A Proposal for a C interface to the BLAS

R. Clint Whaley *

February 5, 1997

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

Contents

1 Introduction **3**

2 Compatibility with present BLAS machinery **3**
2.1 Proposal 3
2.2 Motivation 3

3 ANSI C or old-style **4**
3.1 Proposal 4
3.2 Motivation 4

4 Naming scheme **4**
4.1 Proposal 4
4.2 Motivation 4

5 Array storage **4**
5.1 Proposal 4
5.2 Motivation 4

6 Handling of complex data types **5**
6.1 Proposal 5
6.2 Motivation 5

7 Return values of functions **6**
7.1 Proposal 6
7.2 Motivation 6

*Dept. of Computer Sciences, Univ. of TN, Knoxville, TN 37996, rwhaley@cs.utk.edu

8 Include file **6**
8.1 Proposal 6
8.2 Motivation 6

9 Indices **6**
9.1 Proposal 6
9.2 Motivation 6

10 Calling sequences **7**
10.1 Proposal 7
10.2 Motivation 7

11 Summation **7**

A Proposed ANSI C BLAS declarations **8**

1 Introduction

This note discusses some issues involved in choosing a C interface for the BLAS. It does not discuss implementation details. These will be discussed once an interface is agreed upon. The only time our discussion will touch implementation level details will be in insuring that the interface not be incompatible with the existing BLAS machinery.

2 Compatibility with present BLAS machinery

2.1 Proposal

We strongly propose that whatever interface we eventually arrive it, that it be structured in such a way to allow it to obtain its performance through calling the existing Fortran77 BLAS.

2.2 Motivation

My understanding is that the C interface we are discussing here is not necessarily the one we would propose were the BLAS a new idea which occurred first for the C language. Rather, this interface is intended to allow C programmers to use the already existing BLAS in a straightforward way. No new functionality will be proposed here, and we will therefore be constrained by limitations such as the data storage chosen by the present (Fortran77) BLAS in our options for this interface.

The fact that we require our interface to be compatible with the Fortran BLAS is what makes this a short term goal, rather than a new proposal. The assumption that we will be compatible with the present BLAS underlies this entire proposal. If this assumption is not valid, there is no point in discussing any other points brought up in this note.

I believe that an entirely unconstrained C interface would probably be better, but find it unlikely that there is enough demand to validate the effort involved in such a standard. Perhaps vendor input can confirm or deny this view.

At any rate, I strongly believe that we must be able to use the present BLAS if this is to be a short term goal. I believe the proper forum for an all new C interface would be the language binding of whatever new standard emerges.

I do believe that there is enough demand to justify a more modest C interface which can make use of the present BLAS machinery. A reference implementation consisting of wrappers around the Fortran77 BLAS can be readily supplied. This means that vendors are able to use their already existing BLAS to supply performance; they will have a satisfactory C interface without any new coding required. They will address the needs of their C users (not as completely as an entirely new interface would, of course), without having to repeat effort already expended on the present BLAS. This kind of proposal, which does not require large investments of time and resources, is much more likely to be widely supported.

3 ANSI C or old-style

3.1 Proposal

It is our tentative proposal that our interface assume ANSI C if there is clear benefit from such a choice. However, perhaps users or vendors could comment on the availability of ANSI as opposed to old-style C.

3.2 Motivation

We need to determine whether our interface need be implementable on versions of C prior to the ANSI standard. This effects choices we might make due to functionality differences between old-style and ANSI C. In particular, the solution to the problem raised in section 7 is dependent on this decision.

4 Naming scheme

4.1 Proposal

In order to make the naming scheme simple, we suggest simply prepending the familiar BLAS name with `C_`. For example, the Fortran77 routine `DGEMM` becomes `C_dgemm`.

4.2 Motivation

The C and Fortran77 interface cannot share the exact same namespace, since on some machines the languages' namespaces are not distinct. The simple naming scheme guarantees unique namespaces, while allowing easy translation between the interfaces.

5 Array storage

5.1 Proposal

We propose that arrays be passed as contiguous column-major storage. We should point out to the naive user that the array must be contiguous (and thus he needs some tricks if he is using a standard C 2D array). We should also make clear that if the user stores his array row-major, he can often keep this orientation in using the BLAS by setting the transpose parameter appropriately.

5.2 Motivation

The BLAS were originally proposed for Fortran77, which made the choice of matrix storage simple: Fortran77 supports column-major array storage. We need to address this issue more carefully with C, which should allow us a broader range of options.

As discussed above, I believe that we cannot ask the vendors to re-implement the underlying routines for additional language bindings. If we accept this as true, we must be able to pass the BLAS a column-major contiguous array. This immediately rules out taking C 2D arrays as our operand matrices. A 2D C array is row-major, and the rows are not constrained to be contiguous. This implies that a column of a 2D C array cannot be addressed as a vector with a constant stride, which reduces significantly the usage of the level 1 BLAS. We will therefore need to require that the user ensure that his matrices are contiguous memory.

It has been pointed out that we can accept matrices as row-major (more in-line with common C usage), and pass them to the underlying BLAS routine as transpose (thus implicitly making them column-major). However, I believe these identities break down in the complex case, where if the user passes us a matrix to be operated on which is a conjugate transpose, we would need additional space and a data copy in order to perform the conjugation.

I also believe (with myself as the only real data point), that a good deal of our user base for such an interface will come from users who up until now have been calling the Fortran77 BLAS directly from C, and thus they will be used to ensuring that their arrays are contiguous column-major storage.

6 Handling of complex data types

6.1 Proposal

We tentatively propose that complex numbers be accepted as a pointer to their underlying type. We have found this approach the most robust for previous software. If we choose this option, however, it limits our choices as discussed in section 7.

6.2 Motivation

C has no native data type for complex variables. We see essentially two choices for handling complex data types:

1. Express all complex variables as pointers to their underlying type (for example, Fortran77's `COMPLEX16` would be accepted as `double *`). The assumption here would be that complex variables are stored as two contiguous doubles or floats.

2. Use type definitions, available in a C interface BLAS include file, defining the complex data types.

If we use ANSI C (which has the ability to return structures from functions, something not standard in old style C), using the second option allows us to support functions returning complex values. However, this would require that the user use our data type himself.

The Fortran77 standard guarantees that complex numbers are stored in two consecutive numeric storage units in a storage sequence. Therefore, the first option is guaranteed to work on all systems.

Elements of both ideas are dissatisfactory. In other codes, we have found that simple typedefs for complex numbers must be changed based on the platform, which can lead to maintenance problems. We therefore tentatively give the nod to using pointers to the underlying type for complex numbers.

7 Return values of functions

7.1 Proposal

We tentatively propose to make all BLAS functions returning values return these values in output parameters. We believe this option is less fragile (though arguably less elegant, as well).

7.2 Motivation

There is some difficulties for those BLAS which are functions. In particular, returning complex values requires some thought. It can only be done if we constrain ourselves to ANSI C, where we can standardly return a structure. If we pursue this route, there will be some difficulty if the user is not using the same typedef for complex values.

The other option is to define the C interface such that we have no functions returning values. In this scheme, BLAS which formerly returned values gain an output parameter where the value is returned.

Neither of these options is a clear winner. However, it has been my experience that requiring the user to use a particular data type is burdensome.

8 Include file

8.1 Proposal

We propose a standard include file called `Cblas.h`, which minimally contains prototypes for all routines. We need to decide what else such a file should contain.

8.2 Motivation

It will probably be helpful to have a standard include file for this interface, which has prototypes for all CBLAS routines. This include file will be required if we wish to declare typedefs for complex data types.

9 Indices

9.1 Proposal

We propose that indices start at 0, in accordance with C usage. Therefore, `I□AMAX` will return a value $0 \leq$ `C_i□amax` $\leq N-1$. If the user passes in a 0 length vector, the Fortran77 routine returns 0. We propose that the C interface return -1 in such a case.

9.2 Motivation

The BLAS possess the routine `I□AMAX`, which returns the index of a vector which possesses the maximum value. In the Fortran77 interface, the returned value is $1 \leq$ `I□AMAX` $\leq N$. This falls in line with the indexing of Fortran77 arrays. C arrays, however, are indexed from 0 up to $N-1$.

10 Calling sequences

10.1 Proposal

Appendix A shows the proposed ANSI C prototypes for a few example routines. Our general scheme is:

- *Character arguments* are passed as `char *`. This preserves the ability to pass an entire word (for instance "NOTRANSPOSE") if the user so wishes. These arguments may be either upper or lower case.
- *Non-complex scalar input arguments* are passed by value. This allows the user to put in constants when desired (eg., passing 10 on the command line for `N`).
- *Complex scalar input arguments* are passed by address, since they do not exist as a predefined data type in ANSI C (see section 6 for further details).
- *Array arguments* are passed by address (see section 5 for further details).
- *Output scalar arguments* are passed by address.

10.2 Motivation

A C programmer directly calling Fortran77 is constrained to pass all variables as pointers. With a C specific interface, this situation can obviously be improved. We therefore need to dictate which parameters will be passed by value and which will be passed by address.

Note that we make very few changes beyond the pass by address or pass by value distinction. For instance, most character arguments could be replaced by predefined macros or enumerated types. I believe, however, that much of our user base for this interface will be people familiar with the Fortran77 interface, and thus we get instant familiarity by staying as close to the Fortran interface as C conveniently allows.

11 Summation

In summation, we recommend the following:

- The interface should not rule out use of the BLAS machinery used by the Fortran77 BLAS
- The naming scheme be the same as for Fortran77, with `C_` prepended.
- The array storage be column-major contiguous memory
- Complex variables be accepted as pointers to their underlying types
- BLAS functions be recast as subroutines
- A standard CBLAS include file called `Cblas.h` should be defined
- All indices start at 0 instead of 1.

A Proposed ANSI C BLAS declarations

Here are the ANSI C prototypes for some selected routines, as they would appear in our proposal. Output parameters are underlined.

void C_sdot	(int N, float *<u>dot</u>, float *X, int incX, float *Y, int incY)
void C_ccopy	(int N, float *X, int incX, float *<u>Y</u>, int incY)
void C_zgemv	(char *trans, int M, int N, double *alpha, double *A, int lda, double *X, int incX, double *beta, double *Y, int incY)
void C_dgemm	(char *transA, char *transB, int M, int N, int K, double alpha, double *A, int lda, double *B, int ldb, double beta, double *<u>C</u>, int ldc)