chapter 1

Introduction

1.1 Introduction

This document defines the BLAS Technical Forum standard, a specification of a set of kernel routines for linear algebra, historically called the Basic Linear Algebra Subprograms and commonly known as the BLAS. In addition to this publication, the complete standard can be found on the BLAS Technical Forum webpage (`http://www.netlib.org/blas/blast-forum/`).

Numerical linear algebra, particularly the solution of linear systems of equations, linear least squares problems, eigenvalue problems and singular value problems, is fundamental to most calculations in scientific computing, and is often the computationally intense part of such calculations. Designers of computer programs involving linear algebraic operations have frequently chosen to implement certain low level operations, such as the dot product or the matrix vector product, as separate subprograms. This may be observed both in many published codes and in codes written for specific applications at many computer installations.

This approach encourages structured programming and improves the self-documenting quality of the software by specifying basic building blocks and identifying these operations with unique mnemonic names. Since a significant amount of execution time in complicated linear algebraic programs may be spent in a few low level operations, reducing the execution time spent in these operations leads to an overall reduction in the execution time of the program. The programming of some of these low level operations involves algorithmic and implementation subtleties that need care, and can be easily overlooked. If there is general agreement on standard names and parameter lists for some of these basic operations, then portability and efficiency can also be achieved.

The first major concerted effort to achieve agreement on the specification of a set of linear algebra kernels resulted in the Level 1 Basic Linear Algebra Subprograms (BLAS)[1] [39] and associated test suite. The Level 1 BLAS are the specification and implementation in Fortran of subprograms for scalar and vector operations. This was the result of a collaborative project in 1973-77. Following the distribution of the initial version of the specifications to people active in the development of numerical linear algebra software, a series of open meetings were held at conferences and, as a result, extensive modifications were made in an effort to improve the design and make the subprograms more robust. The Level 1 BLAS were extensively and successfully exploited by LINPACK [23], a software package for the solution of dense and banded linear equations and linear least squares problems.

With the advent of vector machines, hierarchical memory machines and shared memory parallel machines, specifications for the Level 2 and 3 BLAS [26, 25], concerned with matrix-vector and

[1]Originally known just as the BLAS, but in the light of subsequent developments now known as the Level 1 BLAS

matrix-matrix operations respectively, were drawn up in 1984-86 and 1987-88. These specifications made it possible to construct new software to utilize the memory hierarchy of modern computers more effectively. In particular, the Level 3 BLAS allowed the construction of software based upon block-partitioned algorithms, typified by the linear algebra software package LAPACK [6]. LAPACK is state-of-the-art software for the solution of dense and banded linear equations, linear least squares, eigenvalue and singular value problems, makes extensive use of all levels of BLAS and particularly utilizes the Level 2 and 3 BLAS for portable performance. LAPACK is widely used in application software and is supported by a number of hardware and software vendors.

To a great extent, the user community embraced the BLAS, not only for performance reasons, but also because developing software around a core of common routines like the BLAS is good software engineering practice. Highly efficient machine-specific implementations of the BLAS are available for most modern high-performance computers. The BLAS have enabled software to achieve high performance with portable code.

The original BLAS concentrated on dense and banded operations, but many applications require the solution of problems involving sparse matrices, and there have also been efforts to specify computational kernels for sparse vector and matrix operations [22, 27].

In the spirit of the earlier BLAS meetings and the standardization efforts of the MPI and HPF forums, a technical forum was established to consider expanding the BLAS in the light of modern software, language, and hardware developments. The BLAS Technical Forum meetings began with a workshop in November 1995 at the University of Tennessee. Meetings were hosted by universities, government institutions, and software and hardware vendors. Detailed minutes were taken for each of the meetings, and these minutes are available on the BLAS Technical Forum webpage (`http://www.netlib.org/blas/blast-forum/`).

Various working groups within the Technical Forum were established to consider issues such as the overall functionality, language interfaces, sparse BLAS, distributed-memory dense BLAS, extended and mixed precision BLAS, interval BLAS, and extensions to the existing BLAS. The rules of the forum were adopted from those used for the MPI and HPF forums. In other words, final acceptance of each of the chapters in the BLAS Technical Forum standard were decided at the meetings using *Robert's Rules*. Drafts of the document were also available on the BLAS Technical Forum webpage, and attendees were permitted to edit chapters, give comments, and vote on-line in "virtual meetings", as well as to conduct discussions on the email reflector. The efforts of these working groups are summarized in this document. Most of these discussions resulted in definitive proposals which led to the specifications given in Chapters 2 - 4. Not all of the discussions resulted in definitive proposals, and such discussions are summarized in the Journal of Development in the hope that they may encourage future efforts to take those discussions to a successful conclusion.

A major aim of the standards defined in this document is to enable linear algebra libraries (both public domain and commercial) to interoperate efficiently, reliably and easily. We believe that hardware and software vendors, higher level library writers and application programmers all benefit from the efforts of this forum and are the intended end users of these standards.

The specification of the original BLAS was given in the form of Fortran 66 and subsequently Fortran 77 subprograms. In this document we provide specifications for Fortran 95[2], Fortran 77 and C. Reference implementations of the standard are provided on the BLAS Technical Forum webpage (`http://www.netlib.org/blas/blast-forum/`). Alternative language bindings for C++ and Java were also discussed during the meetings of the forum, but the specifications for these bindings were postponed for a future series of meetings.

The remainder of this chapter is organized as follows. Section 1.2 provides motivation for the

[2]the current Fortran standard

functionality. Section 1.3 outlines the organization of the document, and section 1.4 summarizes the nomenclature and conventions used in the document. Section 1.5 presents tables of functionality for the routines, and section 1.6 discusses issues concerning the numerical accuracy of the BLAS. Section 1.7 briefly describes the presentation of the specifications for the routines, and section 1.8 details the error handling mechanisms utilized within the routines.

1.2 Motivation

The motivation for the kernel operations is proven functionality. Many of the new operations are based upon auxiliary routines in LAPACK [6] (e.g., SUMSQ, GEN_GROT, GEN_HOUSE, SORT, GE_NORM, GE_COPY). Only after the LAPACK project was begun was it realized that there were operations like the matrix copy routine (GE_COPY), the computation of a norm of a matrix (GE_NORM) and the generation of Householder transformations (GEN_HOUSE) that occurred so often that it was wise to make separate routines for them.

A second group of these operations extended the functionality of some of the existing BLAS (e.g., AXPBY, WAXPBY, GER, SYR/HER, SPR/HPR, SYR2/HER2, SPR2/HPR2). For example, the Level 3 BLAS for the rank k update of a symmetric matrix only allows a positive update, which means that it cannot be used for the reduction of a symmetric matrix to tridiagonal form (to facilitate the computation of the eigensystem of a symmetric matrix), or for the factorization of a symmetric indefinite matrix, or for a quasi-Newton update in an optimization routine.

Other extensions (e.g., AXPY_DOT, GE_SUM_MV, GEMVT, TRMVT, GEMVER) perform two Level 1 BLAS (or Level 2 BLAS) routine calls simultaneously to increase performance by reducing memory traffic.

One important feature of the new standard is the inclusion of sparse matrix computational routines. Because there are many formats commonly used to represent sparse matrices, the Level 2 and Level 3 Sparse BLAS routines utilize an abstract representation, or handle, rather than a fixed storage description (e.g. compressed row, or skyline storage). This handle-based representation allows one to write portable numerical algorithms using the Sparse BLAS, independent of the matrix storage implementation, and gives BLAS library developers the best opportunity for optimizing and fine-tuning their kernels for specific architectures or application domains.

The original Level 2 BLAS included, as an appendix, the specification of extended precision subprograms. With the widespread adoption of hardware supporting the IEEE extended arithmetic format [37], as well as other forms of extended precision arithmetic, together with the increased understanding of algorithms to successfully exploit such arithmetic, it was felt to be timely to include a complete specification for a set of extra precise BLAS.

1.3 Organization of the Document

This document is divided into chapters, appendices, a journal of development, and an index. It is large, and it is not necessary for a user to read it in its entirety. A user may choose to not read certain chapters or sections within this document, depending upon his/her areas of interest. **Chapters 2–4** contain a functionality discussion and language bindings for dense and band, sparse, and mixed and extended precision BLAS, respectively. The **Journal of Development** presents areas of research that are not yet mature enough to be considered as chapters, but were nevertheless discussed at the meetings of the forum. A **Bibliography** is also provided, as well as an **Index** of routine names.

All users are encouraged to frequently refer to the list of notation denoted in sections 1.4, 2.3, and 3.4.

- **Chapter 1: Introduction** provides a brief overview of the background, motivation, and history of the BLAS Technical Forum effort. It also outlines the structure of the document, conventions in notation, and overall functionality contained in the chapters.
- **Chapter 2: Dense and Banded BLAS** presents the functionality and language bindings for proposed "new" dense and banded BLAS routines for serial and shared memory computing.
- **Chapter 3: Sparse BLAS** presents the functionality and language bindings for proposed "new" sparse BLAS routines for serial and shared memory computing.
- **Chapter 4: Extended and Mixed Precision BLAS** presents the functionality and language bindings for proposed extended- precision and mixed-precision BLAS routines for serial and shared memory computing.
- **Appendix** contains pertinent definitions and implementation details for the chapters.
- **Legacy BLAS** contains alternative language bindings for the legacy Level 1, 2, and 3 BLAS for dense and band matrix computations.
- **Journal of Development** contains separate proposals for environmental enquiry routines, Distributed-memory dense BLAS, Fortran 95 Thin BLAS, and Interval BLAS.

1.4 Nomenclature and Conventions

This section addresses mathematical notation and definitions, as well as the numerical accuracy for the BLAS routines. Language-independent issues are also presented.

1.4.1 Notation

The following notation is used throughout the document.

- A, B, C – matrices
- D, D_L, D_R – diagonal matrices
- H – Householder matrix
- J – symmetric tridiagonal matrix (including 2×2 blocked diagonal)
- P – permutation matrix
- T – triangular matrix
- $op(A)$ – denotes A, or A^T or A^H where A is a matrix.
- transpose – denotes A^T where A is a matrix.
- conjugate-transpose – denotes A^H where A is a complex Hermitian matrix.

- u, v, w, x, y, z – vectors
- $\bar{x}$ – specifies the conjugate of the complex vector x
- $incu$, $incv$, $incw$, $incx$, $incy$, $incz$ – stride between successive elements of the respective vector
- Greek letters - scalars (but not exclusively Greek letters)
- x_i - an element of a one-dimensional array
- $y|_x$ – refers to the elements of y that have common indices with the sparse vector x.
- ϵ - machine epsilon
- $\leftarrow$ – assignment statement
- $\leftrightarrow$ – swap (assignment) statement
- $||\cdot||_p$ – the p-norm of a vector or matrix

Additional notation for sparse matrices can be found in 3.4.

For the mathematical formulation of the operations, as well as their algorithmic presentation, we have chosen to index the vector and matrix operands starting from zero. This decision was taken to simplify the presentation of the document but has no impact on the convention a particular language binding may choose.

1.4.2 Operator Arguments

Some BLAS routines take input-only arguments that are called "operator" arguments. These arguments allow for the specification of multiple related operations to be performed by a single function.

The operator arguments used in this document are norm, sort, side, uplo, trans, conj, diag, jrot, order, index_base, and prec. Their possible meanings are defined as follows:

norm: this argument is used by the routines computing the norm of a vector or matrix. Eight possible distinct values are valid that specify the norm to be computed, namely the one-norm, real one-norm, infinity-norm and real infinity norms for vectors and matrices, the 2-norm for vectors, and the Frobenius-norm, max-norm and real max-norm for matrices.

sort: this argument is used by the sorting routines. Two possible distinct values are valid that specify whether the data should be sorted in increasing or decreasing order.

side: this argument is used only by functions computing the product of two matrices A and B. Two possible distinct values are valid, that specify whether $A \cdot B$ or $B \cdot A$ should be computed.

uplo: this argument refers to triangular and symmetric (Hermitian) matrices. Two possible distinct values are valid distinguishing whether the matrix, or its storage representation, is upper or lower triangular.

trans: this argument is used by the routines applying a matrix, say A, to another vector or another matrix. Three possible distinct values are valid that specify whether the matrix A, its transpose A^T or its conjugate transpose A^H should be applied. We use the notation $op(A)$ to refer to A, A^T or A^H depending on the input value of the trans operator argument.

conj: this argument is used by the complex routines operating with $\bar{x}$ or x.

diag: this argument refers exclusively to triangular matrices. Two possible distinct values are valid distinguishing whether the triangular matrix has unit-diagonal or not.

jrot: this argument is used by the routine to generate Jacobi rotations. Three possible distinct values are valid and specify whether the rotation is an inner rotation, an outer rotation, or a sorted rotation.

order: this argument is used by the C bindings to specify if elements within a row of an array are contiguous, or if elements within a column of an array are contiguous (see section 2.6.6).

index_base: this argument is used by Chapter 3 to specify either one-based or zero-based indexing (see section 3.4.1).

prec: this argument is used in Chapter 4 and specifies the internal precision to be used by an extended precision routine. Four distinct values are valid and specify whether the internal precision is single precision, double precision, indigenous, or extra. Details on these settings can be found in section 4.3.1.

All possible meanings for each operator are listed in section A.3. Their representation is defined in the interface issues for the specific programming language: sections 2.4, 3.6.1, and 4.4.1 for Fortran 95; sections 2.5, 3.6.2, and 4.4.2 for Fortran 77; and sections 2.6, 3.6.3, and 4.4.3 for C. The values of the Fortran 95 derived types (for Chapters 2 and 4) are defined in the Fortran 95 module `blas_operator_arguments`, and the values of the Fortran 95 named constants (for Chapter 3) are defined in `blas_sparse_namedconstants`, see section A.4. Similarly, the values of the Fortran 77 named constants are defined in the Fortran 77 include file `blas_namedconstants.h`, in section A.5. And finally, the values of the C enumerated types are defined in the C include file `blas_enum.h`, in section A.6.

Rationale. The intent is to provide each language binding with the opportunity to choose the most appropriate form these arguments should take. For example, in Fortran 95, derived types with named constants have been selected for Chapters 2 and 4, whereas derived types could not be used in Chapter 3 (see section 3.6.1 for details). In Fortran 77, integers with named constants have been chosen. And finally, in C, operator arguments are represented by enumerated types. (*End of rationale.*)

1.4.3 Scalar Arguments

Many scalar arguments are used in the specifications of the BLAS routines. For example, the size of a vector or matrix operand is determined by the integer argument(s) m and/or n. Note that it is permissible to call the routines with m or n equal to zero, in which case the routine exits immediately without referencing its vector/matrix elements. Some routines return a displacement denoted by the integer argument k. The scaling of a vector or matrix is often denoted by the arguments alpha and beta.

The following symbols are used: a, b, c, d, r, s, t, alpha, beta and tau.

1.4.4 Vector Operands

A n-length vector operand x is specified by two arguments – x and incx. x is an array that contains the entries of the n-length vector x. incx is the stride within x between two successive elements of the vector x.

The following lowercase letters are used to denote a vector: u, v, w, x, y, and z. The corresponding strides are respectively denoted incu, incv, incw, incx, incy, and incz.

Advice to implementors. The increment arguments incu, incv, incw, incx, incy and incz may not be zero. (*End of advice to implementors.*)

Example: The mathematical function returning the inner-product r of two real n-length vectors x and y can be defined by:

$$r = x^T y = \sum_{i=0}^{n-1} x_i y_i.$$

Rationale. The arguments incx, and incy do not play a role in the mathematical formulation of the operation. These arguments allow for the specification of subvector operands in various language bindings. Therefore, some of these arguments may not be present in all language-dependent specifications. (*End of rationale.*)

1.4.5 Matrix Operands

A m-by-n matrix operand A is specified by the argument A. A is a language-dependent data structure containing the entries of the matrix operand A. The representation of the matrix entry $a_{i,j}$ in A is denoted by A(i,j) for all (i,j) in the interval $[0 \ldots m-1] \times [0 \ldots n-1]$.

Capital letters are used to denote a matrix. The functions involving matrices use only four symbols, namely A, B, C, and T.

1.4.6 Naming Conventions

Language bindings are specified for Fortran 95, Fortran 77, and C.

The Fortran 95 language bindings have routine names of the form <**name**>, where <**name**> is in lowercase letters and indicates the computation performed. These bindings use generic interfaces to manipulate the data type of the routine, and thus their names do not contain a letter to denote the data type.

The Fortran 77 and C language bindings have routine names of the form **BLAS_x**<**name**>, where the letter **x**, indicates the data type as follows:

Data type	x	Fortran 77	x	C
s.p. real	S	REAL	s	float
d.p. real	D	DOUBLE PRECISION	d	double
s.p. complex	C	COMPLEX	c	float
d.p.complex	Z	COMPLEX*16 or DOUBLE COMPLEX	z	double

The suffix <**name**> in the routine name indicates the computation performed. In the matrix-vector and matrix-matrix routines of Chapters 2 and 4 (and Appendix C.4), the type of the matrix (or of the most significant matrix) is also specified as part of this <**name**> name of the routine. Most of these matrix types apply to both real and complex matrices; a few apply specifically to one or the other, as indicated below. Note that for Appendix C.4, these matrix types apply to interval matrices.

GB	general band
GE	general (i.e., unsymmetric, in some cases rectangular)
HB	(complex) Hermitian band
HE	(complex) Hermitian
HP	(complex) Hermitian, packed storage
SB	(real) symmetric band
SP	symmetric, packed storage
SY	symmetric
TB	triangular band
TP	triangular, packed storage
TR	triangular (or in some cases quasi-triangular)
US	unstructured sparse

For Fortran 77, routine names are in uppercase letters; however, for the C interfaces all routine names are in lowercase letters. To avoid possible name collisions, programmers are strongly advised not to declare variables or functions with names beginning with these prefixes.

A detailed discussion of the format of the <**name**> naming convention is contained in each respective chapter of the document.

1.5 Overall Functionality

This section summarizes, in tabular form, the functionality of the proposed routines. Issues such as storage formats or data types are not addressed. The functionality of the existing Level 1, 2 and 3 BLAS [39, 22, 26, 25] is a subset of the functionality proposed in this document.

In the original BLAS, each level was categorized by the type of operation; Level 1 addressed scalar and vector operations, Level 2 addressed matrix-vector operations, while Level 3 addressed matrix-matrix operations. The functionality tables in this document are categorized in a similar manner, with additional categories to cover operations which were not addressed in the original BLAS.

Unless otherwise specified, the operations apply to both real and complex arguments. For the sake of compactness the complex operators are omitted, so that whenever a transpose operation is given the conjugate transpose should also be assumed for the complex case.

The last column of each table denotes in which chapter of this document the functionality occurs. Specifically,

- "D" denotes dense and banded BLAS (Chapter 2),
- "S" denotes sparse BLAS (Chapter 3), and
- "E" denotes extended and mixed precision BLAS (Chapter 4).

1.5.1 Scalar and Vector Operations

This section lists scalar and vector operations. The functionality tables are organized as follows. Table 1.1 lists the scalar and vector reduction operations, Table 1.2 lists the vector rotation operations, Table 1.3 lists the vector operations, and Table 1.4 lists those vector operations that involve only data movement.

For the Sparse BLAS, x is a compressed sparse vector and y is a dense vector. Details of data structures are in Section 3.4.1.

For further details of vector norm notation, refer to section 2.1.1.

Dot product	$r \leftarrow \beta r + \alpha x^T y$	D,E
	$r \leftarrow x^T y$	S
Vector norms	$r \leftarrow \|\|x\|\|_1,$	D
	$r \leftarrow \|\|x\|\|_{1R},$	D
	$r \leftarrow \|\|x\|\|_2,$	D
	$r \leftarrow \|\|x\|\|_\infty,$	D
	$r \leftarrow \|\|x\|\|_{\infty R},$	D
Sum	$r \leftarrow \sum_i x_i$	D,E
Min value & location	$k, x_k, ; k = \arg\min_i x_i$	D
Min abs value & location	$k, x_k, k = \arg\min_i(\|Re(x_i)\| + \|Im(x_i)\|)$	D
Max value & location	$k, x_k, ; k = \arg\max_i x_i$	D
Max abs value & location	$k, x_k, k = \arg\max_i(\|Re(x_i)\| + \|Im(x_i)\|)$	D
Sum of squares	$(scl, ssq) \leftarrow \sum x_i^2,$	D
	$ssq \cdot scl^2 = \sum x_i^2$	D

Table 1.1: Reduction Operations

Generate Givens rotation	$(c, s, r) \leftarrow \text{rot}(a, b)$	D
Generate Jacobi rotation	$(a, b, c, s) \leftarrow \text{jrot}(x, y, z)$	D
Generate Householder transform	$(\alpha, x, \tau) \leftarrow \text{house}(\alpha, x),$ $H = I - \alpha u u^T$	D

Table 1.2: Generate Transformations

Reciprocal Scale	$x \leftarrow x/\alpha$	D
Scaled vector accumulation	$y \leftarrow \alpha x + \beta y,$	D,E
	$y \leftarrow \alpha x + y$	S
Scaled vector addition	$w \leftarrow \alpha x + \beta y$	D,E
Combined axpy & dot product	$\begin{cases} \hat{w} \leftarrow w - \alpha v \\ r \leftarrow \hat{w}^T u \end{cases}$	D
Apply plane rotation	$(\ x \quad y\ ) \leftarrow (\ x \quad y\ )R$	D

Table 1.3: Vector Operations

Copy	$y \leftarrow x$	D
Swap	$y \leftrightarrow x$	D
Sort vector	$x \leftarrow \text{sort}(x)$	D
Sort vector & return index vector	$(p, x) \leftarrow \text{sort}(x)$	D
Permute vector	$x \leftarrow Px$	D
Sparse gather	$x \leftarrow y\|_x$	S
Sparse gather and zero	$x \leftarrow y\|_x;\ y\|_x \leftarrow 0$	S
Sparse scatter	$y\|_x \leftarrow x$	S

Table 1.4: Data Movement with Vectors

1.5.2 Matrix-Vector Operations

This section lists matrix-vector operations in table 1.5. The matrix arguments A, B and T are dense or banded or sparse. In addition, where appropriate, the matrix A can be symmetric (Hermitian) or triangular or general. The matrix T represents an upper or lower triangular matrix, which can be unit or non-unit triangular. For the Sparse BLAS, the matrix A is sparse, the matrix T is sparse triangular, and the vectors x and y are dense.

Details of the data structures are discussed in sections 2.2, and 3.4.1.

Matrix-vector product	$y \leftarrow \alpha Ax + \beta y,\ y \leftarrow \alpha A^T x + \beta y$	D,S,E
	$x \leftarrow \alpha Tx,\ x \leftarrow \alpha T^T x$	D,E
	$y \leftarrow \alpha Ax + y,\ y \leftarrow \alpha A^T x + y$	S
Summed matrix-vector multiplies	$y \leftarrow \alpha Ax + \beta Bx$	D,E
Multiple matrix-vector multiplies	$\begin{cases} x \leftarrow T^T y \\ w \leftarrow Tz \end{cases}$	D
	$\begin{cases} x \leftarrow \beta A^T y + z \\ w \leftarrow \alpha Ax \end{cases}$	D
Multiple matrix-vector mults and low rank updates	$\begin{cases} \hat{A} \leftarrow A + u_1 v_1^T + u_2 v_2^T \\ x \leftarrow \beta \hat{A}^T y + z \\ w \leftarrow \alpha \hat{A} x \end{cases}$	D
Triangular solve	$x \leftarrow \alpha T^{-1} x,\ x \leftarrow \alpha T^{-T} x$	D,S,E
Rank one updates	$A \leftarrow \alpha x y^T + \beta A$	D
and symmetric ($A = A^T$)	$A \leftarrow \alpha x x^T + \beta A$	D
rank one & two updates	$A \leftarrow (\alpha x) y^T + y(\alpha x)^T + \beta A$	D

Table 1.5: Matrix-Vector Operations

1.5.3 Matrix Operations

This section lists a variety of matrix operations. The functionality tables are organized as follows. Table 1.6 lists single matrix operations and matrix operations that involve $O(n^2)$ operations, Table 1.7 lists the $O(n^3)$ matrix-matrix operations and Table 1.8 lists those matrix operations that involve only data movement. Where appropriate one or more of the matrices can also be symmetric (Hermitian) or triangular or general. The matrix T represents an upper or lower triangular matrix, which can be unit or non-unit triangular. D, D_L, and D_R represent diagonal matrices, and J represents a symmetric tridiagonal matrix (including 2×2 block diagonal).

Details of the data structures are discussed in sections 2.2, and 3.4.1.

For further details of matrix norm notation, refer to section 2.1.3.

Matrix norms	$r \leftarrow \|\|A\|\|_1, r \leftarrow \|\|A\|\|_{1R}$	D
	$r \leftarrow \|\|A\|\|_F, r \leftarrow \|\|A\|\|_\infty, r \leftarrow \|\|A\|\|_{\infty R}$	D
	$r \leftarrow \|\|A\|\|_{max}, r \leftarrow \|\|A\|\|_{maxR}$	D
Diagonal scaling	$A \leftarrow DA,\ A \leftarrow AD,\ A \leftarrow D_L A D_R$	D
	$A \leftarrow DAD$	D
	$A \leftarrow A + BD$	D
Matrix acc and scale	$C \leftarrow \alpha A + \beta B$	D
Matrix add and scale	$B \leftarrow \alpha A + \beta B,\ B \leftarrow \alpha A^T + \beta B$	D

Table 1.6: Matrix Operations – $O(n^2)$ floating point operations

Matrix-matrix product	$C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha A^T B + \beta C$	D,E
	$C \leftarrow \alpha AB^T + \beta C,\ C \leftarrow \alpha A^T B^T + \beta C$	D,E
	$C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha A^T B + \beta C$	S
Triangular multiply	$B \leftarrow \alpha TB,\ B \leftarrow \alpha BT$	D,E
	$B \leftarrow \alpha T^T B,\ B \leftarrow \alpha BT^T$	D,E
Triangular solve	$B \leftarrow \alpha T^{-1} B,\ B \leftarrow \alpha T^{-T} B$	D,S,E
	$B \leftarrow \alpha BT^{-1},\ B \leftarrow \alpha BT^{-T}$	D,E
Symmetric rank k & $2k$ updates ($C = C^T$)	$C \leftarrow \alpha AA^T + \beta C,\ C \leftarrow \alpha A^T A + \beta C$	D,E
	$C \leftarrow \alpha AJA^T + \beta C,\ C \leftarrow \alpha A^T JA + \beta C$	D
	$C \leftarrow (\alpha A)B^T + B(\alpha A)^T + \beta C,$ $C \leftarrow (\alpha A)^T B + B^T(\alpha A) + \beta C$	D,E
	$C \leftarrow (\alpha AJ)B^T + B(\alpha AJ)^T + \beta C,$ $C \leftarrow (\alpha AJ)^T B + B^T(\alpha AJ) + \beta C$	D

Table 1.7: Matrix-Matrix Operations - $O(n^3)$ floating point operations

Matrix copy	$B \leftarrow A,\ B \leftarrow A^T$	D
Matrix transpose	$A \leftarrow A^T$	D
Permute Matrix	$A \leftarrow PA,\ A \leftarrow AP$	D

Table 1.8: Data Movement with Matrices

1.6 Numerical Accuracy and Environmental Enquiry

To understand the numerical behavior of the routines proposed here, certain floating point parameters are necessary. Detailed error bounds and limitations due to overflow and underflow are discussed in individual chapters (see sections 2.7, 3.7, 4.3.3, and C.4.4) but all of them depend on details of how floating point numbers are represented. These details are available by calling an environmental enquiry function called FPINFO.

Floating point numbers are represented in scientific notation as follows. This discussion follows the IEEE Floating Point Arithmetic Standard 754 [7].[3]

$$x = \pm d.d \cdots d * BASE^E$$

where $d.d \cdots d$ is a number represented as a string of T significant digits in base BASE with the "point" to the right of the leftmost digit, and E is an integer exponent. E ranges from EMIN up to EMAX. This means that the largest representable number, which is also called the *overflow threshold* or OV, is just less than $BASE^{EMAX+1}$, This also means that the smallest positive "normalized" representable number (i.e. where the leading digit of $d.d \cdots d$ is nonzero) is $BASE^{EMIN}$, which is also called the *underflow threshold* or UN.

When overflow occurs (because a computed quantity exceeds OV in absolute value), the result is typically $\pm\infty$, or perhaps an error message. When underflow occurs (because a computed quantity is less than UN in absolute magnitude) the returned result may be either 0 or a tiny number less than UN in magnitude, with minimal exponent EMIN but with a leading zero $(0.d \cdots d)$. Such tiny numbers are often called *denormalized* or *subnormal*, and floating point arithmetic which returns them instead of 0 is said to support *gradual underflow.*

The *relative machine precision* (or *machine epsilon*) of a basic operation $\odot \in \{+,-,*,/\}$ is defined as the smallest $EPS > 0$ satisfying

$$fl(a \odot b) = (a \odot b) * (1 + \delta) \text{ for some } |\delta| \leq EPS$$

for all arguments a and b that do not cause underflow, overflow, division by zero, or an invalid operation. When $fl(a \odot b)$ is a closest floating point number to the true result $a \odot b$ (with ties broken arbitrarily), then rounding is called "proper" and $EPS = .5 * BASE^{1-T}$. Otherwise typically $EPS = BASE^{1-T}$, although it can sometimes be worse if arithmetic is not implemented carefully. We further say that rounding is "IEEE style" if ties are broken by rounding to the nearest number whose least significant digit is even (i.e. whose bottom bit is 0).

The function FPINFO returns the above floating point parameters, among others, to help the user understand the accuracy to which results are computed. FPINFO can return the values for either single precision or double precision. The way the precision is specified is language dependent, as is the choice of floating point parameter to return, and described in section 2.7. The names single and double may have different meanings on different machines: We have long been accustomed to single precision meaning 32-bits on all IEEE and most other machines [7], except for Cray and its emulators where single is 64-bits. And there are historical examples of 60-bit formats on some old CDC machines, etc. Nonetheless, we all agree on single precision as a phrase with a certain system-dependent meaning, and double precision too, meaning at least twice as many significant digits as single.

[3]We ignore implementation details like "hidden bits", as well as unusual representations like logarithmic arithmetic and double-double.

The values returned by FPINFO are as follows, including the values returned for IEEE single and IEEE double, the most common cases. The floating point parameters in column 1 have analogous meanings as the like-named character arguments of the LAPACK subroutine xLAMCH.[4]

Floating point parameter	Description	Value in IEEE single	Value in IEEE double
BASE	base of the machine	2	2
T	number of digits	24	53
RND	1 when proper rounding occurs in addition 0 otherwise	1	1
IEEE	1 when rounding in addition is IEEE style 0 otherwise	1	1
EMIN	minimum exponent before (gradual) underflow	-126	-1022
EMAX	maximum exponent before overflow	127	1023
EPS	machine epsilon = .5*BASE^{1-T} if RND=1 = BASE^{1-T} if RND=0	$2^{-24} \approx 5 \times 10^{-8}$	$2^{-53} \approx 10^{-16}$
PREC	EPS*BASE	2^{-23}	2^{-52}
UN	underflow threshold = BASE^{EMIN}	$2^{-126} \approx 10^{-38}$	$2^{-1022} \approx 10^{-308}$
OV	overflow threshold = BASE^{EMAX+1} * (1−EPS)	$\sim 2^{128} \approx 10^{38}$	$\sim 2^{1024} \approx 10^{308}$
SFMIN	safe minimum, such that 1/SFMIN does not overflow = UN if 1/OV<UN, else (1+EPS)/OV	$2^{-126} \approx 10^{-38}$	$2^{-1022} \approx 10^{-308}$

Table 1.9: Values returned by FPINFO

Chapter 4 defines an additional FPINFO-like function to supplement this one with additional information needed for error bounds.

1.7 Language Bindings

Each specification of a routine corresponds to an operation outlined in the functionality tables. Operations are organized analogous to the order in which they are presented in the functionality tables. The specification has the form:

NAME (*multi-word description of operation*) < *mathematical representation* >

[4]Here are the differences: In xLAMCH, UN was called RMIN and OV was called RMAX. The value of IEEE was computed by xLAMCH but not returned. xLAMCH returned EMIN+1 and EMAX+1 instead of EMIN and EMAX, respectively (this corresponds to a different choice of where to put the "point" in $d.d \cdots d * BASE^E$).

Optional brief textual description of the functionality including any restrictions that apply to all language bindings.

- Fortran 95 binding
- Fortran 77 binding
- C binding

Alternative language bindings for C++ and Java were also discussed during the meetings of the forum, but the specifications for these bindings were postponed for a future series of meetings.

1.8 Error Handling

This document supports two types of error-handling capabilities: an error handler and error return codes. Each chapter of this document, and thus each flavor of BLAS, has the choice of using either capability, whichever is more appropriate. Chapters 2 and 4 rely on an error handler, and Chapter 3 provides error return codes.

One error handler, BLAS_ERROR, is defined. A series of error return codes are also defined. Each function in this document determines when and if an error-handling mechanism is called, and its function specification must document the conditions (if any) which trigger the error handling mechanism.

1.8.1 Return Codes

Routines in the Sparse BLAS chapter utilize return codes since many of the operations need to be recoverable. In Fortran 95 and 77, the error return code of a BLAS routine is returned in the parameter `istat`, usually the last argument in the parameter list. In C, the error code is the return value of the function. In either case, the value of the error code is the integer 0 if the operation was successful. In the event of an error detection, a nonzero value is returned and control returns back to the calling program, as usual. The application is not aborted or halted, and it is the responsibility of the caller to check error status of these BLAS operations.

1.8.2 Error Handlers

The error handler defines some minimal scalar input argument checking.

> *Advice to implementors.* A BLAS supplier is free to provide multiple interfaces to the libraries, so that a second interface may perform no error checking. (*End of advice to implementors.*)

Additional error checking may be performed (for instance, checking that there are no zeros on the diagonal of a triangular solve), but these kinds of tests are too implementation-constraining to be mandated by the standard. Any additional error checking must not abort execution.

When any of the mandated scalar input argument checks fail, if the BLAS error handler is used, it must use the API given below. The default behavior of the BLAS-compliant error handler is to print an informative error message and abort execution. However, the API of this error handler is mandated by this document specifically so that a user can override the default error handler with a user-defined routine, so that this behavior can be changed. It is therefore necessary that the implementor not assume that the error handler stops execution, but rather must return explicitly before altering the routine's operands in the event of an error.

The following are defined as errors by this standard. All Fortran 95, Fortran 77, and C routines must perform the following error check.

- Any value of the operator arguments whose meaning is not specified in section A.3 is invalid.

Additionally, all Fortran 77 and C routines must perform the following error checks, unless otherwise noted in the specification of the routine.

- Any problem dimension or bandwidth (eg., m, n, k, kl, ku) less than zero
- Any vector increment (eg., incw, incx, incy, incz) equal to zero
- Any leading dimension (eg. lda, ldb, ldc, ldt) less than one
- Any leading dimension (eg. lda, ldb, ldc, ldt) less than the relevant dimension of the problem. The relevant dimension of the problem is:
  - n, for a square, symmetric, or triangular matrix
  - m, for a m $\times$ n general, non-transposed matrix
  - n, for a m $\times$ n general, transposed matrix
  - kl + ku + 1 for a m $\times$ n general band matrix
  - k + 1 for a n $\times$ n symmetric or triangular band matrix with k super- or subdiagonals

Each language binding possesses its own unique error handler. However, all error handlers minimally pass three pieces of information:

1. `RNAME`, the name of the routine in which the error occurred.
2. `IFLAG`, an integer flag which, if negative, means that parameter number `-IFLAG` caused the error, and if set to nonnegative, is an implementation-specific error code
3. `IVAL`, the value of parameter number `-IFLAG`.

Each language's BLAS error handler should print an informative error message describing the error, and halt execution. The API of the error handler is explicitly spelled out in each section, so that if this behavior is not desired by the user or higher level library provider, it may be changed by the BLAS user, overriding the BLAS's error handler with one which performs as required.

The API for each language binding is mandated in the following sections; as an advice to the implementor, an example of a BLAS-2000 compliant error handler is included as well.

F95 error handler

The Fortran 95 BLAS do not need to test the option arguments, since these are derived types and hence invalid arguments are flagged by the compiler. The only case where array dimensions are arguments to the Fortran 95 BLAS are the nonsymmetric band routines where m and kl are passed as arguments. The other array dimensions can be determined in the BLAS routines using the intrinsic function SIZE, and arrays should be checked for conformance according to the operation being performed. For example in the operation AB the second dimension of A must equal the first dimension of B. Note that, for consistency, m is included in all of the nonsymmetric band routines although in some cases it is redundant; in those cases it should be tested against the relevant array dimension.

The mandated API of the routine is:

```
    MODULE blas_error_handler
      INTERFACE blas_error
        SUBROUTINE blas_error(rname,iflag,ival)
          INTEGER, INTENT (IN) :: iflag
          INTEGER, OPTIONAL, INTENT (IN) :: ival
          CHARACTER (*), INTENT (IN) :: rname
        END SUBROUTINE blas_error
      END INTERFACE
    END MODULE blas_error_handler
```

A possible implementation would be:

```
    SUBROUTINE blas_error(rname,iflag,ival)
      ! .. Scalar Arguments ..
      ! The optional argument ival must be present when iflag is in (-98,-1)
      INTEGER, INTENT (IN) :: iflag
      INTEGER, OPTIONAL, INTENT (IN) :: ival
      CHARACTER (*), INTENT (IN) :: rname
      ! ..
      SELECT CASE (iflag)
      CASE (-99)
        WRITE (*,1000) rname
      CASE (-98:-1)
        WRITE (*,2000) rname, -iflag, ival
      CASE DEFAULT
        WRITE (*,3000) iflag, rname
      END SELECT

      STOP

1000  FORMAT ('On entry to ',A, &
       ' two or more array argument sizes do not conform')
2000  FORMAT ('On entry to ',A,' argument number',I3, &
       ' had the illegal value of ',I5)
3000  FORMAT ('Unknown error code ',I5,' raised by routine ',A)

    END SUBROUTINE blas_error
```

F77 error handler

The mandated API of the routine is:

```
       SUBROUTINE BLAS_ERROR( RNAME, IFLAG, IVAL )
       CHARACTER*(*) RNAME
       INTEGER IFLAG, IVAL
```

A possible implementation would be:

```
       SUBROUTINE BLAS_ERROR( RNAME, IFLAG, IVAL )
       CHARACTER*(*) RNAME
```

```
      INTEGER IFLAG, IVAL

      IF( IFLAG.LT.0 ) THEN
         WRITE(*,1000) RNAME, -IFLAG, IVAL
      ELSE
         WRITE(*,2000) IFLAG, RNAME
      END IF
      STOP

1000  FORMAT('On entry to ',A, ' parameter number', I3,
     $       ' had the illegal value of', I)
2000  FORMAT('Unknown error code ',I,' raised by routine',A)
      END
```

C error handler

The mandated API of the routine is:

```
void BLAS_error(char *rname, int iflag, int ival, char *form, ...)
```

A possible implementation would be:

```
#include <stdio.h>
#include <stdarg.h>
void BLAS_error(char *rname, int iflag, int ival, char *form, ...)
{
   va_list argptr;

   va_start(argptr, form);
   fprintf(stderr, "Error #%d from routine %s:\n", iflag, rname);
   if (form) vfprintf(stderr, form, argptr);
   else if (iflag < 0)
      fprintf(stderr,
         "   Parameter number %d to routine %s had the illegal value %d\n",
              -iflag, rname, ival);
   else fprintf(stderr, "   Unknown error code %d from routine %s\n",
                iflag, rname);
   exit(iflag);
}
```