Chapter 1

LegacyBLAS

1.1 New routines

1.2 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; also discussed are other solutions to that particular problem, and the reasons those options were not chosen.

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 Fortran77 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 (e.g., gcc).

1.2.1 Naming scheme

The naming scheme consists of taking the Fortran77 routine name, making it lower case, and adding the prefix cblas_. Therefore, the routine DGEMM becomes cblas_dgemm.
CHAPTER 1. LEGACYBLAS

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 Fortran77 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.

1.2.2 Indices

The Fortran77 BLAS return indices in the range \( 1 \leq I \leq N \) (where \( N \) is the number of entries in the dimension in question, and \( I \) is the index), in accordance with Fortran77 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 involves indices is the function \texttt{I_AMAX}. This function is declared to be of type \texttt{CBLAS_INDEX}, which is guaranteed to be an integer type (i.e., no cast is required when assigning to any integer type). \texttt{CBLAS_INDEX} will usually correspond to \texttt{size_t} to ensure any array can be indexed, but implementors might choose the integer type which matches their Fortran77 \texttt{INTEGER}, for instance.

1.2.3 Character arguments

All arguments which were characters in the Fortran77 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 Fortran77 interface are: \texttt{SIDE}, \texttt{UPLO}, \texttt{TRANSPOSE}, and \texttt{DIAG}. This interface adds another such argument to all routines involving two-dimensional arrays, \texttt{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};
```
1.2. C INTERFACE TO LEGACY BLAS

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 Fortran77 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).

1.2.4 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 CBLAS_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
\begin{verbatim}
struct CBLAS_COMPLEX {float v[2];};
\end{verbatim}
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 contiguity, 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.

1.2.5 Return values of complex functions

BLAS routines which return complex values in Fortran77 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 \begin{verbatim}
struct CBLAS_COMPLEX {float r; float i;};
\end{verbatim}

If one is willing to use this structure throughout one's code, then this provides a natural and convenient mechanism. If, however, the programmer
1.2. C INTERFACE TO LEGACY BLAS

has utilized a different structure for complex, this ease of use breaks down. Then, something like the following code fragment is required:

```c
CBLAS_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.

1.2.6 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 (e.g. 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

```c
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 Fortran77 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.
This solution comes after much discussion. It was discovered that C users in general seemed to split into two camps. Those people who have a history of mixing C and Fortran77 (in particular making use of the Fortran77 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 user might be described as the C purists. These users are not interested in overt C/Fortran77 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 Fortran77 BLAS. Recent discussion, 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 1.4 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 2n extra storage, and 3n extra operations.

**Considered methods**

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 Fortran77 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 \( m \times n \) 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:

\[
\text{float **A, **subA;}
\]

\[
\text{subA = malloc(m*sizeof(float*));}
\]

\[
\text{for (k=0; k != m; k++) subA[k] = A[i+k] + j;}
\]

\[
\text{cblas_rout(\ldots \text{subA} \ldots);}
\]

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 subportion to access. Thus you have a call that looks like: \text{cblas_rout(\ldots A, i, j, \ldots);}.

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 Fortran77 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: \text{cblas_rout(\ldots &A[i][j] \ldots);}.

### 1.2.7 C interface include file

The C interface to the BLAS will have a standard include file, called \text{cblas.h}, which minimally contains the definition of the CBLAS types and ANSI/ISO C prototypes for all BLAS routines. The standard \text{cpp} strategy should be employed to ensure this file can be included multiple times without error. Section 1.3 shows a minimal \text{cblas.h}.

### 1.2.8 Error checking

The reference Fortran77 BLAS implementation contains error checking. The C interface is required to supply corresponding error checking. If the Fortran77 code is used to implement the C interface, most of the error checking may be done by Fortran77 code, assuming error reporting is changed to reflect the differing C interface.
1.2.9 Rules for obtaining the C interface from the Fortran77

- The Fortran77 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 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.

- 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.

1.3 cblas.h include file

```c
#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};
```
1.3. CBLAS.H INCLUDE FILE

```c
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_dznm2(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 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,
1.3. CBLAS.H INCLUDE FILE

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_zgbsv(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_ztrsv(const enum CBLAS_ORDER order,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag,
const int N, const void *A, const int lda,
void *X, const int incX);
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_syr(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_spr(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_syr2(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 *Ap);
CHAPTER I. LEGACYBLAS

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_dsyrv(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 *Y, const int incY, double *A);
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, const 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, const 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 *a);
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_zhemv(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 *a,
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_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 *Ap);

/ *
* Prototypes for level 3 BLAS
* 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_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_csymmm(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_csr2k(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_zsymmm(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_zsr2k(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_che2k(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

1.4 Using Fortran77 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 Fortran77 BLAS development 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
1.4. USING FORTRAN77 BLAS TO SUPPORT ROW-MAJOR BLAS OPERATIONS

Fortran77 (column-major) BLAS with triangular matrices coming from C (possibly row-major), we will be reversing the setting of UPL0, 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 UPL0, 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 Fortran77, 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 Fortran77 \( A^T \). 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 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 \texttt{Upper}, \texttt{NoTranspose} becomes \texttt{Lower}, \texttt{Transpose} rule, we will show it for only the upper case. In order to make the notation more concise, let us define $x^c$ to be $\text{conj}(x)$.

### 1.4.1 Level 2 BLAS

#### GEMV

C call \hspace{0.5cm} \texttt{cblas_cgemv} (CblasNoTrans, m, n, $\alpha$, A, lda, x, incx, $\beta$, y, incy)

\hspace{1cm} \texttt{op} \hspace{0.5cm} $y \leftarrow \alpha A x + \beta y$

F77 call \hspace{0.5cm} \texttt{CGEMV} (\texttt{T'}, n, m, $\alpha$, A, lda, x, incx, $\beta$, y, incy)

C call \hspace{0.5cm} \texttt{cblas_cgemv} (CblasTrans, m, n, $\alpha$, A, lda, x, incx, $\beta$, y, incy)

\hspace{1cm} \texttt{op} \hspace{0.5cm} $y \leftarrow \alpha A^T x + \beta y$

F77 call \hspace{0.5cm} \texttt{CGEMV} (\texttt{N'}, n, m, $\alpha$, A, lda, x, incx, $\beta$, y, incy)

C call \hspace{0.5cm} \texttt{cblas_cgemv} (CblasConjTrans, m, n, $\alpha$, A, lda, x, incx, $\beta$, y, incy)

\hspace{1cm} \texttt{op} \hspace{0.5cm} $y \leftarrow \alpha A^H x + \beta y \Rightarrow (y^c \leftarrow \alpha^* A^T x^c + \beta^* y^c)^c$

F77 call \hspace{0.5cm} \texttt{CGEMV} (\texttt{N'}, n, m, $\alpha$, A, lda, xc, 1, $\beta$, 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 form $xc = x^c$. Then, we set $y = y^c$, and make the call. This gives us the conjugate of the answer, so we once again set $y = y^c$. Therefore, we see that to support the conjugate transpose, we will need to allocate an $n$-element vector, and perform $2m + n$ extra operations.

#### HEMV/SYMV

HEMV and SYMV are handled the same. Neither requires extra workspace or operations.
1.4. USING FORTRAN77 BLAS TO SUPPORT ROW-MAJOR BLAS OPERATIONS

C call    cblas_chemv(CblasUpper, n, α, A, lda, x, incx, β, y, incy)
op        y ← αAx + βy
F77 call  CHEMV('L', n, α, A, lda, x, incx, β, y, incy)

C call    cblas_chemv(Lower, n, α, A, lda, x, incx, β, y, incy)
op        y ← αAx + βy
F77 call  CHEMV('U', n, α, A, lda, x, incx, β, y, incy)

TRMV/TRSV

C call    cblas_ctrmv(CblasUpper, CblasNoTrans, diag, n, α, A, lda, x, incx)
op        x ← αAx
F77 call  CTRMV('L', 'T', diag, n, α, A, lda, x, incx)

C call    cblas_ctrmv(CblasUpper, CblasTrans, diag, n, α, A, lda, x, incx)
op        x ← αATx
F77 call  CTRMV('L', 'N', diag, n, α, A, lda, x, incx)

C call    cblas_ctrmv(CblasUpper, CblasConjTrans, diag, n, α, A, lda, x, incx)
op        x ← αAHx ⇒ (x* = α*A†x*)
F77 call  CTRMV('L', 'N', diag, n, α, A, lda, 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 \( \alpha^* \), 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_geru(m, n, α, x, incx, y, incy, A, lda)
C op      \( A ← αxy^T + A \)
F77 op    \( A^T ← αyx^T + A^T \)
F77 call  GERU(m, n, α, y, incy, x, incx, A, lda)

No extra storage or operations are required.
GERC

C call   cblas_cgerc(m, n, α, x, incx, y, incy, A, lda)
C op     \( A \leftarrow αxy^H + A \)
F77 op   \( A^T \leftarrow α(x^Hy^H)^T + A^T = αy^cx^T + A^T \)
F77 call cGERU(n, m, α, 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(CblasUpper, n, α, x, incx, A, lda)
C op     \( A \leftarrow αxx^H + A \)
F77 op   \( A^T \leftarrow αx^c x^T + A^T \)
F77 call CHER('L', n, α, xc, 1, A, lda)

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

HER2

C call   cblas_cher2(CblasUpper, n, α, x, incx, y, incy, A, lda)
C op     \( A \leftarrow αxy^H + y(αx)^H + A \)
F77 op   \( A^T \leftarrow αy^c x^T + α^c x^c y^T + A^T = αy^c (x^c)^H + x^c (αy^c)^H + A^T \)
F77 call CHER2('L', n, α, yc, 1, xc, 1, A, lda)

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

SYR

C call   cblas_ssyr(CblasUpper, n, α, x, incx, A, lda)
C op     \( A \leftarrow αxx^T + A \)
F77 op   \( A^T \leftarrow αx^c x^T + A^T \)
F77 call SSYR('L', n, α, x, incx, A, lda)

No extra storage or operations required.

SYR2

C call   cblas_ssyr2(CblasUpper, n, α, x, incx, y, incy, A, lda)
C op     \( A \leftarrow αy^c y^T + αyx^T + A \)
F77 op   \( A^T \leftarrow αy^c x^T + αyx^c + A^T \)
F77 call SSYR2('L', n, α, y, incy, x, incx, A, lda)
1.4. USING FORTRAN77 BLAS TO SUPPORT ROW-MAJOR BLAS OPERATIONS

No extra storage or operations required.
1.4.2 Level 3 BLAS

GEMM

C call  
cblas__gemm(BlasNoTrans, BlasNoTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op  
C ← αAB + βC
F77 op  
C′ ← αB′A′ + βCT
F77 call  
cgemm('N', 'N', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call  
cblas__gemm(BlasNoTrans, BlasTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op  
C ← αABT + βC
F77 op  
C′ ← αBA′T + βCT
F77 call  
cgemm('N', 'T', n, m, k, α, B, ldb, A, lda, β, C, ldc)

C call  
cblas__gemm(BlasTrans, BlasNoTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op  
C ← αAT B + βC
F77 op  
C′ ← αB′A′ + βCT
F77 call  
cgemm('T', 'N', n, m, k, α, A, lda, B, ldb, β, C, ldc)

C call  
cblas__gemm(BlasTrans, BlasTrans, m, n, k, α, A, lda, B, ldb, β, C, ldc)
C op  
C ← αAT B + βC
F77 op  
C′ ← αB′A′ + βCT
F77 call  
cgemm('T', 'T', n, m, k, α, A, lda, B, ldb, β, C, ldc)
1.4. USING FORTRAN77 BLAS TO SUPPORT ROW-MAJOR BLAS OPERATIONS

SYMM/HEMM

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

C call  
cblas_chemm(Right, CblasUpper, m, n, \alpha, A, lda, B, ldb, \beta, C, ldc)
C op    
C ← \alpha BA + \beta C
F77 op  
C^T ← \alpha A^T B^T + \beta C^T
F77 call 
CHEMM('L', 'L', n, m, \alpha, A, lda, B, ldb, \beta, C, ldc)

SYRK

C call  
cblas_csyrk(CblasUpper, CblasNoTrans, n, k, \alpha, A, lda, \beta, C, ldc)
C op    
C ← \alpha A A^T + \beta C
F77 op  
C^T ← \alpha A^T A^T + \beta C^T
F77 call 
CSYRK('L', 'T', n, k, \alpha, A, lda, B, ldb, \beta, C, ldc)

C call  
cblas_csyrk(CblasUpper, CblasTrans, n, k, \alpha, A, lda, \beta, C, ldc)
C op    
C ← \alpha A^T A + \beta C
F77 op  
C^T ← \alpha A^T A + \beta C^T
F77 call 
CSYRK('L', 'N', n, k, \alpha, A, lda, B, ldb, \beta, 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 UPL0 to accommodate the differing storage of C. TRANS0SE is then varied to handle the storage effects on A.
HERK

C call  
cblas_cherk(CblasUpper, CblasNoTrans, n, k, α, A, lda, β, C, ldc)
C op
C ← αAA^H + βC
F77 op  
C^T ← αA^H A^T + βC^T
F77 call  
HERK('L', 'C', n, k, α, A, lda, B, ldb, β, C, ldc)

C call  
cblas_cherk(CblasUpper, CblasConjTrans, n, k, α, A, lda, β, C, ldc)
C op
C ← αA^H A + βC
F77 op  
C^T ← αA^T A^T + βC^T
F77 call  
HERK('L', 'N', n, k, α, A, lda, B, ldb, β, C, ldc)

SYR2K

C call  
cblas_syr2k(CblasUpper, CblasNoTrans, n, k, α, A, lda, B, ldb, β, C, ldc)
C op
C ← αAB^T + βC
F77 op  
C^T ← αAB^T + αAB^T + βC^T = αAB^T + αAB^T + βC
F77 call  
SYR2K('L', 'T', n, k, α, A, lda, B, ldb, β, C, ldc)

C call  
cblas_syr2k(CblasUpper, CblasTrans, n, k, α, A, lda, B, ldb, β, C, ldc)
C op
C ← αA^T B + αB^T A + βC
F77 op  
C^T ← αB^T A + αA^T B + βC^T = αA^T B + αB^T A + βC
F77 call  
SYR2K('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 Fortran77, saving that the C operations wishes to form C^T instead of C. So once again we flip the setting of UPL0 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_her2k(CblasUpper, CblasNoTrans, n, k, α, A, lda, B, ldb, β, C, ldc)
C op
C ← αAB^H + α^H BA + βC
F77 op  
C^T ← αA^H B^T + α^H A^T B^T + βC^T = αA^H B^T + α^H A^T B^T + βC
F77 call  
HER2K('L', 'C', n, k, α^H, A, lda, B, ldb, β, C, ldc)

C call  
cblas_her2k(CblasUpper, CblasConjTrans, n, k, α, A, lda, B, ldb, β, C, ldc)
C op
C ← αA^H B + α^H B^T A + βC
F77 op  
C^T ← αB^H A^T + αA^T B^T + βC^T = αA^T B^T + αB^T A^T + βC
F77 call  
HER2K('L', 'N', n, k, α^H, A, lda, B, ldb, β, C, ldc)
1.4. USING FORTRAN77 BLAS TO SUPPORT ROW-MAJOR BLAS OPERATIONS

TRMM/TRSM

Because of their identical use of the SIDE, ULO, and TRANS 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 ULO, we implicitly change its TRANS setting as well. With this in mind, we have:

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

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

C call    cblas_ctrmm(CblasLeft, CblasUpper, CblasConjTrans, diag, m, n, α, A, lda, B, ldb)
C op      $B \leftarrow αA^H B$
F77 op    $B^T \leftarrow αB^T A^C$
F77 call  CTRMM('R', 'L', 'C', diag, m, n, α, A, lda, B, ldb)

1.4.3 Banded routines

The above tricks 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 Fortran77, 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 stride 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 $K_L$ 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.

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
\hline
& \text{Super diagonal KU} & & & & & & & & & & \\
\hline
& \text{Super diagonal 2} & & & & & & & & & & \\
\hline
& \text{Super diagonal 1} & & & & & & & & & & \\
\hline
& \text{main diagonal (D)} & & & & & & & & & & \\
\hline
& \text{Sub diagonal 1} & & & & & & & & & & \\
\hline
& \text{Sub diagonal 2} & & & & & & & & & & \\
\hline
& \text{Sub diagonal KL} & & & & & & & & & & \\
\hline
\end{array}
\]

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.

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
\hline
K_L & D & KU \\
\hline
| | | | | | | | | | | | \\
| | | | | | | | | | | | \\
| | | | | | | | | | | | \\
| | | | | | | | | | | | \\
| | | | | | | | | | | | \\
| | | | | | | | | | | | \\
| | | | | | | | | | | | \\
| | | | | | | | | | | | \\
| | | | | | | | | | | | \\
| | | | | | | | | | | | \\
\hline
\end{array}
\]

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
1.4. USING FORTRAN77 BLAS TO SUPPORT ROW-MAJOR BLAS OPERATIONS

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 Fortran77 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 Fortran77 banded array. The matrix transpose, and its Fortran77 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 $K_L$.

1.4.4 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 Fortran77, we will simply switch the setting of UPL0. This will mean that the rows of the matrix will be read in as the columns, but this is OK, 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 UPL0 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.