D R A F T

Document for the Basic Linear Algebra Subprograms (BLAS) Standard

Basic Linear Algebra Subprograms Technical (BLAST) Forum

May 19, 1998

Contents

The discussion in each chapter is not limited only to the points listed, any related topics can be included at the discretion of the chapter authors. Final acceptance of each section will be decided at the next meeting using *Robert's Rules*. See BLAST Forum homepage for more detailed rules.

Annex B

LegacyBLAS

B.1 C interface to legacy BLAS

This section discusses the proposed C interface to the legacy BLAS in some detail. Every mention of "BLAS" in this chapter should be taken to mean the legacy BLAS. Each interface decision is discussed in its own section. Each section also contains a *Considered methods* subsection, where other solutions to that particular problem are discussed, along with the reasons why those options were not chosen. These *Considered methods* subsections are indented and *italicized* in order to distinguish them from the rest of the text.

It is largely agreed among the group (and unanimous among the vendors) that user demand for a C interface to the BLAS is insufficient to motivate vendors to support a completely separate standard. This proposal therefore confines itself to an interface which can be readily supported on top of the already existing Fortran 77 callable BLAS (i.e., the legacy BLAS).

The interface is expressed in terms of ANSI/ISO C. Very few platforms fail to provide ANSI/ISO C compilers at this time, and for those platforms, free ANSI/ISO C compilers are almost always available (eg., `gcc`).

B.1.1 Naming scheme

The naming scheme consists of taking the Fortran 77 routine name, making it lower case, and adding the prefix `cblas_`. Therefore, the routine `DGEMM` becomes `cblas_dgemm`.

Considered methods

> *Various other naming schemes have been proposed, such as adding* `C_` *or* `c_` *to the name. Most of these schemes accomplish the requirement of separating the Fortran 77 and C name spaces. It was argued, however, that the addition of the* `blas` *prefix unifies the naming scheme in a logical and useful way (making it easy to search for BLAS use in a code, for instance), while not placing too great a burden on the typist. The letter* `c` *is used to distinguish this language interface from possible future interfaces.*

B.1.2 Indices and I_AMAX

The Fortran 77 BLAS return indices in the range $1 \le I \le N$ (where N is the number of entries in the dimension in question, and I is the index), in accordance with Fortran 77 array indexing conventions. This allows functions returning indices to be directly used to index standard arrays. The C interface therefore returns indices in the range $0 \leq I < N$ for the same reason.

The only BLAS routine which returns an index is the function `I_AMAX`. This function is declared to be of type `CBLAS_INDEX`, which is guaranteed to be an integer type (i.e., no cast is required when assigning to any integer type). `CBLAS_INDEX` will usually correspond to `size_t` to ensure any array can be indexed, but implementors might choose the integer type which matches their Fortran 77 `INTEGER`, for instance. It is defined that zero is returned as the index for a zero length vector (eg., For $N = 0$, `I_AMAX` will always return zero).

B.1.3 Character arguments

All arguments which were characters in the Fortran 77 interface are handled by enumerated types in the C interface. This allows for tighter error checking, and provides less opportunity for user error. The character arguments present in the Fortran 77 interface are: `SIDE`, `UPLO`, `TRANSPOSE`, and `DIAG`. This interface adds another such argument to all routines involving two dimensional arrays, `ORDER`. The standard dictates the following enumerated types:

```
enum CBLAS_ORDER     {CblasRowMajor=101, CblasColMajor=102};
enum CBLAS_TRANSPOSE {CblasNoTrans=111, CblasTrans=112, CblasConjTrans=113};
enum CBLAS_UPLO      {CblasUpper=121, CblasLower=122};
enum CBLAS_DIAG      {CblasNonUnit=131, CblasUnit=132};
enum CBLAS_SIDE      {CblasLeft=141, CblasRight=142};
```

Considered methods

The other two most commonly suggested methods were accepting these arguments as either `char *` *or* `char`. *It was noted that both of these options require twice as many comparisons as normally required to branch (so that the character may be either upper or lower case). Both methods also suffered from ambiguity (what does it mean to have* `DIAG='H'`*, for instance). If* `char` *was chosen, the words could not be written out as they can for the Fortran 77 interface (you couldn't write "NoTranspose"). If* `char *` *were used, some compilers might fail to optimize string constant use, causing unnecessary memory usage.*

The main advantage of enumerated data types, however, is that much of the error checking can be done at compile time, rather than at runtime (i.e., if the user fails to pass one of the valid options, the compiler can issue the error).

There was much discussion as to whether the integer values should be specified, or whether only the enumerated names should be so specified. The group could find no substansive way in which specifying the integer values would restrict an implementor, and specifying the integer values was seen as an aid to inter-language calls.

B.1.4 Handling of complex data types

All complex arguments are accepted as `void *`. A complex element consists of two consecutive memory locations of the underlying data type (i.e., `float` or `double`), where the first location contains the real component, and the second contains the imaginary part of the number.

In practice, programmers' methods of handling complex types in C vary. Some use various data structures (some examples are discussed below). Others accept complex numbers as arrays of the underlying type.

Complex numbers are accepted as void pointers so that widespread type casting will not be required to avoid warning or errors during compilation of complex code.

An ANSI/ISO committee is presently working on an extension to ANSI/ISO C which defines complex data types. The definition of a complex element is the same as given above, and so the handling of complex types by this interface will not need to be changed when ANSI/ISO C standard is extended.

Considered methods

> *Probably the most strongly advocated alternative was defining complex numbers via a structure such as*
> `struct NON_PORTABLE_COMPLEX {float r; float i;};` *The main problem with this solution is the lack of portability. By the ANSI/ISO C standard, elements in a structure are not guaranteed to be contiguous. With the above structure, padding between elements has been experimentally observed (on the CRAY T3D), so this problem is not purely theoretical.*
>
> *To get around padding problems within the structure, a structure such as*
> `struct NON_PORTABLE_COMPLEX {float v[2];};` *has been suggested. With this structure there will obviously be no padding between the real and imaginary parts. However, there still exists the possibility of padding between elements within an array. More importantly, this structure does not lend itself nearly as well as the first to code clarity.*
>
> *A final proposal is to define a structure which may be addressed the same as the one above (i.e.,* `ptr->r`*,* `ptr->i`*), but whose actual definition is platform dependent. Then, hopefully, various vendors will either use the above structure and ensure via their compilers its contiguousness, or they will create a different structure which can be accessed in the same way.*
>
> *This requires vendors to support something which is not in the ANSI C standard, and so there is no way to ensure this would take place. More to the point, use of such a structure turns out to not offer much in the way of real advantage, as discussed in the following section.*
>
> *All of these approaches require the programmer to either use the specified data type throughout the code which will call the BLAS, or to perform type casting on each BLAS call. When complex numbers are accepted as void pointers, no type casting or data type is dictated, with the only restriction being that a complex number have the definition given above.*

B.1.5 Return values of complex functions

BLAS routines which return complex values in Fortran 77 are instead recast as subroutines in the C interface, with the return value being an output parameter added to the end of the argument list. This allows the output parameter to be accepted as void pointers, as discussed above.

Further, the name is suffixed by `_sub`. There are two main reasons for this name change. First, the change from a function to a subroutine is a significant change, and thus the name should reflect this. More importantly, the "traditional" name space is specifically reserved for use when the forthcoming ANSI/ISO C extension is finalized. When this is done, this C interface will be extended to include functions using the "traditional" names which utilize the new ANSI/ISO complex type to return the values.

Considered methods

This is the area where use of a structure is most desired. Again, the most common suggestion is a structure such as `struct NON_PORTABLE_COMPLEX {float r; float i;};`.

If one is willing to use this structure throughout one's code, then this provides a natural and convenient mechanism. If, however, the programmer has utilized a different structure for complex, this ease of use breaks down. Then, something like the following code fragment is required:

```
NON_PORTABLE_COMPLEX ctmp;
float cdot[2];

ctmp = cblas_cdotc(n, x, 1, y, 1);
cdot[0] = ctmp.r;
cdot[1] = ctmp.i;
```

which is certainly much less convenient than: `cblas_cdotc_sub(n, x, 1, y, 1, cdot)`.

It should also be noted that the primary reason for having a function instead of a subroutine is already invalidated by C's lack of a standard complex type. Functions are most useful when the result may be used directly as part of an in-line computation. However, since ANSI/ISO C lacks support for complex arithmetic primitives or operator overloading, complex functions cannot be standardly used in this way. Since the function cannot be used as a part of a larger expression, nothing is lost by recasting it as a subroutine; indeed a slight performance win may be obtained.

B.1.6 Array arguments

Arrays are constrained to being contiguous in memory. They are accepted as pointers, not as arrays of pointers. This means that the C definition of a two dimensional array may not be used directly, since each row is an arbitrary pointer (i.e., the address of the second row cannot be obtained from the address of the first row). Note that if the user somehow ensures the C array is actually contiguous (eg. by allocating it himself), C two dimensional arrays can indeed be used.

All BLAS routines which take one or more two dimensional arrays as arguments receive precisely one extra parameter as their first argument. This argument is of the enumerated type
`enum CBLAS_ORDER {CblasRowMajor=101, CblasColMajor=102};`.
If this parameter is set to `CblasRowMajor`, it is assumed that elements within a row of the array(s) are contiguous in memory, while elements within array columns are separated by a constant stride given in the `stride` parameter (this parameter corresponds to the leading dimension [e.g. `LDA`] in the Fortran 77 interface).

If the order is given as `CblasColMajor`, elements within array columns are assumed to be contiguous, with elements within array rows separated by `stride` memory elements.

Note that there is only one `CBLAS_ORDER` parameter to a given routine: all array operands are required to use the same ordering.

Considered methods

This solution comes after much discussion. C users appear to split roughly into two camps. Those people who have a history of mixing C and Fortran 77 (in particular making use of the Fortran 77 BLAS from C), tend to use column-major arrays in order to allow ease of inter-language operations. Because of the flexibility of pointers, this is *not appreciably harder than using row-major arrays, even though C "natively" possesses row-major arrays.*

The second camp of C users are not interested in overt C/Fortran 77 interoperability, and wish to have arrays which are row-major, in accordance with standard C conventions. The idea that they must recast their row-oriented algorithms to column-major algorithms is unacceptable; many in this camp would probably not utilize any BLAS which enforced a column-major constraint.

Because both camps are fairly widely represented within the target audience, it is impossible to choose one solution to the exclusion of the other.

Column-major array storage can obviously be supported directly on top of the legacy Fortran 77 BLAS. Recent work, particularly code provided by D.P. Manley of DEC, has shown that row-major array storage may also be supported in this way with little cost. Appendix B.3 discusses this issue in detail. To preview it here, we can say the level 1 and 3 BLAS require no extra operations or storage to support row-major operations on top of the legacy BLAS. Level 2 real routines also require no extra operations or storage. Some complex level 2 routines involving the conjugate transpose will require extra storage and operations in order to form explicit conjugates. However, this will always involve vectors, not the matrix. In the worst case, we will need n extra storage, and $3n$ sign changes.

One proposal was to accept arrays as arrays of pointers, instead of as a single pointer. This would correspond exactly to the standard ANSI/ISO C two dimensional array. The problems with this approach are manifold. First, the existing Fortran 77 BLAS could not be used, since they demand contiguous (though strided) storage.

Beyond this, many of the vectors used in level 1 and level 2 BLAS come from rows or columns of two dimensional arrays. Elements within columns of row-major arrays are not uniformly strided, which means that a `n`*-element column vector would need* `n` *pointers to represent it. This then leads to vectors being accepted as arrays of pointers as well.*

Now, assuming both our one and two dimensional arrays are accepted as arrays of pointers, we have a problem when we wish to perform sub-array access. If we wish to pass an $m \times n$ subsection of a two dimensional array starting at row i and column j, we must allocate n pointers, and assign them in a section of code such as:

```
float **A, **subA;

subA = malloc(m*sizeof(float*));
for (k=0; k != m; k++) subA[k] = A[i+k] + j;
cblas_rout(... subA ...);
```

The same operation must be done if we wish to use a row or column as a vector. This is not only an inconvenience, but can add up to a non-negligible performance loss as well.

A fix for these problems is that one and two dimensional arrays be passed as arrays of pointers, and then indices are passed in to indicate the sub-portion to access. Thus you have a call that looks like: `cblas_rout(... A, i, j, ...);`*. This solution still requires some additional tweaks to allow using two dimensional array rows and columns as vectors. Further, it is still not possible to support this interface on top of the Fortran 77 BLAS. Finally, users presently using contiguous storage arrays will have to malloc the array of pointers as shown above.*

With the adopted solution, the array is passed as a single pointer, which, assuming the array has been contiguously allocated, can be easily obtained by: `cblas_rout(... &A[i][j] ...);`.

B.1.7 C interface include file

The C interface to the BLAS will have a standard include file, called `cblas.h`, which minimally contains the definition of the CBLAS types and ANSI/ISO C prototypes for all BLAS routines. It is not an error to include this file multiple times. Section B.2 contains an example of a minimal `cblas.h`.

ADVICE TO THE IMPLEMENTOR:
Note that the vendor is not constrained to using precisely this include file; only the enumerated type definitions are fully specified. The implementor is free to make any other changes which are not apparent to the user. For instance, all matrix dimensions might be accepted as `size_t` *instead of* `int`*, or the implementor might choose to make some routines inline.*

B.1.8 Error checking

The C interface to the legacy BLAS must supply error checking corresponding to that provided by the reference Fortran 77 BLAS implementation.

B.1.9 Rules for obtaining the C interface from the Fortran 77

- The Fortran 77 routine name is changed to lower case, and prefixed by `cblas_`.
- All routines which accept two dimensional arrays (i.e., level 2 and 3), acquire a new parameter of type `CBLAS_ORDER` as their first argument, which determines if the two dimensional arrays are row or column major.
- *Character arguments* are replaced by the appropriate enumerated type, as shown in Section B.1.3.
- *Input arguments* are declared with the `const` modifier.
- *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 as void pointers, since they do not exist as a predefined data type in ANSI/ISO C.
- *Array arguments* are passed by address.
- *Output scalar arguments* are passed by address.
- *Complex functions* become subroutines which return the result via a void pointer, added as the last parameter. The name is suffixed with `_sub`.

B.2 cblas.h include file

```
#ifndef CBLAS_H
#define CBLAS_H
#include <stddef.h>
```

```
/*
 * Enumerated and derived types
 */
#define CBLAS_INDEX size_t  /* this may vary between platforms */
enum CBLAS_ORDER     {CblasRowMajor=101, CblasColMajor=102};
enum CBLAS_TRANSPOSE {CblasNoTrans=111, CblasTrans=112, CblasConjTrans=113};
enum CBLAS_UPLO      {CblasUpper=121, CblasLower=122};
enum CBLAS_DIAG      {CblasNonUnit=131, CblasUnit=132};
enum CBLAS_SIDE      {CblasLeft=141, CblasRight=142};

/*
 * ===========================================================================
 * Prototypes for level 1 BLAS functions (complex are recast as routines)
 * ===========================================================================
 */
float  cblas_sdsdot(const int N, const float alpha, const float *X,
                    const int incX, const float *Y, const int incY);
double cblas_dsdot(const int N, const float *X, const int incX, const float *Y,
                   const int incY);
float  cblas_sdot(const int N, const float  *X, const int incX,
                  const float  *Y, const int incY);
double cblas_ddot(const int N, const double *X, const int incX,
                  const double *Y, const int incY);

/*
 * Functions having prefixes Z and C only
 */
void   cblas_cdotu_sub(const int N, const void *X, const int incX,
                       const void *Y, const int incY, void *dotu);
void   cblas_cdotc_sub(const int N, const void *X, const int incX,
                       const void *Y, const int incY, void *dotc);

void   cblas_zdotu_sub(const int N, const void *X, const int incX,
                       const void *Y, const int incY, void *dotu);
void   cblas_zdotc_sub(const int N, const void *X, const int incX,
                       const void *Y, const int incY, void *dotc);


/*
 * Functions having prefixes S D SC DZ
 */
float  cblas_snrm2(const int N, const float *X, const int incX);
float  cblas_sasum(const int N, const float *X, const int incX);

double cblas_dnrm2(const int N, const double *X, const int incX);
double cblas_dasum(const int N, const double *X, const int incX);
```

```
float  cblas_scnrm2(const int N, const void *X, const int incX);
float  cblas_scasum(const int N, const void *X, const int incX);

double cblas_dznrm2(const int N, const void *X, const int incX);
double cblas_dzasum(const int N, const void *X, const int incX);


/*
 * Functions having standard 4 prefixes (S D C Z)
 */
CBLAS_INDEX cblas_isamax(const int N, const float  *X, const int incX);
CBLAS_INDEX cblas_idamax(const int N, const double *X, const int incX);
CBLAS_INDEX cblas_icamax(const int N, const void   *X, const int incX);
CBLAS_INDEX cblas_izamax(const int N, const void   *X, const int incX);

/*
 * ===========================================================================
 * Prototypes for level 1 BLAS routines
 * ===========================================================================
 */

/*
 * Routines with standard 4 prefixes (s, d, c, z)
 */
void cblas_sswap(const int N, float *X, const int incX,
                 float *Y, const int incY);
void cblas_scopy(const int N, const float *X, const int incX,
                 float *Y, const int incY);
void cblas_saxpy(const int N, const float alpha, const float *X,
                 const int incX, float *Y, const int incY);

void cblas_dswap(const int N, double *X, const int incX,
                 double *Y, const int incY);
void cblas_dcopy(const int N, const double *X, const int incX,
                 double *Y, const int incY);
void cblas_daxpy(const int N, const double alpha, const double *X,
                 const int incX, double *Y, const int incY);

void cblas_cswap(const int N, void *X, const int incX,
                 void *Y, const int incY);
void cblas_ccopy(const int N, const void *X, const int incX,
                 void *Y, const int incY);
void cblas_caxpy(const int N, const void *alpha, const void *X,
                 const int incX, void *Y, const int incY);

void cblas_zswap(const int N, void *X, const int incX,
                 void *Y, const int incY);
void cblas_zcopy(const int N, const void *X, const int incX,
```

```
                 void *Y, const int incY);
void cblas_zaxpy(const int N, const void *alpha, const void *X,
                 const int incX, void *Y, const int incY);


/*
 * Routines with S and D prefix only
 */
void cblas_srotg(float *a, float *b, float *c, float *s);
void cblas_srotmg(float *d1, float *d2, float *b1, const float b2, float *P);
void cblas_srot(const int N, float *X, const int incX,
                float *Y, const int incY, const float c, const float s);
void cblas_srotm(const int N, float *X, const int incX,
                float *Y, const int incY, const float *P);

void cblas_drotg(double *a, double *b, double *c, double *s);
void cblas_drotmg(double *d1, double *d2, double *b1, const double b2, double *P);
void cblas_drot(const int N, double *X, const int incX,
                double *Y, const int incY, const double c, const double s);
void cblas_drotm(const int N, double *X, const int incX,
                double *Y, const int incY, const double *P);


/*
 * Routines with S D C Z CS and ZD prefixes
 */
void cblas_sscal(const int N, const float alpha, float *X, const int incX);
void cblas_dscal(const int N, const double alpha, double *X, const int incX);
void cblas_cscal(const int N, const void *alpha, void *X, const int incX);
void cblas_zscal(const int N, const void *alpha, void *X, const int incX);
void cblas_csscal(const int N, const float alpha, void *X, const int incX);
void cblas_zdscal(const int N, const double alpha, void *X, const int incX);

/*
 * ===========================================================================
 * Prototypes for level 2 BLAS
 * ===========================================================================
 */

/*
 * Routines with standard 4 prefixes (S, D, C, Z)
 */
void cblas_sgemv(const enum CBLAS_ORDER order,
                 const enum CBLAS_TRANSPOSE TransA, const int M, const int N,
                 const float alpha, const float *A, const int lda,
                 const float *X, const int incX, const float beta,
                 float *Y, const int incY);
void cblas_sgbmv(const enum CBLAS_ORDER order,
```

```
                 const enum CBLAS_TRANSPOSE TransA, const int M, const int N,
                 const int KL, const int KU, const float alpha,
                 const float *A, const int lda, const float *X,
                 const int incX, const float beta, float *Y, const int incY);
void cblas_strmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const float *A, const int lda,
                 float *X, const int incX);
void cblas_stbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const int K, const float *A, const int lda,
                 float *X, const int incX);
void cblas_stpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const float *Ap, float *X, const int incX);
void cblas_strsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const float *A, const int lda, float *X,
                 const int incX);
void cblas_stbsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const int K, const float *A, const int lda,
                 float *X, const int incX);
void cblas_stpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const float *Ap, float *X, const int incX);

void cblas_dgemv(const enum CBLAS_ORDER order,
                 const enum CBLAS_TRANSPOSE TransA, const int M, const int N,
                 const double alpha, const double *A, const int lda,
                 const double *X, const int incX, const double beta,
                 double *Y, const int incY);
void cblas_dgbmv(const enum CBLAS_ORDER order,
                 const enum CBLAS_TRANSPOSE TransA, const int M, const int N,
                 const int KL, const int KU, const double alpha,
                 const double *A, const int lda, const double *X,
                 const int incX, const double beta, double *Y, const int incY);
void cblas_dtrmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const double *A, const int lda,
                 double *X, const int incX);
void cblas_dtbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const int K, const double *A, const int lda,
                 double *X, const int incX);
void cblas_dtpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const double *Ap, double *X, const int incX);
```

```
void cblas_dtrsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const double *A, const int lda, double *X,
                 const int incX);
void cblas_dtbsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const int K, const double *A, const int lda,
                 double *X, const int incX);
void cblas_dtpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const double *Ap, double *X, const int incX);

void cblas_cgemv(const enum CBLAS_ORDER order,
                 const enum CBLAS_TRANSPOSE TransA, const int M, const int N,
                 const void *alpha, const void *A, const int lda,
                 const void *X, const int incX, const void *beta,
                 void *Y, const int incY);
void cblas_cgbmv(const enum CBLAS_ORDER order,
                 const enum CBLAS_TRANSPOSE TransA, const int M, const int N,
                 const int KL, const int KU, const void *alpha,
                 const void *A, const int lda, const void *X,
                 const int incX, const void *beta, void *Y, const int incY);
void cblas_ctrmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const void *A, const int lda,
                 void *X, const int incX);
void cblas_ctbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const int K, const void *A, const int lda,
                 void *X, const int incX);
void cblas_ctpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const void *Ap, void *X, const int incX);
void cblas_ctrsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const void *A, const int lda, void *X,
                 const int incX);
void cblas_ctbsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const int K, const void *A, const int lda,
                 void *X, const int incX);
void cblas_ctpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const void *Ap, void *X, const int incX);

void cblas_zgemv(const enum CBLAS_ORDER order,
                 const enum CBLAS_TRANSPOSE TransA, const int M, const int N,
                 const void *alpha, const void *A, const int lda,
```

```
                 const void *X, const int incX, const void *beta,
                 void *Y, const int incY);
void cblas_zgbmv(const enum CBLAS_ORDER order,
                 const enum CBLAS_TRANSPOSE TransA, const int M, const int N,
                 const int KL, const int KU, const void *alpha,
                 const void *A, const int lda, const void *X,
                 const int incX, const void *beta, void *Y, const int incY);
void cblas_ztrmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const void *A, const int lda,
                 void *X, const int incX);
void cblas_ztbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const int K, const void *A, const int lda,
                 void *X, const int incX);
void cblas_ztpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const void *Ap, void *X, const int incX);
void cblas_ztrsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const void *A, const int lda, void *X,
                 const int incX);
void cblas_ztbsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const int K, const void *A, const int lda,
                 void *X, const int incX);
void cblas_ztpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
                 const int N, const void *Ap, void *X, const int incX);


/*
 * Routines with S and D prefixes only
 */
void cblas_ssymv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const int N, const float alpha, const float *A,
                 const int lda, const float *X, const int incX,
                 const float beta, float *Y, const int incY);
void cblas_ssbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const int N, const int K, const float alpha, const float *A,
                 const int lda, const float *X, const int incX,
                 const float beta, float *Y, const int incY);
void cblas_sspmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const int N, const float alpha, const float *Ap,
                 const float *X, const int incX,
                 const float beta, float *Y, const int incY);
void cblas_sger(const enum CBLAS_ORDER order, const int M, const int N,
                const float alpha, const float *X, const int incX,
```

```
                const float *Y, const int incY, float *A, const int lda);
void cblas_ssyr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                const int N, const float alpha, const float *X,
                const int incX, float *A, const int lda);
void cblas_sspr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                const int N, const float alpha, const float *X,
                const int incX, float *Ap);
void cblas_ssyr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                const int N, const float alpha, const float *X,
                const int incX, const float *Y, const int incY, float *A,
                const int lda);
void cblas_sspr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                const int N, const float alpha, const float *X,
                const int incX, const float *Y, const int incY, float *A);

void cblas_dsymv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const int N, const double alpha, const double *A,
                 const int lda, const double *X, const int incX,
                 const double beta, double *Y, const int incY);
void cblas_dsbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const int N, const int K, const double alpha, const double *A,
                 const int lda, const double *X, const int incX,
                 const double beta, double *Y, const int incY);
void cblas_dspmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const int N, const double alpha, const double *Ap,
                 const double *X, const int incX,
                 const double beta, double *Y, const int incY);
void cblas_dger(const enum CBLAS_ORDER order, const int M, const int N,
                const double alpha, const double *X, const int incX,
                const double *Y, const int incY, double *A, const int lda);
void cblas_dsyr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                const int N, const double alpha, const double *X,
                const int incX, double *A, const int lda);
void cblas_dspr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                const int N, const double alpha, const double *X,
                const int incX, double *Ap);
void cblas_dsyr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                const int N, const double alpha, const double *X,
                const int incX, const double *Y, const int incY, double *A,
                const int lda);
void cblas_dspr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                const int N, const double alpha, const double *X,
                const int incX, const double *Y, const int incY, double *A);


/*
 * Routines with C and Z prefixes only
 */
```

```
void cblas_chemv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const int N, const void *alpha, const void *A,
                 const int lda, const void *X, const int incX,
                 const void *beta, void *Y, const int incY);
void cblas_chbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const int N, const int K, const void *alpha, const void *A,
                 const int lda, const void *X, const int incX,
                 const void *beta, void *Y, const int incY);
void cblas_chpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const int N, const void *alpha, const void *Ap,
                 const void *X, const int incX,
                 const void *beta, void *Y, const int incY);
void cblas_cgeru(const enum CBLAS_ORDER order, const int M, const int N,
                 const void *alpha, const void *X, const int incX,
                 const void *Y, const int incY, void *A, const int lda);
void cblas_cgerc(const enum CBLAS_ORDER order, const int M, const int N,
                 const void *alpha, const void *X, const int incX,
                 const void *Y, const int incY, void *A, const int lda);
void cblas_cher(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                const int N, const float alpha, const void *X, const int incX,
                void *A, const int lda);
void cblas_chpr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                const int N, const float *alpha, const void *X,
                const int incX, void *A);
void cblas_cher2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
                const void *alpha, const void *X, const int incX,
                const void *Y, const int incY, void *A, const int lda);
void cblas_chpr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
                const void *alpha, const void *X, const int incX,
                const void *Y, const int incY, void *Ap);

void cblas_zhemv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const int N, const void *alpha, const void *A,
                 const int lda, const void *X, const int incX,
                 const void *beta, void *Y, const int incY);
void cblas_zhbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const int N, const int K, const void *alpha, const void *A,
                 const int lda, const void *X, const int incX,
                 const void *beta, void *Y, const int incY);
void cblas_zhpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                 const int N, const void *alpha, const void *Ap,
                 const void *X, const int incX,
                 const void *beta, void *Y, const int incY);
void cblas_zgeru(const enum CBLAS_ORDER order, const int M, const int N,
                 const void *alpha, const void *X, const int incX,
                 const void *Y, const int incY, void *A, const int lda);
void cblas_zgerc(const enum CBLAS_ORDER order, const int M, const int N,
                 const void *alpha, const void *X, const int incX,
```

```
                 const void *Y, const int incY, void *A, const int lda);
void cblas_zher(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                const int N, const double alpha, const void *X, const int incX,
                void *A, const int lda);
void cblas_zhpr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
                const int N, const double *alpha, const void *X,
                const int incX, void *A);
void cblas_zher2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
                const void *alpha, const void *X, const int incX,
                const void *Y, const int incY, void *A, const int lda);
void cblas_zhpr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo, const int N,
                const void *alpha, const void *X, const int incX,
                const void *Y, const int incY, void *Ap);

/*
 * ===========================================================================
 * Prototypes for level 3 BLAS
 * ===========================================================================
 */

/*
 * Routines with standard 4 prefixes (S, D, C, Z)
 */
void cblas_sgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_TRANSPOSE TransB, const int M, const int N,
                 const int K, const float alpha, const float *A,
                 const int lda, const float *B, const int ldb,
                 const float beta, float *C, const int ldc);
void cblas_ssymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
                 const enum CBLAS_UPLO Uplo, const int M, const int N,
                 const float alpha, const float *A, const int lda,
                 const float *B, const int ldb, const float beta,
                 float *C, const int ldc);
void cblas_ssyrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE Trans, const int N, const int K,
                 const float alpha, const float *A, const int lda,
                 const float beta, float *C, const int ldc);
void cblas_ssyr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
                  const enum CBLAS_TRANSPOSE Trans, const int N, const int K,
                  const float alpha, const float *A, const int lda,
                  const float *B, const int ldb, const float beta,
                  float *C, const int ldc);
void cblas_strmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
                 const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_DIAG Diag, const int M, const int N,
                 const float alpha, const float *A, const int lda,
                 float *B, const int ldb);
void cblas_strsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
```

```
                  const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA,
                  const enum CBLAS_DIAG Diag, const int M, const int N,
                  const float alpha, const float *A, const int lda,
                  float *B, const int ldb);

void cblas_dgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_TRANSPOSE TransB, const int M, const int N,
                 const int K, const double alpha, const double *A,
                 const int lda, const double *B, const int ldb,
                 const double beta, double *C, const int ldc);
void cblas_dsymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
                 const enum CBLAS_UPLO Uplo, const int M, const int N,
                 const double alpha, const double *A, const int lda,
                 const double *B, const int ldb, const double beta,
                 double *C, const int ldc);
void cblas_dsyrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE Trans, const int N, const int K,
                 const double alpha, const double *A, const int lda,
                 const double beta, double *C, const int ldc);
void cblas_dsyr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
                  const enum CBLAS_TRANSPOSE Trans, const int N, const int K,
                  const double alpha, const double *A, const int lda,
                  const double *B, const int ldb, const double beta,
                  double *C, const int ldc);
void cblas_dtrmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
                 const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_DIAG Diag, const int M, const int N,
                 const double alpha, const double *A, const int lda,
                 double *B, const int ldb);
void cblas_dtrsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
                 const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_DIAG Diag, const int M, const int N,
                 const double alpha, const double *A, const int lda,
                 double *B, const int ldb);

void cblas_cgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_TRANSPOSE TransB, const int M, const int N,
                 const int K, const void *alpha, const void *A,
                 const int lda, const void *B, const int ldb,
                 const void *beta, void *C, const int ldc);
void cblas_csymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
                 const enum CBLAS_UPLO Uplo, const int M, const int N,
                 const void *alpha, const void *A, const int lda,
                 const void *B, const int ldb, const void *beta,
                 void *C, const int ldc);
void cblas_csyrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE Trans, const int N, const int K,
                 const void *alpha, const void *A, const int lda,
```

```
                 const void *beta, void *C, const int ldc);
void cblas_csyr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
                  const enum CBLAS_TRANSPOSE Trans, const int N, const int K,
                  const void *alpha, const void *A, const int lda,
                  const void *B, const int ldb, const void *beta,
                  void *C, const int ldc);
void cblas_ctrmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
                 const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_DIAG Diag, const int M, const int N,
                 const void *alpha, const void *A, const int lda,
                 void *B, const int ldb);
void cblas_ctrsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
                 const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_DIAG Diag, const int M, const int N,
                 const void *alpha, const void *A, const int lda,
                 void *B, const int ldb);

void cblas_zgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_TRANSPOSE TransB, const int M, const int N,
                 const int K, const void *alpha, const void *A,
                 const int lda, const void *B, const int ldb,
                 const void *beta, void *C, const int ldc);
void cblas_zsymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
                 const enum CBLAS_UPLO Uplo, const int M, const int N,
                 const void *alpha, const void *A, const int lda,
                 const void *B, const int ldb, const void *beta,
                 void *C, const int ldc);
void cblas_zsyrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE Trans, const int N, const int K,
                 const void *alpha, const void *A, const int lda,
                 const void *beta, void *C, const int ldc);
void cblas_zsyr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
                  const enum CBLAS_TRANSPOSE Trans, const int N, const int K,
                  const void *alpha, const void *A, const int lda,
                  const void *B, const int ldb, const void *beta,
                  void *C, const int ldc);
void cblas_ztrmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
                 const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_DIAG Diag, const int M, const int N,
                 const void *alpha, const void *A, const int lda,
                 void *B, const int ldb);
void cblas_ztrsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
                 const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_DIAG Diag, const int M, const int N,
                 const void *alpha, const void *A, const int lda,
                 void *B, const int ldb);
```

```
/*
 * Routines with prefixes C and Z only
 */
void cblas_chemm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
                 const enum CBLAS_UPLO Uplo, const int M, const int N,
                 const void *alpha, const void *A, const int lda,
                 const void *B, const int ldb, const void *beta,
                 void *C, const int ldc);
void cblas_cherk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE Trans, const int N, const int K,
                 const float alpha, const void *A, const int lda,
                 const float beta, void *C, const int ldc);
void cblas_cher2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
                  const enum CBLAS_TRANSPOSE Trans, const int N, const int K,
                  const void *alpha, const void *A, const int lda,
                  const void *B, const int ldb, const float beta,
                  void *C, const int ldc);

void cblas_zhemm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
                 const enum CBLAS_UPLO Uplo, const int M, const int N,
                 const void *alpha, const void *A, const int lda,
                 const void *B, const int ldb, const void *beta,
                 void *C, const int ldc);
void cblas_zherk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE Trans, const int N, const int K,
                 const double alpha, const void *A, const int lda,
                 const double beta, void *C, const int ldc);
void cblas_zher2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
                  const enum CBLAS_TRANSPOSE Trans, const int N, const int K,
                  const void *alpha, const void *A, const int lda,
                  const void *B, const int ldb, const double beta,
                  void *C, const int ldc);

#endif
```

B.3 Using Fortran 77 BLAS to support row-major BLAS operations

This section is not part of the standard per se. Rather, it exists as an advice to the implementor on how row-major BLAS operations may be implemented using column-major BLAS. This allows vendors to leverage years of Fortran 77 BLAS developement in producing the C BLAS.

Before this issue is examined in detail, a few general observations on array storage are helpful. We must distinguish between the matrix and the array which is used to store the matrix. The matrix, and its rows and columns, have mathematical meaning. The array is simply the method of storing the matrix, and its rows and columns are significant only for memory addressing.

Thus we see we can store the columns of a matrix in the rows of an array, for instance. When this occurs in the BLAS, the matrix is said to be stored in transposed form.

A row-major array stores elements along a row in contiguous storage, and separates the column elements by some constant stride (often the actual length of a row). Column-major arrays have contiguous columns, and strided rows. The importance of this is to note that a row-major array storing a matrix in the natural way, is a transposed column-major array (i.e., it can be thought of as a column-major array where the rows of the matrix are stored in the columns of the array).

Similarly, an upper triangular row-major array corresponds to a transposed lower triangular column-major array (the same is true in reverse [i.e., lower-to-upper], obviously). To see this, simply think of what a upper triangular matrix stored in a row-major array looks like. The first n entries contain the first matrix row, followed by a non-negative gap, followed by the second matrix row.

If this same array is viewed as column-major, the first n entries are a column, instead of a row, so that the columns of the array store the rows of the matrix (i.e., it is transposed). This means that if we wish to use the Fortran 77 (column-major) BLAS with triangular matrices coming from C (possibly row-major), we will be reversing the setting of `UPLO`, while simultaneously reversing the setting of `TRANS` (this gets slightly more complicated when the conjugate transpose is involved, as we will see).

Finally, note that if a matrix is symmetric or Hermitian, its rows are the same as its columns, so we may merely switch `UPLO`, without bothering with `TRANS`.

In the BLAS, there are two separate cases of importance. one dimensional arrays (storage for vectors) have the same meaning in both C and Fortran 77, so if we are solving a linear algebra problem who's answer is a vector, we will need to solve the same problem for both languages. However, if the answer is a matrix, in terms of calling routines which use column-major storage from one using row-major storage, we will want to solve the *transpose* of the problem.

To get an idea of what this means, consider a contrived example. Say we have routines for simple matrix-matrix and matrix-vector multiply. The vector operation is $y \leftarrow A \times x$, and the matrix operation is $C \leftarrow A \times B$. Now say we are implementing these as calls from row-major array storage to column-major storage. Since the matrix-vector multiply's answer is a vector, the problem we are solving is remains the same, but we must remember that our C array A is a Fortran 77 A^T. On the other hand, the matrix-matrix multiply has a matrix for a result, so when the differing array storage is taken into account, the problem we want to solve is $C^T \leftarrow B^T \times A^T$.

This last example demonstrates another general result. Some level 3 BLAS contain a `SIDE` parameter, determining which side a matrix is applied on. In general, if we are solving the transpose of this operation, the side parameter will be reversed.

With these general principles, it is possible to show that all that row-major level 3 BLAS can be expressed in terms of column-major BLAS without any extra array storage or extra operations. In the level 2 BLAS, no extra storage or array accesses are required for the real routines. Complex routines involving the conjugate transpose, however, may require a n-element temporary, and up to $3n$ more operations (vendors may avoid all extra workspace and operations by overloading the `TRANS` option for the level 2 BLAS: letting it also allow conjugation without doing the transpose). The level 1 BLAS, which deal exclusively with vectors, are unaffected by this storage issue.

With these ideas in mind, we will now show how to support a row-major BLAS on top of a column major BLAS. This information will be presented in tabular form. For brevity, row-major storage will be referred to as coming from C (even though column-major arrays can also come from C), while column-major storage will be referred to as F77.

Each table will show a BLAS invocation coming from C, the operation that the BLAS should perform, the operation required once F77 storage is taken into account (if this changes), and the call to the appropriate F77 BLAS. Not every possible combination of parameters is shown, since many are simply reflections of another (i.e., when we are applying the `Upper, NoTranspose` becomes `Lower, Transpose` rule, we will show it for only the upper case. In order to make the notation more concise, let us define $\overline{x}$ to be $conj(x)$.

B.3.1 Level 2 BLAS

GEMV

C call	cblas_cgemv(CblasRowMajor, CblasNoTrans, m, n, α, A, lda, x, incx, β, y, incy)
op	$y \leftarrow \alpha A x + \beta y$
F77 call	CGEMV('T', n, m, α, A, lda, x, incx, β, y, incy)

C call	cblas_cgemv(CblasRowMajor, CblasTrans, m, n, α, A, lda, x, incx, β, y, incy)
op	$y \leftarrow \alpha A^T x + \beta y$
F77 call	CGEMV('N', n, m, α, A, lda, x, incx, β, y, incy)

C call	cblas_cgemv(CblasRowMajor, CblasConjTrans, m, n, α, A, lda, x, incx, β, y, incy)
op	$y \leftarrow \alpha A^H x + \beta y \Rightarrow \overline{(\overline{y} \leftarrow \overline{\alpha} A^T \overline{x} + \overline{\beta}\overline{y})}$
F77 call	CGEMV('N', n, m, $\overline{\alpha}$, A, lda, $\overline{x}$, 1, $\overline{\beta}$, $\overline{y}$, incy)

Note that we switch the value of transpose to handle the row/column major ordering difference. In the last case, we will require n elements of workspace so that we may store the conjugated vector $\overline{x}$. Then, we set $y = \overline{y}$, and make the call. This gives us the conjugate of the answer, so we once again set $y = \overline{y}$. Therefore, we see that to support the conjugate transpose, we will need to allocate an n-element vector, and perform $2m + n$ extra operations.

SYMV

SYMV requires no extra workspace or operations.

C call	cblas_csymv(CblasRowMajor, CblasUpper, n, α, A, lda, x, incx, β, y, incy)
op	$y \leftarrow \alpha A x + \beta y \Rightarrow y \leftarrow \alpha A^T x + \beta y$
F77 call	CSYMV('L', n, α, A, lda, x, incx, β, y, incy)

HEMV

HEMV routine requires $3n$ conjugations, and n extra storage.

C call	cblas_chemv(CblasRowMajor, CblasUpper, n, α, A, lda, x, incx, β, y, incy)
op	$y \leftarrow \alpha A x + \beta y \Rightarrow y \leftarrow \alpha A^H x + \beta y \Rightarrow \overline{(\overline{y} \leftarrow \overline{\alpha} A^T \overline{x} + \overline{\beta}\overline{y})}$
F77 call	CHEMV('L', n, $\overline{\alpha}$, A, lda, $\overline{x}$, incx, $\overline{\beta}$, $\overline{y}$, incy)

TRMV/TRSV

C call	`cblas_ctrmv(CblasRowMajor, CblasUpper, CblasNoTrans, diag, n,` α`, A, lda, x, incx)`
op	$x \leftarrow \alpha A x$
F77 call	`CTRMV('L', 'T', diag, n,` α`, A, lda, x, incx)`

C call	`cblas_ctrmv(CblasRowMajor, CblasUpper, CblasTrans, diag, n,` α`, A, lda, x, incx)`
op	$x \leftarrow \alpha A^T x$
F77 call	`CTRMV('L', 'N', diag, n,` α`, A, lda, x, incx)`

C call	`cblas_ctrmv(CblasRowMajor, CblasUpper, CblasConjTrans, diag, n,` α`, A, lda, x, incx)`
op	$x \leftarrow \alpha A^H x \Rightarrow \overline{(\overline{x} = \overline{\alpha} A^T \overline{x})}$
F77 call	`CTRMV('L', 'N', diag, n,` $\overline{\alpha}$`, A, lda,` $\overline{x}$`, incx)`

Again, we see that we will need some extra operations when we are handling the conjugate transpose. We need a temporary scalar to hold $\overline{\alpha}$, and then we conjugate x before the call, giving us the conjugate of the answer we seek. We then conjugate this again to return the correct answer. This routine therefore needs $2n$ extra operations for the complex conjugate case.

The calls with the C array being `Lower` are merely the reflection of these calls, and thus are not shown. The analysis for TRMV is the same, since it involves the same principle of what a transpose of a triangular matrix is.

GER/GERU

This is our first routine that has a matrix as the solution. Recalling that this means we solve the transpose of the original problem, we get:

C call	`cblas_cgeru(CblasRowMajor, m, n,` α`, x, incx, y, incy, A, lda)`
C op	$A \leftarrow \alpha x y^T + A$
F77 op	$A^T \leftarrow \alpha y x^T + A^T$
F77 call	`CGERU(n, m,` α`, y, incy, x, incx, A, lda)`

No extra storage or operations are required.

GERC

C call	`cblas_cgerc(CblasRowMajor, m, n,` α`, x, incx, y, incy, A, lda)`
C op	$A \leftarrow \alpha x y^H + A$
F77 op	$A^T \leftarrow \alpha (x y^H)^T + A^T = \alpha \overline{y} x^T + A^T$
F77 call	`CGERU(n, m,` α`,` $\overline{y}$`, incy, x, incx, A, lda)`

Note that we need to allocate n-element workspace to hold the conjugated y, and we call `GERU`, not `GERC`.

HER

C call	`cblas_cher(CblasRowMajor, CblasUpper, n,` α`, x, incx, A, lda)`
C op	$A \leftarrow \alpha x x^H + A$
F77 op	$A^T \leftarrow \alpha \overline{x} x^T + A^T$
F77 call	`CHER('L', n,` α`,` $\overline{x}$`, 1, A, lda)`

Again, we have an n-element workspace and n extra operations.

HER2

C call	`cblas_cher2(CblasRowMajor, CblasUpper, n,` α`, x, incx, y, incy, A, lda)`
C op	$A \leftarrow \alpha x y^H + y(\alpha x)^H + A$
F77 op	$A^T \leftarrow \alpha \overline{y} x^T + \overline{\alpha x} y^T + A^T = \alpha \overline{y}(\overline{x})^H + \overline{x}(\alpha \overline{y})^H + A^T$
F77 call	`CHER2('L', n,` α`,` $\overline{y}$`, 1,` $\overline{x}$`, 1, A, lda)`

So we need $2n$ extra workspace and operations to form the conjugates of x and y.

SYR

C call	`cblas_ssyr(CblasRowMajor, CblasUpper, n,` α`, x, incx, A, lda)`
C op	$A \leftarrow \alpha x x^T + A$
F77 op	$A^T \leftarrow \alpha x x^T + A^T$
F77 call	`SSYR('L', n,` α`, x, incx, A, lda)`

No extra storage or operations required.

SYR2

C call	`cblas_ssyr2(CblasRowMajor, CblasUpper, n,` α`, x, incx, y, incy, A, lda)`
C op	$A \leftarrow \alpha x y^T + \alpha y x^T + A$
F77 op	$A^T \leftarrow \alpha y x^T + \alpha x y^T + A^T$
F77 call	`SSYR2('L', n,` α`, y, incy, x, incx, A, lda)`

No extra storage or operations required.

B.3.2 Level 3 BLAS

GEMM

C call	cblas_cgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha AB + \beta C$
F77 op	$C^T \leftarrow \alpha B^T A^T + \beta C^T$
F77 call	CGEMM('N', 'N', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasNoTrans, CblasTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha AB^T + \beta C$
F77 op	$C^T \leftarrow \alpha BA^T + \beta C^T$
F77 call	CGEMM('T', 'N', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasNoTrans, CblasConjTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha AB^H + \beta C$
F77 op	$C^T \leftarrow \alpha \overline{B} A^T + \beta C^T$
F77 call	CGEMM('C', 'N', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasTrans, CblasNoTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^T B + \beta C$
F77 op	$C^T \leftarrow \alpha B^T A + \beta C^T$
F77 call	CGEMM('N', 'T', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasTrans, CblasTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^T B^T + \beta C$
F77 op	$C^T \leftarrow \alpha BA + \beta C^T$
F77 call	CGEMM('T', 'T', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasTrans, CblasConjTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^T B^H + \beta C$
F77 op	$C^T \leftarrow \alpha \overline{B} A + \beta C^T$
F77 call	CGEMM('C', 'T', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasConjTrans, CblasNoTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^H B + \beta C$
F77 op	$C^T \leftarrow \alpha B^T \overline{A} + \beta C^T$
F77 call	CGEMM('N', 'C', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasConjTrans, CblasTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^H B^T + \beta C$
F77 op	$C^T \leftarrow \alpha B \overline{A} + \beta C^T$
F77 call	CGEMM('T', 'C', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call	cblas_cgemm(CblasRowMajor, CblasConjTrans, CblasConjTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op	$C \leftarrow \alpha A^H B^H + \beta C$
F77 op	$C^T \leftarrow \alpha \overline{BA} + \beta C^T$
F77 call	CGEMM('C', 'C', n, m, k, α, B, ldb, A, lda, β, C, ldc)

SYMM/HEMM

C call	`cblas_chemm(CblasRowMajor, CblasLeft, CblasUpper, m, n,` α`, A, lda, B, ldb,` β`, C, ldc)`
C op	$C \leftarrow \alpha AB + \beta C$
F77 op	$C^T \leftarrow \alpha B^T A^T + \beta C^T$
F77 call	`CHEMM('R', 'L', n, m,` α`, A, lda, B, ldb,` β`, C, ldc)`

C call	`cblas_chemm(CblasRowMajor, CblasRight, CblasUpper, m, n,` α`, A, lda, B, ldb,` β`, C, ldc)`
C op	$C \leftarrow \alpha BA + \beta C$
F77 op	$C^T \leftarrow \alpha A^T B^T + \beta C^T$
F77 call	`CHEMM('L', 'L', n, m,` α`, A, lda, B, ldb,` β`, C, ldc)`

SYRK

C call	`cblas_csyrk(CblasRowMajor, CblasUpper, CblasNoTrans, n, k,` α`, A, lda,` β`, C, ldc)`
C op	$C \leftarrow \alpha AA^T + \beta C$
F77 op	$C^T \leftarrow \alpha AA^T + \beta C^T$
F77 call	`CSYRK('L', 'T', n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`

C call	`cblas_csyrk(CblasRowMajor, CblasUpper, CblasTrans, n, k,` α`, A, lda,` β`, C, ldc)`
C op	$C \leftarrow \alpha A^T A + \beta C$
F77 op	$C^T \leftarrow \alpha A^T A + \beta C^T$
F77 call	`CSYRK('L', 'N', n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`

In reading the above descriptions, it is important to remember a few things. First, the symmetric matrix is C, and thus we change `UPLO` to accommodate the differing storage of C. `TRANSPOSE` is then varied to handle the storage effects on A.

HERK

C call	`cblas_cherk(CblasRowMajor, CblasUpper, CblasNoTrans, n, k,` α`, A, lda,` β`, C, ldc)`
C op	$C \leftarrow \alpha AA^H + \beta C$
F77 op	$C^T \leftarrow \alpha \overline{A} A^T + \beta C^T$
F77 call	`CHERK('L', 'C', n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`

C call	`cblas_cherk(CblasRowMajor, CblasUpper, CblasConjTrans, n, k,` α`, A, lda,` β`, C, ldc)`
C op	$C \leftarrow \alpha A^H A + \beta C$
F77 op	$C^T \leftarrow \alpha A^T \overline{A} + \beta C^T$
F77 call	`CHERK('L', 'N', n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`

SYR2K

C call	`cblas_csyr2k(CblasRowMajor, CblasUpper, CblasNoTrans, n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`
C op	$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$
F77 op	$C^T \leftarrow \alpha BA^T + \alpha AB^T + \beta C^T = \alpha AB^T + \alpha BA^T + \beta C^T$
F77 call	`CSYR2K('L', 'T', n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`

C call	`cblas_csyr2k(CblasRowMajor, CblasUpper, CblasTrans, n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`
C op	$C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$
F77 op	$C^T \leftarrow \alpha B^T A + \alpha A^T B + \beta C^T = \alpha A^T B + \alpha B^T A + \beta C^T$
F77 call	`CSYR2K('L', 'N', n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`

Note that we once again wind up with an operation that looks the same from C and Fortran 77, saving that the C operations wishes to form C^T, instead of C. So once again we flip the setting of `UPLO` to handle the difference in the storage of C. We then flip the setting of `TRANS` to handle the storage effects for A and B.

HER2K

C call	`cblas_cher2k(CblasRowMajor, CblasUpper, CblasNoTrans, n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`
C op	$C \leftarrow \alpha A B^H + \overline{\alpha} B A^H + \beta C$
F77 op	$C^T \leftarrow \alpha \overline{B} A^T + \overline{\alpha}\,\overline{A} B^T + \beta C^T = \overline{\alpha}\,\overline{A} B^T + \alpha \overline{B} A^T + \beta C^T$
F77 call	`CHER2K('L', 'C', n, k,` $\overline{\alpha}$`, A, lda, B, ldb,` β`, C, ldc)`

C call	`cblas_cher2k(CblasRowMajor, CblasUpper, CblasConjTrans, n, k,` α`, A, lda, B, ldb,` β`, C, ldc)`
C op	$C \leftarrow \alpha A^H B + \overline{\alpha} B^H A + \beta C$
F77 op	$C^T \leftarrow \alpha B^T \overline{A} + \overline{\alpha} A^T \overline{B} + \beta C^T = \overline{\alpha} A^T \overline{B} + \alpha B^T \overline{A} + \beta C^T$
F77 call	`CHER2K('L', 'N', n, k,` $\overline{\alpha}$`, A, lda, B, ldb,` β`, C, ldc)`

TRMM/TRSM

Because of their identical use of the `SIDE`, `UPLO`, and `TRANSA` parameters, TRMM and TRSM share the same general analysis. Remember that A is a triangular matrix, and thus when we handle its storage by flipping `UPLO`, we implicitly change its `TRANS` setting as well. With this in mind, we have:

C call	`cblas_ctrmm(CblasRowMajor, CblasLeft, CblasUpper, CblasNoTrans, diag, m, n,` α`, A, lda, B, ldb)`
C op	$B \leftarrow \alpha A B$
F77 op	$B^T \leftarrow \alpha B^T A^T$
F77 call	`CTRMM('R', 'L', 'N', diag, n, m,` α`, A, lda, B, ldb)`

C call	`cblas_ctrmm(CblasRowMajor, CblasLeft, CblasUpper, CblasTrans, diag, m, n,` α`, A, lda, B, ldb)`
C op	$B \leftarrow \alpha A^T B$
F77 op	$B^T \leftarrow \alpha B^T A$
F77 call	`CTRMM('R', 'L', 'T', diag, n, m,` α`, A, lda, B, ldb)`

C call	`cblas_ctrmm(CblasRowMajor, CblasLeft, CblasUpper, CblasConjTrans, diag, m, n,` α`, A, lda, B, ldb)`
C op	$B \leftarrow \alpha A^H B$
F77 op	$B^T \leftarrow \alpha B^T \overline{A}$
F77 call	`CTRMM('R', 'L', 'C', diag, n, m,` α`, A, lda, B, ldb)`

B.3.3 Banded routines

The above techniques can be used for the banded routines only if a C (row-major) banded array has some sort of meaning when expanded as a Fortran banded array. It turns out that when this is done, you get the transpose of the C array, just as in the dense case.

In Fortran 77, the banded array is an array whose rows correspond to the diagonals of the matrix, and whose columns contain the selected portion of the matrix column. To rephrase this, the diagonals of the matrix are stored in strided storage, and the relevant pieces of the columns of the matrix are stored in contiguous memory. This makes sense: in a column-based algorithm, you will want your columns to be contiguous for efficiency reasons.

In order to ensure our columns are contiguous, we will structure the banded array as shown below. Notice that the first K_U rows of the array store the superdiagonals, appropriately spaced to line up correctly in the column direction with the main diagonal. The last K_L rows contain the subdiagonals.

```
       ------  Super diagonal KU
  -----------  Super diagonal 2
 ------------  Super diagonal 1
-------------  main diagonal (D)
------------   Sub diagonal 1
-----------    Sub diagonal 2
------         Sub diagonal KL
```

If we have a row-major storage, and thus a row-oriented algorithm, we will similarly want our rows to be contiguous in order to ensure efficiency. The storage scheme that is thus dictated is shown below. Notice that the first K_L columns store the subdiagonals, appropriately padded to line up with the main diagonal along rows.

```
 KL   D   KU
      | | | |
    | | | | |
  | | | | | |
| | | | | |
| | | | |
| | | |
```

Now, let us contrast these two storage schemes. Both store the diagonals of the matrix along the non-contiguous dimension of the matrix. The column-major banded array stores the matrix columns along the contiguous dimension, whereas the row-major banded array stores the matrix rows along the contiguous storage.

This gives us our first hint as to what to do: rows stored where columns should be, indicated, in the dense routines, that we needed to set a transpose parameter. We will see that we can do this for the banded routines as well.

We can further note that in the column-major banded array, the first part of the non-contiguous dimension (i.e. the first rows) store superdiagonals, whereas the first part of the non-contiguous dimension of row-major arrays (i.e., the first columns) store the subdiagonals.

We now note that when you transpose a matrix, the superdiagonals of the matrix become the subdiagonals of the matrix transpose (and vice versa).

Along the contiguous dimension, we note that we skip K_U elements before coming to our first entry in a column-major banded array. The same happens in our row-major banded array, except that the skipping factor is K_L.

All this leads to the idea that when we have a row-major banded array, we can consider it as a transpose of the Fortran 77 column-major banded array, where we will swap not only m and n, but also K_U and K_L. An example should help demonstrate this principle. Let us say we have the matrix $A = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \end{bmatrix}$

If we express this entire array in banded form (a fairly dumb thing to do, but good for example purposes), we get $K_U = 3$, $K_L = 1$. In row-major banded storage this becomes: $C_b = \begin{bmatrix} X & 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 & X \end{bmatrix}$

So, we believe this should be the transpose if interpreted as a Fortran 77 banded array. The matrix transpose, and its Fortran 77 banded storage is shown below:

$$A^T = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} \Rightarrow F_b = \begin{bmatrix} X & 2 \\ 1 & 4 \\ 3 & 6 \\ 5 & 8 \\ 7 & X \end{bmatrix}$$

Now we simply note that since C_b is row major, and F_b is column-major, they are actually the same array in memory.

With the idea that row-major banded matrices produce the transpose of the matrix when interpreted as column-major banded matrices, we can use the same analysis for the banded BLAS as we used for the dense BLAS, noting that we must also always swap K_U and K_L.

B.3.4 Packed routines

Packed routines are much simpler than banded. Here we have a triangular, symmetric or Hermitian matrix which is packed so that only the relevant triangle is stored. Thus if we have an upper triangular matrix stored in column-major packed storage, the first element holds the relevant portion of the first column of the matrix, the next two elements hold the relevant portion of the second column, etc.

With an upper triangular matrix stored in row-major packed storage, the first N elements hold the first row of the matrix, the next $N-1$ elements hold the next row, etc.

Thus we see in the Hermitian and symmetric cases, to get a row-major packed array correctly interpreted by Fortran 77, we will simply switch the setting of `UPLO`. This will mean that the rows of the matrix will be read in as the columns, but this is not a problem, as we have seen before. In the symmetric case, since $A = A^T$ the column and rows are the same, so there is obviously no problem. In the Hermitian case, we must be sure that the imaginary component of the diagonal is not used, and it assumed to be zero. However, the diagonal element in a row when our matrix is upper will correspond to the diagonal element in a column when our matrix is called lower, so this is handled as well.

In the triangular cases, we will need to change both `UPLO` and `TRANS`, just as in the dense routines.

With these ideas in mind, the analysis for the dense routines may be used unchanged for packed.

B.4 Fortran 95 BLAS and an Interface to the Legacy BLAS

B.4.1 Introduction

We present a proposed specification for Fortran 95[1] BLAS which can be used as an interface to the legacy BLAS.

The current proposal is implemented as an interface to the Fortran 77 BLAS, but has been designed so that it can also be the prototype for Fortran 95 thin BLAS.

A preliminary version of a Fortran 90 BLAS proposal has been circulated informally (but not very widely) for a number of years, and code which implements that version has been available from NAG Ltd. That proposal was presented at the BLAS Workshop in Knoxville in November 1995 [26]. This proposal takes into account a number of suggestions and comments made on the preliminary version.

This proposal is designed to cover – as far as seems sensible – roughly the same functionality as the original Level 1, 2 and 3 BLAS. Many of the Fortran 77 Level 1 BLAS can be replaced by simple array expressions and assignments in Fortran 95, without loss of convenience or performance (assuming a reasonable degree of optimisation by the compiler). Fortran 95 also allows groups of related Level 2 and Level 3 BLAS to be merged together, each group being covered by a single interface.

B.4.2 Design of the Fortran 95 Interfaces

The proposed design utilizes the following features of the Fortran 95 language.

Generic interfaces: all routines are accessed through *generic* interfaces. A single generic name covers several specific instances whose arguments may differ in the following properties:

- **data type** (real or complex). However, the relevant arguments must be either all real or all complex. (We do not, for example, cater for multiplying a real matrix by a complex matrix, though this functionality could easily be added to the design if there was a need for it.)
- **precision** (or equivalently, kind-value). However, all real or complex arguments must have the same precision. For the interface to the Fortran 77 BLAS we allow single and double precision.
- **rank** Some arguments may either have rank 2 (to store a matrix) or rank 1 (to store a vector).
- **different argument lists** Some of the arguments may not appear in a specific instance. In this case a predefined value or a predefined action is assumed. The following table contains the predefined value or action for these arguments. Some of these arguments are key arguments that will be described below under derived types.

argument	value or action if the argument is not used
alpha	1.0 or (1.0,0.0)
beta	0.0 or (0.0,0.0)
op_x	operate with x
lower	reference upper triangle only
right_side	operate on the left-hand side
unit_diag	non-unit triangular

[1] Fortran 95 is the current ISO Fortran standard

Assumed-shape arrays: all array arguments are *assumed-shape* arrays, which must have the exact shape required to store the corresponding matrix or vector. Hence arguments to specify array-dimensions or problem-dimensions are not required. The routines assume that the supplied arrays have valid and consistent shapes. Zero dimensions (implying empty arrays) are allowed.

Derived types: in the Fortran 77 Level 2 and 3 BLAS, we use character arguments to specify different options for the operation to be performed. In this proposal we suggest instead using dummy arguments whose actual argument must be a named constant of a derived type, which is defined within the BLAS module (and accessible via the BLAS module). All the user needs to do is supply the correct name. Such arguments we call *key arguments*. The next section lists the proposed key arguments. Key arguments do not need to be supplied, in which case a default operation is performed.

B.4.3 Key Arguments

The following table lists the key arguments, the related BLAS named constants and the equivalent F77 BLAS character values.

dummy argument	named constant	meaning	F77 argument
op_x	*not present*	operate with x	TRANSx = 'N'
	blas_trans	operate with x^T	TRANSx = 'T'
	blas_conj	operate with $\bar{x}$	TRANSx = 'C'
	blas_conj_trans	operate with x^H	TRANSx = 'H'
lower	*not present*	reference upper triangle only	UPLO = 'U'
	blas_lower	reference lower triangle only	UPLO = 'L'
right_side	*not present*	operate on the left-hand side	SIDE = 'L'
	blas_right	operate on the right-hand side	SIDE = 'R'
unit_diag	*not present*	non-unit triangular	DIAG = 'N'
	blas_unit_diag	unit triangular	DIAG = 'U'

We could have chosen to use *optional* arguments rather than key arguments, but we chose the latter with a view to a future thin interface because key arguments do not involve any run time checking.

B.4.4 The Fortran 95 Interfaces

In the following subsections each interface is summarized in the form of a *subroutine* statement (or in one case a *function* statement), in which all the potential arguments appear. Arguments that need not be supplied are enclosed in square brackets, for example:

```
subroutine trmm( [alpha,] a, [op_a,] b, [lower,] [right_side,] [unit_diag] )
```

This is followed by a table which lists the different variants of the operation, depending either on the ranks of some of the arguments or on the key arguments. The scalar values *alpha* and *beta* take the defaults given in the following table:

Argument	default value in real case	default value in complex case
alpha	1.0	(1.0,0.0)
beta	0.0	(0.0,0.0)

The following table shows the notation that is used for the values of the key arguments `op_x`, `lower`, `right_side` and `unit_diag`.

Dummy argument	Notation in table	Corresponding named constant
`op_x`	T	`blas_trans`
	C	`blas_conj`
	H	`blas_conj_trans`
`lower`	L	`blas_lower`
`right_side`	R	`blas_right`
`unit_diag`	U	`blas_unit_diag`

Level 1 BLAS

`function nrm2( x )`

Operation: return $\|x\|_2$.

`subroutine swap( x, y )`

Operation: $x \leftrightarrow y$.

`subroutine rot( x, y, c, s )`

Operation: Apply a plane rotation

Note that the Fortran 77 Level 1 BLAS did not contain a complex version, but LAPACK does contain an auxiliary routine `CROT` and we propose to interface to that in the complex case.

`subroutine rotg( a, b, c, s )`

Operation: Generate a plane rotation

Note that, as with `rot` the Fortran 77 Level 1 BLAS did not contain a complex version, but LAPACK does contain an auxiliary routine `CLARTG` and we propose to interface to that in the complex case. **Question:** Should we interface to the LAPACK routine `SLARTG`, rather than `SROTG`, in the real case also?

Level 2 and 3 BLAS Routines with Conventional Storage

By conventional storage we mean storing a matrix in a 2-dimensional array, such that a_{ij} is in the corresponding element of the array.

`subroutine gemm( [alpha,] a, [op_a,] b, [op_b], [beta,] c )`

Rank of `a`	Rank of `b`	Rank of `c`	`op_a`	`op_b`	Operation	F77 BLAS
2	2	2			$C \leftarrow \alpha AB + \beta C$	`_GEMM`
2	2	2		T	$C \leftarrow \alpha AB^T + \beta C$	`_GEMM`
2	2	2		H	$C \leftarrow \alpha AB^H + \beta C$	`_GEMM`
2	2	2	T		$C \leftarrow \alpha A^T B + \beta C$	`_GEMM`
2	2	2	T	T	$C \leftarrow \alpha A^T B + \beta C$	`_GEMM`
2	2	2	H		$C \leftarrow \alpha A^H B + \beta C$	`_GEMM`
2	2	2	H	H	$C \leftarrow \alpha A^H B^H + \beta C$	`_GEMM`
2	1	1			$c \leftarrow \alpha Ab + \beta c$	`_GEMV`
2	1	1	T		$c \leftarrow \alpha A^T b + \beta c$	`_GEMV`
2	1	1	H		$c \leftarrow \alpha A^H b + \beta c$	`_GEMV`
1	1	2			$C \leftarrow \alpha ab^T + \beta C$	`_GER_`
1	1	2		C	$C \leftarrow \alpha ab^H + \beta C$	`_GER_`

Question: Should we allow 'H T' and 'T H' options? Although we have not included them in this table they have been included in the specific procedures (see B.4.8) and in the document for gemm.

```
subroutine symm( [alpha,] a, b, [beta,] c, [lower,] [right_side] )
subroutine hemm( [alpha,] a, b, [beta,] c, [lower,] [right_side] )
```

Rank of `b`	Rank of `c`	`right_side`	Operation	F77 BLAS
2	2		$C \leftarrow \alpha AB + \beta C$	`_SYMM/_HEMM`
2	2	R	$C \leftarrow \alpha BA + \beta C$	`_SYMM/_HEMM`
1	1		$c \leftarrow \alpha Ab + \beta c$	`_SYMV/_HEMV`

A is a symmetric or Hermitian matrix.

```
subroutine syrk( [alpha,] a, [op_a,] [beta,] c, [lower] )
subroutine herk( [alpha,] a, [op_a,] [beta,] c, [lower] )
```

Rank of `a`	`op_a`	Operation	F77 BLAS
2		$C \leftarrow \alpha AA^T + \beta C$	`_SYRK`
2		$C \leftarrow \alpha AA^H + \beta C$	`_HERK`
2	T	$C \leftarrow \alpha A^T A + \beta C$	`_SYRK`
2	H	$C \leftarrow \alpha A^H A + \beta C$	`_HERK`
1		$C \leftarrow \alpha aa^T + \beta C$	`_SYR`
1		$C \leftarrow \alpha aa^H + \beta C$	`_HER`

C is a symmetric or Hermitian matrix.

```
subroutine syr2k( [alpha,] a, [op_a,] b, [beta,] c, [lower] )
subroutine her2k( [alpha,] a, [op_a,] b, [beta,] c, [lower] )
```

Rank of `a`	Rank of `b`	`op_a`	Operation	F77 BLAS
2	2		$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$	`_SYR2K`
2	2		$C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C$	`_HER2K`
2	2	T	$C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$	`_SYR2K`
2	2	H	$C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C$	`_HER2K`
1	1		$C \leftarrow \alpha ab^T + \alpha ba^T + \beta C$	`_SYR2`
1	1		$C \leftarrow \alpha ab^H + \bar{\alpha} ba^H + \beta C$	`_HER2`

C is a symmetric or Hermitian matrix.

```
subroutine trmm( [alpha,] a, [op_a,] b, [lower,] [right_side,] [unit_diag] )
```

Rank of `b`	`op_a`	`right_side`	Operation	F77 BLAS
2			$B \leftarrow \alpha AB$	`_TRMM`
2	T		$B \leftarrow \alpha A^T B$	`_TRMM`
2	H		$B \leftarrow \alpha A^H B$	`_TRMM`
2		R	$B \leftarrow \alpha BA$	`_TRMM`
2	T	R	$B \leftarrow \alpha BA^T$	`_TRMM`
2	H	R	$B \leftarrow \alpha BA^H$	`_TRMM`
1			$b \leftarrow \alpha Ab$	`_TRMV`
1	T		$b \leftarrow \alpha A^T b$	`_TRMV`
1	H		$b \leftarrow \alpha A^H b$	`_TRMV`

A is a triangular matrix.

```
subroutine trsm( [alpha,] a, [op_a,] b, [lower,] [right_side,] [unit_diag] )
```

Rank of b	op_a	right_side	Operation	F77 BLAS
2			$B \leftarrow \alpha A^{-1}B$	_TRSM
2	T		$B \leftarrow \alpha A^{-T}B$	_TRSM
2	H		$B \leftarrow \alpha A^{-H}B$	_TRSM
2		R	$B \leftarrow \alpha BA^{-1}$	_TRSM
2	T	R	$B \leftarrow \alpha BA^{-T}$	_TRSM
2	H	R	$B \leftarrow \alpha BA^{-H}$	_TRSM
1			$b \leftarrow \alpha A^{-1}b$	_TRSV
1	T		$b \leftarrow \alpha A^{-T}b$	_TRSV
1	H		$b \leftarrow \alpha A^{-H}b$	_TRSV

A is a triangular matrix.

Level 2 BLAS Routines with Packed Storage

By *packed* storage, we mean storing the upper or lower triangle of a symmetric or triangular matrix in a 1-dimensional array.

```
subroutine spmv( [alpha,] a, b, [beta,] c, [lower] )
subroutine hpmv( [alpha,] a, b, [beta,] c, [lower] )
```

Operation	F77 BLAS
$c \leftarrow \alpha Ab + \beta c$	_SPMV/_HPMV

A is a symmetric or Hermitian matrix.

```
subroutine spr( [alpha,] a, [beta,] c, [lower] )
subroutine hpr( [alpha,] a, [beta,] c, [lower] )
```

Operation	F77 BLAS
$C \leftarrow \alpha aa^T + \beta C$	_SPR
$C \leftarrow \alpha aa^H + \beta C$	_HPR

C is a symmetric or Hermitian matrix.

```
subroutine spr2( [alpha,] a, b, [beta,] c, [lower] )
subroutine hpr2( [alpha,] a, b, [beta,] c, [lower] )
```

Operation	F77 BLAS
$C \leftarrow \alpha ab^T + \alpha ba^T + \beta C$	_SPR2
$C \leftarrow \alpha ab^H + \bar{\alpha} ba^H + \beta C$	_HPR2

C is a symmetric or Hermitian matrix.

```
subroutine tpmv( [alpha,] a, [op_a,] b, [lower,] [unit_diag] )
```

op_a	Operation	F77 BLAS
	$b \leftarrow \alpha A b$	_TPMV
T	$b \leftarrow \alpha A^T b$	_TPMV
H	$b \leftarrow \alpha A^H b$	_TPMV

A is a triangular matrix.

```
subroutine tpsv( [alpha,] a, [op_a,] b, [lower,] [unit_diag] )
```

op_a	Operation	F77 BLAS
	$b \leftarrow \alpha A^{-1} b$	_TPSV
T	$b \leftarrow \alpha A^{-T} b$	_TPSV
H	$b \leftarrow \alpha A^{-H} b$	_TPSV

A is a triangular matrix.

Level 2 BLAS Routines for Band Matrices

```
subroutine gbmv( [alpha,] a, [op_a,] b, [beta,] c, kd )
```

op_a	Operation	F77 BLAS
	$C \leftarrow \alpha A b + \beta c$	_GBMV
T	$C \leftarrow \alpha A^T b + \beta c$	_GBMV
H	$C \leftarrow \alpha A^H b + \beta c$	_GBMV

A is a general band matrix with `kd` superdiagonals supplied.

```
subroutine sbmv( [alpha,] a, b, [beta,] c, [lower] )
subroutine hbmv( [alpha,] a, b, [beta,] c, [lower] )
```

Operation	F77 BLAS
$c \leftarrow \alpha A b + \beta c$	_SBMV/_HBMV

A is a symmetric or Hermitian band matrix.

```
subroutine tbmv( [alpha,] a, [op_a,] , [lower,] [unit_diag] )
```

op_a	Operation	F77 BLAS
	$b \leftarrow \alpha A b$	_TBMV
T	$b \leftarrow \alpha A^T b$	_TBMV
H	$b \leftarrow \alpha A^H b$	_TBMV

A is a triangular band matrix.

```
subroutine tbsv( [alpha,] a, [op_a,] , [lower,] [unit_diag] )
```

op_a	Operation	F77 BLAS
	$b \leftarrow \alpha A b$	_TBSV
T	$b \leftarrow \alpha A^{-T} b$	_TBSV
H	$b \leftarrow \alpha A^{-H} b$	_TBSV

A is a triangular band matrix.

B.4.5 Error Checking

We propose that an additional argument, `info`, is allowed to be added at the end of the argument list of all the routines and, when `info` is present, then the Fortran 95 BLAS perform checks on their arguments. So for example the full argument list for `gemm` becomes:

```
subroutine gemm( [alpha,] a, [op_a,] b, [op_b], [beta,] c, [info] )
```

We propose that the Fortran 95 BLAS perform the following checks on their arguments:

invalid values: as in the Fortran 77 BLAS, but fewer checks need to be performed in Fortran 95 because there are fewer arguments, indeed only the argument `kd` in the band matrix routines needs to be checked.

invalid shape (of an assumed-shape array): for example, if an array which is supposed to hold a symmetric matrix in conventional storage is not square.

inconsistent shapes (of two or more assumed-shape arrays): for example, if the two vector arguments of `swap` do not have the same length.

This proposal doubles the number of specific routies needed, but the checking routine needs only to do the checks and then call the non-checking routine.

B.4.6 Some Questions

1. Should we allow the value `blas_conj_trans` of the key argument `op_x` to be used in the real case? The equivalent facility is available in the Fortran 77 BLAS, but here it would mean supplying the additional specific instances of the routines. (These are trivial to produce, but it does add to the already quite large number of routines).

2. Should we provide Level 2 versions for complex symmetric matrices. These are not provided for in the Fortran 77 BLAS, but we could readily provide them by calling the Level 3 Fortran 77 BLAS. It was something of an oversight not to have originally provided them.

3. Should we provide complex facilities for plane rotations by interfacing to LAPACK routines? Again this seems to have been something of an oversight in the original Fortran 77 BLAS. If we do provide this facility, the additional question is whether we use `SROTG` or `SLARTG`?

4. Should we allow conjugation as an option to more of the routines?

5. Should we allow operations such as $C \leftarrow \alpha A^T B^H + \beta C$ in `gemm`?

6. Should we combine the `sy` and `he` (and `sp` and `hp`) interfaces into one generic interface by allowing a key argument such as blas_comp_sym? When present this argument would signify a complex symmetric matrix rather than a complex Hermitian matrix. A problem with this suggestion is the choice of name for the generic routine; for example, should we use `syrk` or `herk`. Through the renaming facility of Fortran 95 we could allow both names.

7. Naturally other groupings of generic interfaces are possible. Have we chosen a reasonably natural grouping? For example, we could retain more closely the Level 2 and Level 3 structure of the Fortran 77 BLAS, rather than combining some of the Level 2 and Level 3 operations, which would probably make the documentation of the routines easier and clearer. To illustrate this, we include documentation for two versions of `gemm`.

B.4.7 A Fortran 95 Thin Interface

We believe that the specification proposed here can form the basis for a thin interface to the Fortran 95 BLAS, since the proposed specification requires no runtime decisions to be made and each specific instance of the generic interface performs a specific task.

A debug version of the routines can readily be provided and accessed by supplying the argument `info`. This version would of course perform run time checks, but once debugging is complete the argument `info` can then be removed.

B.4.8 Appendix: Specific procedures for gemm

This appendix contains a list of the specific procedures for the generic procedure gemm.

```
!
! 36 procedures each calls the F77 BLAS subroutine ZGEMM
!  a, b and c are rank-2
!
!            alpha a      op_a       b     op_b        beta c   operation
!
! zgemm_301 (alpha,a,blas_conj_trans,b,blas_conj_trans,beta,c)  C < alpha A(H) B(H) + beta C
! zgemm_302 (alpha,a,blas_conj_trans,b,blas_conj_trans,     c)  C < alpha A(H) B(H) + C
! zgemm_303 (alpha,a,blas_conj_trans,b,blas_trans,     beta,c)  C < alpha A(H) B(T) + beta C
! zgemm_304 (alpha,a,blas_conj_trans,b,blas_trans,          c)  C < alpha A(H) B(T) + C
! zgemm_305 (alpha,a,blas_conj_trans,b,                beta,c)  C < alpha A(H) B + beta C
! zgemm_306 (alpha,a,blas_conj_trans,b,                     c)  C < alpha A(H) B + C
! zgemm_307 (alpha,a,blas_trans,     b,blas_conj_trans,beta,c)  C < alpha A(T) B(H) + beta C
! zgemm_308 (alpha,a,blas_trans,     b,blas_conj_trans,     c)  C < alpha A(T) B(H) + C
! zgemm_309 (alpha,a,blas_trans,     b,blas_trans,     beta,c)  C < alpha A(T) B(T) + beta C
! zgemm_310 (alpha,a,blas_trans,     b,blas_trans,          c)  C < alpha A(T) B(T) + C
! zgemm_311 (alpha,a,blas_trans,     b,                beta,c)  C < alpha A(T) B + beta C
! zgemm_312 (alpha,a,blas_trans,     b,                     c)  C < alpha A(T) B + C
! zgemm_313 (alpha,a,                b,blas_conj_trans,beta,c)  C < alpha A B(H) + beta C
! zgemm_314 (alpha,a,                b,blas_conj_trans,     c)  C < alpha A B(H) + C
! zgemm_315 (alpha,a,                b,blas_trans,     beta,c)  C < alpha A B(T) + beta C
! zgemm_316 (alpha,a,                b,blas_trans,          c)  C < alpha A B(T) + C
! zgemm_317 (alpha,a,                b,                beta,c)  C < alpha A B + beta C
! zgemm_318 (alpha,a,                b,                     c)  C < alpha A B + C
! zgemm_319 (      a,blas_conj_trans,b,blas_conj_trans,beta,c)  C < A(H) B(H) + beta C
! zgemm_320 (      a,blas_conj_trans,b,blas_conj_trans,     c)  C < A(H) B(H) + C
! zgemm_321 (      a,blas_conj_trans,b,blas_trans,     beta,c)  C < A(H) B(T) + beta C
! zgemm_322 (      a,blas_conj_trans,b,blas_trans,          c)  C < A(H) B(T) + C
! zgemm_323 (      a,blas_conj_trans,b,                beta,c)  C < A(H) B + beta C
! zgemm_324 (      a,blas_conj_trans,b,                     c)  C < A(H) B + C
! zgemm_325 (      a,blas_trans,     b,blas_conj_trans,beta,c)  C < A(T) B(H) + beta C
! zgemm_326 (      a,blas_trans,     b,blas_conj_trans,     c)  C < A(T) B(H) + C
! zgemm_327 (      a,blas_trans,     b,blas_trans,     beta,c)  C < A(T) B(T) + beta C
! zgemm_328 (      a,blas_trans,     b,blas_trans,          c)  C < A(T) B(T) + C
! zgemm_329 (      a,blas_trans,     b,                beta,c)  C < A(T) B + beta C
! zgemm_330 (      a,blas_trans,     b,                     c)  C < A(T) B + C
! zgemm_331 (      a,                b,blas_conj_trans,beta,c)  C < A B(H) + beta C
! zgemm_332 (      a,                b,blas_conj_trans,     c)  C < A B(H) + C
! zgemm_333 (      a,                b,blas_trans,     beta,c)  C < A B(T) + beta C
! zgemm_334 (      a,                b,blas_trans,          c)  C < A B(T) + C
! zgemm_335 (      a,                b,                beta,c)  C < A B + beta C
! zgemm_336 (      a,                b,                     c)  C < A B + C
!
!  Here we allow the H T  and  T H  options.
!
!
! 12 procedures each calls the F77 BLAS subroutine ZGEMV
!  a is rank-2, and b and c are rank-1
!
!            alpha a      op_a       b beta c   operation
!
! zgemv_201 (alpha,a,blas_conj_trans,b,beta,c)  c < alpha A(H) b + beta c
! zgemv_202 (alpha,a,blas_conj_trans,b,     c)  c < alpha A(H) b + c
! zgemv_203 (alpha,a,blas_trans,     b,beta,c)  c < alpha A(T) b + beta c
! zgemv_204 (alpha,a,blas_trans,     b,     c)  c < alpha A(T) b + c
! zgemv_205 (alpha,a,                b,beta,c)  c < alpha A b + beta c
! zgemv_206 (alpha,a,                b,     c)  c < alpha A b + c
! zgemv_207 (      a,blas_conj_trans,b,beta,c)  c < A(H) b + beta c
! zgemv_208 (      a,blas_conj_trans,b,     c)  c < A(H) b + c
! zgemv_209 (      a,blas_trans,     b,beta,c)  c < A(T) b + beta c
! zgemv_200 (      a,blas_trans,     b,     c)  c < A(T) b + c
! zgemv_211 (      a,                b,beta,c)  c < A b + beta c
! zgemv_212 (      a,                b,     c)  c < A b + c
!
```

```
!
! 4 procedures each calls the F77 BLAS subroutine ZGERU
!  c is rank-2, and a and b are rank-1
!
!           alpha a b beta c   operation
!
! zgeru_201 (alpha,a,b,beta,c)  C < alpha a b(T) + beta C
! zgeru_202 (alpha,a,b,     c)  C < alpha a b(T) + C
! zgeru_203 (      a,b,beta,c)  C < a b(T) + beta C
! zgeru_204 (      a,b,     c)  C < a b(T) + C
!
!
! 4 procedures each calls the F77 BLAS subroutine ZGERC
!  c is rank-2, and a and b are rank-1
!
!           alpha a b    op_b   beta c   operation
!
! zgerc_201 (alpha,a,b,blas_conj,beta,c)  C < alpha a b(H) + beta C
! zgerc_202 (alpha,a,b,blas_conj,     c)  C < alpha a b(H) + C
! zgerc_203 (      a,b,blas_conj,beta,c)  C < a b(H) + beta C
! zgerc_204 (      a,b,blas_conj,     c)  C < a b(H) + C
!
!
!
! 36 procedures each calls the F77 BLAS subroutine DGEMM
!  a, b and c are rank-2
! (Only 16 procedures are needed, but we allow for  op_? = blas_conj_trans
! for similarity with the complex case).
!
!           alpha a      op_a       b     op_b        beta c   operation
!
! dgemm_301 (alpha,a,blas_conj_trans,b,blas_conj_trans,beta,c)  C < alpha A(H) B(H) + beta C
! dgemm_302 (alpha,a,blas_conj_trans,b,blas_conj_trans,     c)  C < alpha A(H) B(H) + C
! dgemm_303 (alpha,a,blas_conj_trans,b,blas_trans,     beta,c)  C < alpha A(H) B(T) + beta C
! ...
! ...
! dgemm_334 (      a,               b,blas_trans,          c)  C < A B(T) + C
! dgemm_335 (      a,               b,               beta,c)  C < A B + beta C
! dgemm_336 (      a,               b,                    c)  C < A B + C
!
!
! 12 procedures each calls the F77 BLAS subroutine DGEMV
!  a is rank-2, and b and c are rank-1
! (Only 8 procedures are needed, but we allow for  op_a = blas_conj_trans
! for similarity with the complex case).
!
!           alpha a      op_a       b beta c   operation
!
! dgemv_201 (alpha,a,blas_conj_trans,b,beta,c)  c < alpha A(H) b + beta c
! dgemv_202 (alpha,a,blas_conj_trans,b,     c)  c < alpha A(H) b + c
! ...
! dgemv_211 (      a,               b,beta,c)  c < A b + beta c
! dgemv_212 (      a,               b,     c)  c < A b + c
!
!
! 8 procedures each calls the F77 BLAS subroutine DGER
!  c is rank-2, and a and b are rank-1
! (Only 4 procedures are needed, but we allow for  op_b = blas_conj
! for similarity with the complex case).
!
!           alpha a b     op_b  beta c   operation
!
! dger_201  (alpha,a,b,          beta,c)  C < alpha a b(T) + beta C
! dger_202  (alpha,a,b,               c)  C < alpha a b(T) + C
! dger_203  (      a,b,          beta,c)  C < a b(T) + beta C
! dger_204  (      a,b,               c)  C < a b(T) + C
! dger_205  (alpha a,b,blas_conj,beta,c)  C < alpha a b(H) + beta C
! dger_206  (alpha a,b,blas_conj,     c)  C < alpha a b(H) + C
```

```
! dger_207  (      a,b,blas_conj,beta,c)  C < a b(H) + beta C
! dger_208  (      a,b,blas_conj,     c)  C < a b(H) + C
```

B.5 gemm – Current Version

Procedure gemm

B.5.1 Description

`gemm` is a generic procedure which performs one of following operations:

rank of `a`	rank of `b`	rank of `c`	`op_a`	`op_b`	operation	F77 BLAS
2	2	2			$C \leftarrow \alpha AB + \beta C$	**_GEMM**
2	2	2		T	$C \leftarrow \alpha AB^T + \beta C$	**_GEMM**
2	2	2		H	$C \leftarrow \alpha AB^H + \beta C$	**_GEMM**
2	2	2	T		$C \leftarrow \alpha A^T B + \beta C$	**_GEMM**
2	2	2	T	T	$C \leftarrow \alpha A^T B^T + \beta C$	**_GEMM**
2	2	2	T	H	$C \leftarrow \alpha A^T B^H + \beta C$	**_GEMM**
2	2	2	H		$C \leftarrow \alpha A^H B + \beta C$	**_GEMM**
2	2	2	H	T	$C \leftarrow \alpha A^H B^T + \beta C$	**_GEMM**
2	2	2	H	H	$C \leftarrow \alpha A^H B^H + \beta C$	**_GEMM**
2	1	1			$c \leftarrow \alpha Ab + \beta c$	**_GEMV**
2	1	1	T		$c \leftarrow \alpha A^T b + \beta c$	**_GEMV**
2	1	1	H		$c \leftarrow \alpha A^H b + \beta c$	**_GEMV**
1	1	2			$C \leftarrow \alpha ab^T + \beta C$	**_GER_**
1	1	2		C	$C \leftarrow \alpha ab^H + \beta C$	**_GER_**

(If A is real, then $A^H = A^T$.)

Do we need the H T and T H options?
We allow the H T and T H options to simplify the documentation of `op_a` and `op_b`.

B.5.2 Usage

```
CALL gemm([alpha,] a, [op_a,] b, [op_b,] [beta,] c)
```

One or more of the arguments in square brakets can be dropped. The order of the supplied arguments must remain unchanged.

Interfaces

Distinct interfaces are provided for each of the combinations of the following cases:

Real / complex data

Real data: `alpha`, `a`, `b`, `beta` and `c` are of type real(kind=wp).

Complex data: `alpha`, `a`, `b`, `beta` and `c` are of type complex(kind=wp).

different ranks

f77_gemm: `a`, `b` and `c` are rank-2 arrays.

f77_gemv: `a` is a rank-2 array while `b` and `c` are rank-1 arrays.

f77_ger: `c` is a rank-2 array while `a` and `b` are rank-1 arrays.

B.5.3 Arguments

All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as 'x(n)' is used in the argument descriptions to specify that the array `x` must have exactly n elements.

The procedure derives the values of the following problem parameters from the shape of the supplied arrays.

m — the first dimension of `c`, if `c` is rank-2 (m =`SIZE(c,1)`), or the size of `c` if it is rank-1 (m =`SIZE(c)`)

n — the second dimension of `c` if it is rank-2 (n =`SIZE(c,2)`)

k — the intermediate dimension

Mandatory arguments

One or more of the arguments `alpha`, `op_a`, `op_b` and `beta` can be dropped. The order of the supplied arguments must remain unchanged.

alpha — real(kind=*wp*)/ complex(kind=*wp*), intent(in)

Input: the value of α if different from one.

Note: if α is exactly one, you need not supply this argument.

a(m) / **a**(p, q) — real(kind=*wp*)/ complex(kind=*wp*), intent(in)

Input: the matrix A or vector a.
If `a` is rank-2 then:

if `op_a` is not supplied, the shape of `a` must be (m, k);

if `op_a` is supplied, the shape of `a` must be (k, m).

op_a — a "key" argument, intent(in)

Input: if `op_a` is supplied, it specifies whether the operation involves the transpose A^T or its conjugate-transpose A^H ($= A^T$ if A is real). In this case `op_a` must have one of the following values (which are named constants, each of a different derived type, defined by the BLAS, and accessible from the module `blas`):

`blas_trans`: if the operation involves the transpose A^T rather than the matrix A;

`blas_conj_trans`: if the operation involves the conjugate-transpose A^H rather than the matrix A.

For further explanation of "key" arguments, see the ????.

Note: for real matrices, `blas_conj_trans` is equivalent to `blas_trans`.

Constraints: `op_a` *must not* be supplied if `a` is rank-1 or if the operation does not involve the transpose or the conjugate-transpose of A.

b(r) / **b**(r, s) — real(kind=*wp*)/ complex(kind=*wp*), intent(in)

Input: the matrix B or vector b.
If `b` is rank-1 then:

if `a` is rank-1, the shape of `b` must be (m);

if `a` is rank-2, the shape of `b` must be `SIZE(op_a(a),2)`.

If `b` is rank-2 then:

if `op_b` is not supplied, the shape of `b` must be (k, n);

if `op_b` is supplied, the shape of `b` must be (n, k).

op_b — a "key" argument, intent(in)

Input: if `op_b` is supplied and `b` is rank-2, it specifies whether the operation involves the transpose B^T or its conjugate-transpose B^H ($= B^T$ if B is real). In this case `op_b` must have one of the following values (which are named constants, each of a different derived type, defined by the BLAS, and accessible from the module `blas`):

`blas_trans`: if the operation involves the transpose B^T rather than the matrix B;

`blas_conj_trans`: if the operation involves the conjugate-transpose B^H rather than the matrix B.

If `op_b` is supplied and `b` is rank-1, it specifies that the operation involves the conjugate of b^T (b^H) rather than b^T. In this case `op_b` must be `blas_conj` (which is a named constant of a derived type, defined by the BLAS, and accessible from the module `blas`).

For further explanation of "key" arguments, see the ????.

Note: for real matrices, `blas_conj_trans` is equivalent to `blas_trans`. For real arrays `blas_conj` does not have any effect.

Constraints: `op_b` *must not* be supplied if the operation does not involve the conjugate of b, the transpose of B or the conjugate-transpose of B.

beta — real(kind=*wp*)/ complex(kind=*wp*), intent(in)

Input: the value of β if different from zero.

Note: if β is exactly zero, you need not supply this argument.

c(m) / **c**(m, n) — real(kind=*wp*)/ complex(kind=*wp*), intent(inout)

Input: the matrix C or vector c. If `beta` is not supplied `c` need not be initialized.

Output: the matrix C or vector c after applying the operation.

B.5.4 Examples of usage

One or more of the arguments alpha, op_a, op_b and beta can be dropped. The order of the supplied arguments must remain unchanged.

To perform $C \leftarrow \alpha AB^H$ use the call:

```
CALL gemm (alpha,a,b,blas_conj_trans,c)
```

To perform $C \leftarrow AB + \beta C$ use the call:

```
CALL gemm (a,b,beta,c)
```

To perform $C \leftarrow \alpha AB + \beta C$ use the call:

```
CALL gemm (alpha,a,b,beta,c)
```

To perform $C \leftarrow A^T B^T + \beta C$ use the call:

```
CALL gemm (a,blas_trans,b,blas_trans,beta,c)
```

To perform $c \leftarrow \alpha A^T b + \beta c$ use the call:

```
CALL gemm (alpha,a,blas_trans,b,beta,c)
```

To perform $C \leftarrow ab^H + \beta C$ use the call:

```
CALL gemm (a,b,blas_conj,beta,c)
```

B.6 gemm – Alternative Version

Procedure gemm

B.6.1 Description

`gemm` is a generic procedure which performs one of following operations:

op_a	op_b	operation	F77 BLAS
		$C \leftarrow \alpha AB + \beta C$	_GEMM
	T	$C \leftarrow \alpha AB^T + \beta C$	_GEMM
	H	$C \leftarrow \alpha AB^H + \beta C$	_GEMM
T		$C \leftarrow \alpha A^T B + \beta C$	_GEMM
T	T	$C \leftarrow \alpha A^T B^T + \beta C$	_GEMM
T	H	$C \leftarrow \alpha A^T B^H + \beta C$	_GEMM
H		$C \leftarrow \alpha A^H B + \beta C$	_GEMM
H	T	$C \leftarrow \alpha A^H B^T + \beta C$	_GEMM
H	H	$C \leftarrow \alpha A^H B^H + \beta C$	_GEMM

(If A is real, then $A^H = A^T$.)

Do we need the H T and T H options?
We allow the H T and T H options to simplify the documentation of `op_a` and `op_b`.

B.6.2 Usage

```
CALL gemm([alpha,] a, [op_a,] b, [op_b,] [beta,] c)
```

One or more of the arguments in square brakets can be dropped. The order of the supplied arguments must remain unchanged.

Interfaces

Distinct interfaces are provided for each of the combinations of the following cases:

Real / complex data

Real data: `alpha`, `a`, `b`, `beta` and `c` are of type real(kind=wp).

Complex data: `alpha`, `a`, `b`, `beta` and `c` are of type complex(kind=wp).

B.6.3 Arguments

All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as 'x(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

The procedure derives the values of the following problem parameters from the shape of the supplied arrays.

m — the first dimension of `c` (m =`SIZE(c,1)`)

n — the second dimension of `c` (n =`SIZE(c,2)`)

k — the intermediate dimension

Mandatory arguments

One or more of the arguments `alpha`, `op_a`, `op_b` and `beta` can be dropped. The order of the supplied arguments must remain unchanged.

alpha — real(kind=wp)/ complex(kind=wp), intent(in)

Input: the value of α if different from one.

Note: if α is exactly one, you need not supply this argument.

a(m) / **a**(p, q) — real(kind=wp)/ complex(kind=wp), intent(in)

Input: the matrix A.

If `op_a` is not supplied, the shape of `a` must be (m, k);

if `op_a` is supplied, the shape of `a` must be (k, m).

op_a — a "key" argument, intent(in)

Input: if `op_a` is supplied, it specifies whether the operation involves the transpose A^T or its conjugate-transpose A^H ($= A^T$ if A is real). In this case `op_a` must have one of the following values (which are named constants, each of a different derived type, defined by the BLAS, and accessible from the module `blas`):

`blas_trans`: if the operation involves the transpose A^T rather than the matrix A;

`blas_conj_trans`: if the operation involves the conjugate-transpose A^H rather than the matrix A.

For further explanation of "key" arguments, see the ????.

Note: if the operation does not involve the transpose or the conjugate-transpose of A, you *must not* supply this argument.

Note: for real matrices, `blas_conj_trans` is equivalent to `blas_trans`.

b(r) / **b**(r, s) — real(kind=wp)/ complex(kind=wp), intent(in)

Input: the matrix B.

If `op_b` is not supplied, the shape of `b` must be (k, n);

if `op_b` is supplied, the shape of `b` must be (n, k).

op_b — a "key" argument, intent(in)

Input: if `op_b` is supplied, it specifies whether the operation involves the transpose B^T or its conjugate-transpose B^H ($= B^T$ if B is real). In this case `op_b` must have one of the following values (which are named constants, each of a different derived type, defined by the BLAS, and accessible from the module `blas`):

`blas_trans`: if the operation involves the transpose B^T rather than the matrix B;

`blas_conj_trans`: if the operation involves the conjugate-transpose B^H rather than the matrix B.

For further explanation of "key" arguments, see the ????.

Note: if the operation does not involve the transpose or the conjugate-transpose of B, you *must not* supply this argument.

Note: for real matrices, `blas_conj_trans` is equivalent to `blas_trans`.

beta — real(kind=wp)/ complex(kind=wp), intent(in)
Input: the value of β if different from zero.
Note: if β is exactly zero, you need not supply this argument.

c(m) / **c**(m,n) — real(kind=wp)/ complex(kind=wp), intent(inout)
Input: the matrix C. If `beta` is not supplied `c` need not be initialized.
Output: the matrix C after applying the operation.

B.6.4 Examples of usage

One or more of the arguments `alpha`, `op_a`, `op_b` and `beta` can be dropped. The order of the supplied arguments must remain unchanged.

To perform $C \leftarrow \alpha AB^H$ use the call:

```
CALL gemm (alpha,a,b,blas_conj_trans,c)
```

To perform $C \leftarrow AB + \beta C$ use the call:

```
CALL gemm (a,b,beta,c)
```

To perform $C \leftarrow \alpha AB + \beta C$ use the call:

```
CALL gemm (alpha,a,b,beta,c)
```

To perform $C \leftarrow A^T B^T + \beta C$ use the call:

```
CALL gemm (a,blas_trans,b,blas_trans,beta,c)
```

Bibliography

[1] *The EM Symmetric Sparse Supernodal Format.* Elegant Mathematics, Inc., Bothell, Washington, 1993.

[2] *The EM Unsymmetric Sparse Supernodal Format.* Elegant Mathematics, Inc., Bothell, Washington, 1993.

[3] M. Aboelaze, N. Chrisochoides, and E. Houstis. The Parallelization of Level 2 and 3 BLAS Operations on Distributed Memory Machines. Technical Report CSD-TR-91-007, Purdue University, West Lafayette, IN, 1991.

[4] R. Agarwal, S. Balle, F. Gustavson, M. Joshi, and P. Palkar. A Three-Dimensional Approach to Parallel Matrix Multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[5] R. Agarwal, F. Gustavson, and M. Zubair. A High Performance Matrix Multiplication Algorithm on a Distributed-Memory Parallel Computer, Using Overlapped Communication. *IBM Journal of Research and Development*, 38(6):673–681, 1994.

[6] E. Anderson, Z. Bai, C. H. Bischof, J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. C. Sorensen. *LAPACK Users' Guide.* SIAM, Philadelphia, PA, USA, 2nd edition, 1995. (Also available in Japanese, published by Maruzen, Tokyo, translated by Dr Oguni).

[7] P. Bangalore. The Data-Distribution-Independent Approach to Scalable Parallel Libraries. Master's thesis, Mississippi State University, 1995.

[8] R. E. Bank and D. J. Rose. Marching algorithms for elliptic boundary value problems. the constant coefficient case. *SIAM Journal of Numerical Analysis*, 14:792–829, 1977.

[9] Randolph E. Bank and Craig C. Douglas. Sparse matrix multiplication package (SMMP). *Advances in Computational Mathematics*, 1, 1993.

[10] B. Bialecki and G. Fairweather. Matrix decomposition algorithms in orthogonal spline collocation for separable elliptic boundary value problems. *SIAM J. Sci. Comput.*, 16:330–347, 1995.

[11] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proceedings of the International Conference on Supercomputing.* ACM SIGARC, 1997. Vienna, Austria.

[12] R. Bisseling and J. van der Vorst. Parallel Triangular System Solving on a mesh network of Transputers. *SIAM Journal on Scientific and Statistical Computing*, 12:787–799, 1991.

[13] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.

[14] R. Brent and P. Strazdins. Implementation of BLAS Level 3 and LINPACK Benchmark on the AP1000. *Fujitsu Scientific and Technical Journal*, 5(1):61–70, 1993.

[15] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. In J. Dongarra, K. Masden, and J. Waśniewski, editors, *Applied Parallel Computing*, pages 107–114. Springer Verlag, 1995. (also LAPACK Working Note No.100).

[16] J. Choi, J. Dongarra, and D. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994. (also LAPACK Working Note No.57).

[17] J. Choi, J. Dongarra, and D. Walker. PB-BLAS: A Set of Parallel Block Basic Linear Algebra Subroutines. *Concurrency: Practice and Experience*, 8(7):517–535, 1996.

[18] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, and R. van de Geijn. Parallel Implementation of BLAS: General Techniques for Level 3 BLAS. Technical Report TR95-49, Department of Computer Sciences, UT-Austin, 1995. Submitted to Concurrency: Practice and Experience.

[19] J. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination, February 1997. To appear in SIAM J. Matrix Anal. Appl., special issue on Sparse and Structured Matrix Computations and Their Applications (Also LAPACK Working Note 124).

[20] J. Demmel and X. Li. Faster numerical algorithms via exception handling. *IEEE Trans. Comput.*, 43:983–992, 1994.

[21] D. S. Dodson. Corrigendum: Remark on "Algorithm 539: Basic Linear Algebra Subroutines for FORTRAN usage". *ACM Trans. Math. Software*, 9:140, 1983. (See also [39] and [22]).

[22] D. S. Dodson and R. G. Grimes. Remark on algorithm 539: Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Software*, 8:403–404, 1982. (See also [39] and [21]).

[23] D. S. Dodson, R. G. Grimes, and J. G. Lewis. Sparse extensions to the FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 17:253–272, 1991. (Algorithm 692).

[24] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–28, 1990. (Algorithm 679).

[25] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 14:1–32, 399, 1988. (Algorithm 656).

[26] J. J. Dongarra, S. Hammarling, and S. Ostrouchov. BLAS technical workshop. LAPACK Working Note No.109. Technical Report CS-95-317, Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301, USA, 1995. URL: http://www.netlib.org/lapack/lawns/.

[27] I. S. Duff, A. M. Erisman, and J. K. Reid. Direct methods for sparse matrices. In *Monographs on Numerical Analysis*. Oxford University Press, New York, 1986.

[28] Iain Duff, Michele Marrone, and Giuseppe Radicati. A proposal for user level sparse BLAS. Technical Report TR/PA/92/85, CERFACS, 1992.

[29] A. Elster. Basic Matrix Subprograms for Distributed Memory Systems. In D. Walker and Q. Stout, editors, *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 311–316. IEEE Press, 1990.

[30] R. Falgout, A. Skjellum, S. Smith, and C. Still. The Multicomputer Toolbox Approach to Concurrent BLAS and LACS. In *Proceedings of the Scalable High Performance Computing Conference SHPCC-92*. IEEE Computer Society Press, 1992.

[31] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3–4), 1994.

[32] G. Fox, S. Otto, and A. Hey. Matrix Algorithms on a Hypercube I: Matrix Multiplication. *Parallel Computing*, 3:17–31, 1987.

[33] P. E. Gill, G. H. Golub, W. Murray, and M. A. Saunders. Methods for modifying matrix factorizations. *Math. Comp.*, 28:505–535, 1974.

[34] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software*, 4(3):250–269, 1978.

[35] M. Heath and C. Romine. Parallel Solution Triangular Systems on Distributed Memory Multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 9:558–588, 1988.

[36] M. A. Heroux. A proposal for a sparse BLAS toolkit. Technical Report TR/PA/92/90, CERFACS, 1992.

[37] S. Huss-Lederman, E. Jacobson, A. Tsao, and G. Zhang. Matrix Multiplication on the Intel Touchstone DELTA. *Concurrency: Practice and Experience*, 6(7):571–594, 1994.

[38] L. Kaufman and D. D. Warner. High-order, fast direct methods for separable elliptical equations. *SIAM Journal of Numerical Analysis*, 21:672–695, 1984.

[39] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Trans. Math. Software*, 5:308–323, 1979. (Algorithm 539. See also [22] and [21]).

[40] G. Li and T. Coleman. A Parallel Triangular Solver for a Distributed-Memory Multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9(3):485–502, 1988.

[41] G. Li and T. Coleman. A New Method for Solving Triangular Systems on Distributed-Memory Message-Passing Multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 10(2):382–396, 1989.

[42] W. Lichtenstein and S. L. Johnsson. Block-Cyclic Dense Linear Algebra. *SIAM Journal on Scientific and Statistical Computing*, 14(6):1259–1288, 1993.

[43] K. Mathur and S. L. Johnsson. Multiplication of Matrices of Arbitrary Shapes on a Data Parallel Computer. *Parallel Computing*, 20:919–951, 1994.

[44] O. McBryan and E. van de Velde. Matrix and Vector Operations on Hypercube Parallel Processors. *Parallel Computing*, 5:117–126, 1987.

[45] R. Pozo. Template numerical toolkit (tnt), 1995. `http://math.nist.gov/tnt`.

[46] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations. Preliminary Version.

[47] P. Strazdins. A High Performance, Portable Distributed BLAS Implementation. In *Proceedings of the Sixth Parallel Computing Workshop*, Fujitsu Parallel Computing Center, 1996.

[48] Thinking Machines Corporation. *CMSSL for Fortran*, 1990.

[49] R. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, Cambridge, Massachusetts, 1997.

[50] R. van de Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. Technical Report UT CS-95-286, LAPACK Working Note No.96, University of Tennessee, 1995.

[51] T. et al. Yang. Distributed memory parallel sparse lu. Ucsb preprint, University of California, Santa Barbara.