C Interface to LAPACK (Proposal)

By
David Bindel[1] (e-mail: Bindel@cs.cornell.edu)
Michael Chuvelev[2] (e-mail: Michael.Chuvelev@Intel.com)
James Demmel[3] (e-mail: Demmel@cs.berkeley.com)
Greg Henry[4] (e-mail: Greg.Henry@Intel.com)
Julie Langou[5] (e-mail: Julie@cs.utk.edu)
Julien Langou[6] (e-mail: Julien.Langou@UCDenver.EDU)
Vladimir Koldakov[7] (e-mail: Vladimir.V.Koldakov@Intel.com)
Shane Story[8] (e-mail: Shane.Story@Intel.com)

Version 1.9 (June 2010)

Introduction

This proposal details a C interface to LAPACK in many ways consistent with the C interface to the Legacy BLAS (CBLAS) [3]. We base this document's format and contents on the C interface to the Legacy BLAS document. There have been several prior proposals for a C interface for LAPACK, but these are all vastly different from the CBLAS. For instance, the CLAPACK project, available at NetLib [1] has a FORTRAN interface with C source, and is useful for those who want to build LAPACK [4] but don't have a FORTRAN compiler. Julien Langou and Rémi Delmas have yet another proposal at [2], but it doesn't support matrices in row major order such as is done in the CBLAS and in most C programs.

C users may like to interface with LAPACK from C with a native C interface. However, some C users may not like using a FORTRAN interface. They might like a C-based wrapper, but creating one for all of LAPACK would be time-consuming and error prone. Also, the C-FORTRAN interfacing is compiler dependent. Users requested a richer interface that would include support for row-major data layout and for automatic workspace allocation.

The MKL team would like to provide a reference implementation of a C interface to LAPACK that supports a richer interface. Because of the huge size of LAPACK, this reference implementation has many auto-generated pieces and the emphasis is on functionality and ease of use, not performance.

Versions 1.4 and above of this proposal contain a two-level merged interface, representative of a collaboration between the Intel® Math Kernel Library (MKL) team and the LAPACK team. At the high level, for drivers and computational LAPACK routines, users gain the rich C-based wrapper that allows row-major and column-major interfaces that they might expect from a C

[1] Cornell University, Department of Computer Science, Assistant Professor
[2] Intel Corporation, Software and Services Group, Intel® Math Kernel Library, LAPACK lead
[3] Professor of Mathematics and Computer Science, University of California at Berkeley
[4] Intel Corporation, Software and Services Group, Intel® Math Kernel Library, Architect
[5] Dept. of Electrical Engineering and Computer Science, University of Tennessee, NETLIB LAPACK Team
[6] Dept. of Mathematical and Statistical Sciences, University of Colorado Denver, NETLIB LAPACK Team
[7] Intel Corporation, Software and Services Group, Intel® Math Kernel Library, Modern Languages Interface lead
[8] Intel Corporation, Software and Services Group, Intel® Math Kernel Library, MKL Manager

interface. The MKL team initially provides this high-level interface. At the low level, users also gain a method to access all LAPACK functions, even auxiliary functions, but without the row-major access or any changes in the API. Julien Langou and Julie Langou will initially provide the low-level interface. Together, the two interfaces make a common, fully-featured C Interface to LAPACK.

Naming Scheme for the High-Level Interface

The naming scheme for the high-level interface is to take the FORTRAN 77 LAPACK routine name, make it lower case, and add the prefix **CLAPACK_**. For example, DGETRF becomes **CLAPACK_dgetrf**. The high-level interface only applies to drivers and computational routines, not auxiliary functions. This naming scheme is consistent with the CBLAS.

Filenames are consistent with the routine names, but in all lower case, so the high-level routine that uses DGETRF above is called clapack_dgetrf.c.

Naming Scheme for the Low-Level Interface

The naming scheme for the low-level interface is to take the FORTRAN 77 LAPACK routine name, make it lower case, and add the prefix **LAPACK_**. For example, DGETRF becomes **LAPACK_dgetrf**. The low-level interface applies to all LAPACK routines, but does not have all the features of the high-level interface (support for row-major matrices, no need to pass work arrays, etc..)

Character arguments

In the C BLAS interface, all character arguments are handled by enumerated types. The BLAS only have four character arguments however: TRANSPOSE, UPLO, DIAG, and SIDE. LAPACK has more, shown in the next table:

Character Argument	**Potential Values**
VECT	‘N’, ‘Q’, ‘P’, ‘B’
NORM	‘1’, ‘O’, ‘E’,‘I’,’M’,’F’,’o’,’i’,’m’,’f’,’e’
DIAG	‘N’, ‘U’
NORMIN	‘Y’, ‘N’
TRANS	‘N’, ‘T’, ‘C’
UPLO	‘U’, ‘L’
FACT	‘F’, ‘N’, ‘E’
EQUED (use char *)	‘N’, ‘R’, ‘C’, ‘B’
JOB BALANC	‘N’,’P’,’S’,’B’
JOB (2nd version)	‘E’,’S’
SIDE	‘R’,’L’
SIDE (2nd version)	‘R’,’L’,’B’

JOBVS JOBVL JOBVR JOBVSL JOBVSR JOBZ VECT (2nd version) JOBV	'N','V'
JOBZ (2nd version) JOBU JOBVT	'A','S','O','N'
JOBU (2nd version) INITV	'U','N'
JOBQ EIGSRC	'Q','N'
SORT	'N','S'
ORDER	'B','E'
SENSE	'N','E','V','B'
BALANC	'N','P','S','B'
COMPQ COMPZ	'N','V','I'
RANGE	'A', 'V', 'I'
DIRECT	'F','B'
STOREV	'C','R'
PIVOT	'V','T','B'
HOWMNY	'A','B','S'

Not only do some of the values like 'N' have different meanings for some of the same character arguments in a group, but some character arguments (JOB, for example) have multiple definitions depending on where in LAPACK they are used.

Our proposed C Interface to LAPACK reference version does not check whether input char parameters are valid. The routines may do unnecessary work such as transposing a matrix if the input parameters are invalid.

The Considered Methods section of the C interface to the BLAS gives reasons for not using both "char" and "char *" over enumerated types. There were two arguments in that document for avoiding "char" and "char *". The first argument is that it requires twice as many comparisons to check case ('A' vs. 'a'). The second argument is that there could be potential user input errors (like DIAG='H'). Since both of these reasons apply to the FORTRAN BLAS as well, these are not strong arguments. However, if "char" was used, then a user couldn't write things out like "NoTranspose", as they could in FORTRAN. If instead "char *" was used, then some compilers might fail to optimize string constant use.

With one exception, we suggest “char” declarations. This does have the disadvantages mentioned in the C Interface to the BLAS, but it also allows for the simplest conversion from the FORTRAN codes. Regardless, the wide variety of possible enumerated types and conflicting definitions as well as what to name each type would make using enumerated types ambiguous.

The one exception to this rule is when the character EQUED is also an output argument. This occurs in ?gesvx, ?gbsvx, ?posvx, ?ppsvx, and ?pbsvx. In this case, we must use “char *”.

Handling of Complex Types

Despite the ANSI/ISO C standard not requiring elements within a structure to be contiguous, we require contiguous elements for complex types. The ANSI C99 Standard, released after the original C interface to the BLAS, does support complex types through “#include <complex.h>”.

Complex type arguments will be declared as **clapack_complex_float**/**clapack_complex_double** (input only scalars), or **clapack_complex_float***/**clapack_complex_double*** (arrays and output scalars). By default, these types are aliased to the complex types defined in the C99 standard.

Additionally, the following auxiliary functions for making complex value of real part, re, and imaginary part, im, will be provided to help deal with real and imaginary part of complex values:

clapack_complex_float **clapack_make_complex_float**(float re, float im);
clapack_complex_double **clapack_make_complex_double**(double re, double im);

and the following macros are defined expanding to a real/imaginary part of the complex value z:

clapack_complex_float_real(z)
clapack_complex_float_imag(z)
clapack_complex_double_real(z)
clapack_complex_double_imag(z)

There are several options to define complex types **clapack_complex_float**/**clapack_complex_double**:

1) C99 complex types (the default):

```
#define clapack_complex_float    float _Complex
#define clapack_complex_double   double _Complex
```

2) C structure option (set by enabling in the configuration file):
-DHAVE_CLAPACK_CONFIG_H -DCLAPACK_COMPLEX_STRUCTURE

```
typedef struct { float real, imag; } _clapack_complex_float;
typedef struct { double real, imag; } _clapack_complex_double;

#define clapack_complex_float      _clapack_complex_float;
#define clapack_complex_double  _clapack_complex_double;
```

3) CPP complex type (set by enabling in the configuration file):
-DHAVE_CLAPACK_CONFIG_H -DCLAPACK_COMPLEX_CPP

#define clapack_complex_float std::complex<float>
#define clapack_complex_double std::complex<double>

4.) custom complex types (set by enabling in the configuration file):
-DCLAPACK_COMPLEX_CUSTOM

You will need to:
- define **clapack_complex_float/clapack_complex_double** types by your own,
- optionally define **clapack_complex_float_real/clapack_complex_float_imag/clapack_complex_double_real/clapack_complex_double_imag** macros if you're going to use them,
- optionally define **clapack_make_complex_float/clapack_make_complex_double** functions if you're going to use them, or if you want to build the testing suite supplied, for the testing system uses these functions.

Return Values of Complex Functions

Some LAPACK functions have complex return values, such as CLATM3. These return the same types as used in the handling of complex types.

C Interface to LAPACK Include File

There is a standard include file ("clapack.h") which contains the definitions of the C Interface to LAPACK types and the prototypes for all non-auxiliary C Interface to LAPACK functions.

There is also a configuration file with parameters that users may wish adjust ("clapack_config.h").

Array Arguments in the High-Level Interface

Arrays will be passed as pointers, not as a pointer to pointers. All C Interface to LAPACK routines that take one or more 2D arrays as a pointer receive a single extra parameter, located first. This argument will be of *int* type and will have to be equal to one of the **CLAPACK_ROW_MAJOR**, **CLAPACK_COL_MAJOR** constants, which will be defined in the **C Interface to LAPACK Include File** as following:

```
#define CLAPACK_ROW_MAJOR 0
#define CLAPACK_COL_MAJOR 1
```

If a routine has multiple array inputs, they must all use the same ordering.

We assume that when LAPACK uses other matrix distribution types, such as storing data in block data structures, the above constants can be expanded upon. The current High-Level implementation will check the incoming parameters and if it's not CLAPACK_ROW_MAJOR or CLAPACK_COL_MAJOR, it will flag an error with INFO=-1 (see INFO parameters below.) We use the *int* type to allow for more layouts than might be possible with *char*.

LAPACK routines use a stride corresponding to the FORTRAN leading dimension (LDA) with all 2D arrays. We must do the same. For RowMajor matrices, elements within a row are assumed to be contiguous and elements from one row to the next are assumed to be a stride/leading dimension apart. For ColMajor matrices, elements within a column are assumed to be contiguous and elements from one column to the next are assumed to be a stride/leading dimension apart.

Neither of the other two proposals for a new C Interface to LAPACK include support for Row-Majoring ordering, which was the primary motivation for the development of this third proposal. Likewise, array arguments in the Low-Level interface remain the same as LAPACK.

Note that not all computational routines in LAPACK have matrix arguments. For instance:
`sdisna ddisna sstebz dstebz ssterf dsterf sgtcon dgtcon cgtcon zgtcon sptcon dptcon cptcon zptcon sgttrf dgttrf cgttrf zgttrf spttrf dpttrf cpttrf zpttrf`
The above routines don't use the first parameter matrix_order in the high-level interface.

Aliasing of arguments

Unless specified otherwise, only input arguments (that is, scalars passed by values and arrays specified with the const qualifier) may be legally aliased on a call to C interface to LAPACK.

INFO parameters

The C interface to LAPACK high-level interface contains an "INFO" parameter that indicates valuable information, such as error and exit conditions. The BLAS has no equivalent. However, all LAPACK routines with an integer INFO parameter will have the INFO parameter dropped in the C Interface to LAPACK, and instead it will be an integer return value.

The low-level interface will keep the INFO parameter where it is in the LAPACK routine.

Both interfaces use INFO exactly as it is used in LAPACK. If INFO returns to the row or column number of a matrix using 1-based indexing in FORTRAN, the value is not adjusted for the C Interface to LAPACK.

There is a simple exception to adjusting the INFO parameter. In some LAPACK routines, a negative value of INFO indicates a parameter was incorrect. For example, if INFO=-3, then parameter number 3 was incorrect. But the high-level interface sometimes introduces an additional first parameter matrix_order as defined in above section "Array Arguments in the High-Level Interface." INFO will be adjusted to refer to the correct value in the C Interface to LAPACK for the high-level interface (-3 will become -4). As mentioned in the previous section, not all routines in the high-level interface have the additional parameter. In that event, if calling CLAPACK_ddisna() and the second parameter M is negative, INFO will be set to -2 in the LAPACK ddisna() routine, and CLAPACK_ddisna() routine will also return -2.

Composing LAPACK and the C Interface for LAPACK

The C Interface will not change the indexes to use 0-based numbering instead of the 1-based indexing common in the FORTRAN LAPACK. This enables users to call a routine like CLAPACK_dgetrf and get pivot values in IPIV that can be used with any LAPACK routine that currently works with DGETRF.

Error Checking

Error checking will be at least as rigorous as the FORTRAN interface to LAPACK. Whatever parameters LAPACK checks, we also check in the C interface to LAPACK. Additionally, NaN checks are made in the high-level interface case (see the NaN Checks section.)

NaN Checks / Inf Checks / Other Input Checks
The high-level interface includes an optional (default true) NaN check on all vector/matrix inputs (not scalars) before calling any LAPACK FORTRAN routine. The option affects all routines. If the inputs contain any NaNs, the input parameter corresponding matrix will be flagged with an INFO parameter error.

In early versions of the interface, NaN checking is only enabled in high-level drivers, not the computational routines. It is the plan on a later version of the interface to do NaN checks on all floating point inputs (scalars included) for all routines in the high-level interface.

We may also extend the interface to include Inf checks and/or do other kinds of error checking. At present, this is an open for later consideration and we've not made plans to do this yet.

Our intention is to do only as much NaN or Inf checking as needed to insure unsurprising behavior (such as convergence failures causing infinite loops.)

Adjusting the configuration header file can turn on or off optional parameters like this.

The low-level interface does not contain NaN checks beyond what exist in the LAPACK FORTRAN routines.

Integers

FORTRAN integers shall be converted to "clapack_int" in the C interface. This will conform with modifiable integer type size, especially given ILP64 programming model: re-defining clapack_int as long int (8 bytes) will be enough to support this model, as clapack_int is defined as int (4 bytes) by default, supporting LP64 programming model.

Logicals

FORTRAN logicals shall be converted to "clapack_logical" in the C interface, which will be defined as clapack_int.

Work Arrays and Two Choices for Users

Unlike the BLAS, LAPACK has a number of WORK arrays that are used in various routines, and sometimes more than one. We address two types of users- ones that don't care about the LAPACK allocation and ones that are used to the current LAPACK methodology and wish to continue using the WORK arrays under their own management. Because of this, the C Interface contains both a high-level interface (without a WORK parameter) and a low-level interface (with a WORK parameter if the corresponding LAPACK uses one).

We provide two examples to illustrate.

Example 1: No Work Array in the Sample Routine (SGETRF)
LAPACK : SGETRF (M, N, A, LDA, IPIV, INFO)
C INTERFACE High-Level Version (Non-work based):
```
clapack_int CLAPACK_sgetrf( int matrix_order, clapack_int m,
                            clapack_int n, float* a, clapack_int lda,
                            clapack_int* ipiv );
```
C INTERFACE Low-Level Version (Work based):
```
void LAPACK_sgetrf( clapack_int m,
                                 clapack_int n, float* a,
                                      clapack_int lda, clapack_int* ipiv );
```

Example 2: Work Array in the Sample Routine (DSYTRF)
LAPACK : DSYTRF (UPLO, N, A, LDA, IPIV, WORK, LWORK, INFO)
C INTERFACE High-Level Version (Non-work based):

```
clapack_int CLAPACK_dsytrf( int matrix_order, char uplo, clapack_int n,
                            double* a, clapack_int lda, clapack_int* ipiv );
```

C INTERFACE Low-Level Version (Work based):

```
void LAPACK_dsytrf( char uplo, clapack_int n,
                                 double* a, clapack_int lda,
                                 clapack_int* ipiv, double* work,
                                 clapack_int lwork, clapack_int *info );
```

Note that ROW_MAJOR format may require more work space (as well as time) than COL_MAJOR format, because it may require an out-of-place transposition.

In some cases, the WORK arrays in LAPACK have a dual meaning: as a work array, and as a mechanism to return information to the user. For instance, in xGESVD, the work array on exit may contain unconverged superdiagonal elements. So, in this event, we return an extra array for the interface without the _work parameter:

Example 3: Work array contains superdiagonal elements like SGESVD:
LAPACK: SGESVD (JOBU, JOBVT, M, N, A, LDA, S, U, LDU, VT, LDVT, WORK, LWORK, INFO)
C INTERFACE High-Level Version (Non-work based):

```
clapack_int CLAPACK_sgesvd( int matrix_order, char jobu, char jobvt,
                            clapack_int m, clapack_int n, float* a,
                            clapack_int lda, float* s, float* u,
                            clapack_int ldu, float* vt, clapack_int ldvt,
                            float* superb );
```

C INTERFACE Low-Level Version (Work Based):

```
void LAPACK_sgesvd ( char jobu, char jobvt,
                                 clapack_int m, clapack_int n, float* a,
                                 clapack_int lda, float* s, float* u,
                                 clapack_int ldu, float* vt,
                                 clapack_int ldvt, float* work,
                                 clapack_int lwork, clapack_int info );
```

LAPACK sometimes has data that (beyond suggested work size) gets passed back to the user via the work array. The long-term solution is to examine each instance on a case-by-case basis and agree with the LAPACK team on a new calling parameter. Short term, we just introduce a place-holder ("superb" array). See Appendix 4 on Superb arrays- these are the cases where the normal "rules" require special attention to understand the C Interface. We also include suggestions for fixing each case.

Memory Management
All memory management is placed through CLAPACK_malloc() and CLAPACK_free(). This allows users to easily use their own memory manager instead. An adjustment to the configuration header file allows this.

This interface should be thread-safe to the extent that these memory management routines and the underlying LAPACK routines are thread-safe.

New Error Codes and CLAPACK_xerbla:
Since the high-level interface doesn't have work arrays, we need to flag errors in the event of a user running out of memory. We run these errors through CLAPACK_xerbla, in the same sense

that parameter errors are run through XERBLA in LAPACK, however we suggest that the code be modified to print more insightful messages than "parameter number 1000 had an illegal value."

```
#define CLAPACK_WORK_MEMORY_ERROR       -1010
#define CLAPACK_TRANSPOSE_MEMORY_ERROR  -1011
```

If "info" is used in the calling sequence, we interpret it as follows:

- If this value is more than -1000, its meaning is the same as the NETLIB LAPACK specification.
- If this value is equal to -1010 (CLAPACK_WORK_MEMORY_ERROR), it means that there was not enough memory to allocate a work array.
- If the value is equal to -1011 (CLAPACK_TRANSPOSE_MEMORY_ERROR), it means that the implementation had insufficient memory to complete a transposition.

There is also a NaN check which could increase the number of potential errors (but only with the non-work interface.)

In older versions of this proposal, we also had a CLAPACK_ORDER_ERROR to verify the incoming first parameter (matrix_order) has legal values. Instead, we just consider that error like any other parameter error.

Rules for conversion of LAPACK Routines into High-Level C Interface to LAPACK Routines:

- The FORTRAN 77 NAME is changed to lower case, with a CLAPACK_ prefix
- All Routines with one or more 2D array parameters should require a new parameter as the first argument, matrix ordering and pass in one of the values CLAPACK_ROW_MAJOR or CLAPACK_COL_MAJOR in order to indicate if all the arrays should be accessed in a Row-Major or Column-Major fashion. The name for this additional first parameter shall be "matrix_order". Note that not all LAPACK routines have an array input.
- CHARACTER arguments are replaced with "char", except when they are output parameters (in which case they are simply replaced with char *).
- Input arrays are declared with the "const" modifier.
- Scalar inputs are passed by value
- Integer inputs are replaced with "clapack_int"
- Logical inputs are replaced with "clapack_logical"
- Array arguments and output scalar arguments are passed by address.
- LAPACK routines that have "INFO" output arguments shall instead have these be function return values.
- Function arguments are passed by pointer.
- WORK arrays are omitted from the parameter lists in the high-level interface. See Appendix 4 for dealing with the small number of cases where the work arrays return pertinent information.

Rules for conversion of LAPACK Routines into Low-Level C Interface to LAPACK Routines:

The rules are simpler for using the low-level interface, however there is also less flexibility.

- The FORTRAN 77 NAME is changed to lower case, with a LAPACK_ prefix
- All functions are declared as void

- CHARACTER arguments are replaced with "char", except when they are output parameters (in which case they are simply replaced with char *).
- Input arrays are declared with the "const" modifier.
- Scalar inputs are passed by value
- Integer inputs are replaced with "clapack_int"
- Logical inputs are replaced with "clapack_logical"
- Array arguments and output scalar arguments are passed by address.
- Function arguments are passed by pointer.

Differences between the C interface to LAPACK and the C interface to the BLAS

Unlike previous proposals, there are only a few differences between the C interface to LAPACK and the C interface to the BLAS

- There are no enumerated types for the C interface to LAPACK
- There are no redundant const qualifiers for input scalar parameters
- Some of the routines hide some of the parameters or use them differently, without significant name changes. For instance, the use of converting INFO into a return value.
- There is no notion of a work array in the parameter lists of the BLAS routines. Therefore, there is no decision about whether or not to keep these in the C interface to the BLAS. The decision here is to drop these unnecessary parameters.

References

[1] NetLib : http://www.netlib.org

[2] Julien Langou and Rémi Delmas, 2006, C Interface proposal: http://icl.cs.utk.edu/~delmas/lapwrapc.html , http://icl.cs.utk.edu/~delmas/presentation_wrapper_C.pdf

[3] C Interface to the BLAS, BLAS Technical Forum, http://www.netlib.org/blas/blast-forum/cinterface.pdf

[4] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorenson, D., LAPACK Users' Guide, SIAM Publications, Philadelphia, PA, 1992

[5] Dongarra, J., Du Croz, J., Hammarling S., Duff, I.S., *A set of Level 3 basic linear algebra subprograms*, ACM TOMS, 16(1):1—17, March 1990

Appendix 1: High-level C Interface Examples

```
FORTRAN
SUBROUTINE DGEQRF( M, N, A, LDA,
TAU, WORK, LWORK, INFO )

INTEGER INFO, LDA, LWORK, M, N
DOUBLE PRECISION A( LDA, * ),
TAU( * ), WORK( * )
SUBROUTINE ZHEEVD( JOBZ, UPLO, N,
A, LDA, W, WORK, LWORK, RWORK,
LRWORK, IWORK, LIWORK, INFO )

CHARACTER JOBZ, UPLO
INTEGER INFO, LDA, LIWORK,
LRWORK, LWORK, N
INTEGER IWORK( * )
DOUBLE PRECISION RWORK( * ), W( *
```

```
High-Level C Interface
clapack_int CLAPACK_dgeqrf( int
matrix_order, clapack_int m, clapack_int
n, double* a, clapack_int lda, double* tau
);


clapack_int CLAPACK_zheevd( int
matrix_order, char jobz, char uplo,
clapack_int n, clapack_complex_double* a,
clapack_int lda, double* w );
```

) COMPLEX*16 A(LDA, *), WORK(*)	
SUBROUTINE **CGECON**(NORM, N, A, LDA, ANORM, RCOND, WORK, RWORK, INFO) CHARACTER NORM INTEGER INFO, LDA, N REAL ANORM, RCOND REAL RWORK(*) COMPLEX A(LDA, *), WORK(*)	clapack_int **CLAPACK_cgecon**(int matrix_order, char norm, clapack_int n, const clapack_complex_float* a, clapack_int lda, float anorm, float* rcond);
SUBROUTINE **SGEESX**(JOBVS, SORT, SELECT, SENSE, N, A, LDA, SDIM, WR, WI, VS, LDVS, RCONDE, RCONDV, WORK, LWORK, IWORK, LIWORK, BWORK, INFO) CHARACTER JOBVS, SENSE, SORT INTEGER INFO, LDA, LDVS, LIWORK, LWORK, N, SDIM REAL RCONDE, RCONDV LOGICAL BWORK(*) INTEGER IWORK(*) REAL A(LDA, *), VS(LDVS, *), WI(*), WORK(*), WR(*) LOGICAL SELECT EXTERNAL SELECT	clapack_int **CLAPACK_sgeesx**(int matrix_order, char jobvs, char sort, CLAPACK_S_SELECT2 select, char sense, clapack_int n, float* a, clapack_int lda, clapack_int* sdim, float* wr, float* wi, float* vs, clapack_int ldvs, float* rconde, float* rcondv);

*Additional datatypes should be used:

```
typedef clapack_logical (*CLAPACK_S_SELECT2) ( const float*, const float* );
typedef clapack_logical (*CLAPACK_S_SELECT3) ( const float*, const float*,
const float* );

typedef clapack_logical (*CLAPACK_D_SELECT2) ( const double*, const double*
);
typedef clapack_logical (*CLAPACK_D_SELECT3) ( const double*, const double*,
const double* );

typedef clapack_logical (*CLAPACK_C_SELECT1) ( const clapack_complex_float*
);
typedef clapack_logical (*CLAPACK_C_SELECT2) ( const clapack_complex_float*,
const clapack_complex_float* );

typedef clapack_logical (*CLAPACK_Z_SELECT1) ( const clapack_complex_double*
);
typedef clapack_logical (*CLAPACK_Z_SELECT2) ( const clapack_complex_double*,
const clapack_complex_double* );
```

Some FORTRAN compilers may rely on a certain value of the logical TRUE predicate. In that event, you must use the same return value in a predicate function for consistency if you link against LAPACK compiled with that FORTRAN compiler.

Appendix 2: Implementation Issues

Generally, LAPACK routines do not need to be re-implemented to support C interfaces. A binding to existing FORTRAN routines can be done via wrappers. An example is shown below:

The example of such an interface based on wrappers is as follows:

```c
clapack_int CLAPACK_dgeqrf( int matrix_order, clapack_int m, clapack_int n,
double* a, clapack_int lda, double* tau ) {
   // locals
   clapack_int m_ = m, n_ = n, lda_ = lda, info, lwork;
   double *at, *work, wopt;
   clapack_int i, j;
   // allocate space for working arrays
   lwork = -1;
   dgeqrf_( &m_, &n_, a, &lda_, tau, &wopt, &lwork, &info );
   lwork = (int) wopt;
   work = (double*) CLAPACK_malloc( lwork*sizeof( double ) );
   if( order == CLAPACK_ROW_MAJOR ) {
      // allocate space for the matrices to be transposed
      at = (double*) malloc( m*n*sizeof( double ) );
      // transpose input matrices
      for( i = 0; i < m; i++ )
         for( j = 0; j < n; j++ )
            at[i+m*j] = a[i*lda+j];
      // call kernel
      dgeqrf_( &m_, &n_, at, &m_, tau, work, &lwork, &info );
      // transpose output matrices back
      for( i = 0; i < m; i++ )
         for( j = 0; j < n; j++ )
            a[i*lda+j] = at[i+m*j];
      // deallocate space
      CLAPACK_free( (void*) at );
   } else { // Column-major
      // call kernel
      dgeqrf_( &m_, &n_, a, &lda_, tau, work, &lwork, &info );
   }
   CLAPACK_free( (void*) work );
   // return info
   return info;
}
```

Disadvantages to using a wrapper approach to access existing FORTRAN subprograms are increased memory footprints and lower performance where a transpose is needed. Another way to implement row-major C interfaces without an increase in memory is to re-organize the *LAPACK* algorithms (for instance, LU factorization to be mapped on row-major ordering) or use tricks to call some complementary routine such as `dgelqf` for `dgeqrf`, as in the example below:

```c
clapack_int CLAPACK_dgeqrf( int matrix_order, clapack_int m, clapack_int n,
double* a, clapack_int lda, double* tau ) {
   // locals
   clapack_int m_ = m, n_ = n, lda_ = lda, info, lwork;
   double *work, wopt;
   if( matrix_order == CLAPACK_ROW_MAJOR ) {
      // allocate space for working arrays
      lwork = -1;
      dgelqf_( &n_, &m_, a, &lda_, tau, &wopt, &lwork, &info );
      lwork = (int) wopt;
      work = (double*) CLAPACK_malloc( lwork*sizeof( double ) );
      // call kernel
      dgelqf_( &n_, &m_, a, &lda_, tau, work, &lwork, &info );
      // deallocate space
      CLAPACK_free( (void*) work );
   } else { // Column-major
      // allocate space for working arrays
      lwork = -1;
      dgeqrf_( &m_, &n_, a, &lda_, tau, &wopt, &lwork, &info );
      lwork = (int) wopt;
      work = (double*) CLAPACK_malloc( lwork*sizeof( double ) );
      // call kernel
      dgeqrf_( &m_, &n_, a, &lda_, tau, work, &lwork, &info );
      // deallocate space
      CLAPACK_free( (void*) work );
   }
   // return info
   return info;
}
```

Another trick is changing UPLO parameter on entry to the FORTRAN subroutine, as in the `dpotrf` example below:

```
clapack_int CLAPACK_dpotrf( int matrix_order, char uplo, clapack_int n,
double* a, clapack_int lda ) {
   // locals
   clapack_int n_ = n, lda_ = lda, info;
   char uplo_ = uplo;
   if( matrix_order == CLAPACK_ROW_MAJOR ) {
      uplo_ = clapack_lsame( uplo, 'u' ) ? 'L' : 'U';
   }
   // call kernel
   dpotrf_( &uplo_, &n_, a, &lda_, &info );
   // return info
   return info;
}
```

The C Interface to LAPACK may not produce the same bitwise identical results as the underlying LAPACK routine for several reasons: data alignment differences, transposition leading to other algorithms, and different optimization abilities with different compilers. Of course, a different compiler may make two different LAPACK builds behave slightly different.

Appendix 3: Testing Issues

One option for C interface testing is to port LAPACK testing system into C language with adaptation to the C interface. This is a normal way of testing.

Another option is to compare the computed results by C interface function with its FORTRAN counterpart obtained with exactly the same parameters. An advantage of this kind of testing is that the tests can be generated.

The example of the generated test fragment is below:

```
int main(void)
{

    // Declare scalars
    char uplo, uplo_i;
    clapack_int n, n_i;
    clapack_int lda, lda_i;
    clapack_int lda_r;
    clapack_int info, info_i;
    int pass;

    // Declare arrays
    double *a = NULL, *a_i = NULL;

    // Declare row_major arrays
    double *a_r = NULL;

    // Init data and call lapack
    init_dpotrf_6(&uplo,&n,&a,&a_r,&lda,&lda_r,&info);
    dpotrf(&uplo,&n,a,&lda,&info);

    // Init data and call clapack work
    init_dpotrf_6(&uplo_i,&n_i,&a_i,&a_r,&lda_i,&lda_r,&info_i);
    info_i = CLAPACK_dpotrf_work(ColMajor,uplo_i,n_i,a_i,lda_i);

    // Compare clapack work results
    pass = compare_dpotrf_6(uplo,uplo_i,n,n_i,a,a_i,lda,lda_i,info,info_i);
    if(pass == 0) {
        printf("pass: CLAPACK_dpotrf_work column-major dpotrf_6\n");
    } else {
        printf("fail: CLAPACK_dpotrf_work column-major dpotrf_6\n");
    }

    // Free clapack data
    free_dpotrf_6(a_i);
    free_dpotrf_6(a_r);

    // Init data and call clapack
    init_dpotrf_6(&uplo_i,&n_i,&a_i,&a_r,&lda_i,&lda_r,&info_i);
    info_i = CLAPACK_dpotrf(ColMajor,uplo_i,n_i,a_i,lda_i);

    // Compare clapack results
    pass = compare_dpotrf_6(uplo,uplo_i,n,n_i,a,a_i,lda,lda_i,info,info_i);
    if(pass == 0) {
        printf("pass: CLAPACK_dpotrf column-major dpotrf_6\n");
    } else {
        printf("fail: CLAPACK_dpotrf column-major dpotrf_6\n");
    }

    // Init data and call row-major clapack work
    init_dpotrf_6(&uplo_i,&n_i,&a_i,&a_r,&lda_i,&lda_r,&info_i);
    info_i = CLAPACK_dpotrf_work(RowMajor,uplo_i,n_i,a_r,lda_r);

    // Transpose row_major arrays back
    trans_dpotrf_6(&uplo_i,&n_i,a_i,a_r,&lda_i,&lda_r,&info_i);

    // Compare clapack work results (row major)
    pass = compare_dpotrf_6(uplo,uplo_i,n,n_i,a,a_i,lda,lda_i,info,info_i);
    if(pass == 0) {
        printf("pass: CLAPACK_dpotrf_work row-major dpotrf_6\n");
    } else {
        printf("fail: CLAPACK_dpotrf_work row-major dpotrf_6\n");
    }

    // Free clapack data
    free_dpotrf_6(a_i);
    free_dpotrf_6(a_r);

    // Free data
    free_dpotrf_6(a);

    return 0;
}
```

Appendix 4: "Superb" Arrays: When Work arrays do more

We list here special cases when the work arrays that the High-level interface omits end up removing information from the user. There are 4 cases that we consider, with four different proposed solutions.

Case 1: Work contains a single element
The following 8 routines contain just a single element from the work array that the user may require: sgesvx, dgesvx, cgesvx, zgesvx, sgbsvx, dgbsvx, cgbsvx, zgbsvx.

In these cases, we suggest introducing a new element to the parameter list at the very end of the list. For instance, the normal calling sequence for sgesvx is:

```
      SUBROUTINE SGESVX( FACT, TRANS, N, NRHS, A, LDA, AF, LDAF, IPIV,
     $                   EQUED, R, C, B, LDB, X, LDX, RCOND, FERR, BERR,
     $                   WORK, IWORK, INFO )
      CHARACTER          EQUED, FACT, TRANS
      INTEGER            INFO, LDA, LDAF, LDB, LDX, N, NRHS
      REAL               RCOND
      INTEGER            IPIV( * ), IWORK( * )
      REAL               A( LDA, * ), AF( LDAF, * ), B( LDB, * ),
     $                   BERR( * ), C( * ), FERR( * ), R( * ),
     $                   WORK( * ), X( LDX, * )
```

Unfortunately, on output, WORK(1) contains the reciprocal pivot growth factor. Because of this, we suggest that we add a new parameter "rpivot" for this reciprocal pivot.

Our suggestion is:

```
clapack_int CLAPACK_sgesvx( int matrix_order, char fact, char trans,
clapack_int n, clapack_int nrhs, float* a, clapack_int lda, float* af,
clapack_int ldaf, clapack_int* ipiv, char* equed, float* r, float* c, float*
b, clapack_int ldb, float* x, clapack_int ldx, float* rcond, float* ferr,
float* berr, float* rpivot );
```

The same "rpivot" is used for all 7 routines although its precision would vary from float * to double * depending on whether the underlying routine is single or double precision accordingly.

Case 2: Work contains a sequence of variable length
The following four routines contain a sequence of elements of variable length from the work array that the user may require: sgesvd, dgesvd, cgesvd, zgesvd.

In these cases, we suggest the use of the superb array to capture this input. For instance, the normal calling sequence for sgesvd is:

```
      SUBROUTINE SGESVD( JOBU, JOBVT, M, N, A, LDA, S, U, LDU, VT, LDVT,
     $                   WORK, LWORK, INFO )
      CHARACTER          JOBU, JOBVT
      INTEGER            INFO, LDA, LDU, LDVT, LWORK, M, N
      REAL               A( LDA, * ), S( * ), U( LDU, * ),VT( LDVT, * ), WORK( * )
```

Unfortunately, on output, when INFO > 0, unconverged superdiagonal elements of the bidiagonal B are stored in WORK, where the size is of these depends on M, N. Our suggestion is to store these extra elements in a "superb" array:

```
clapack_int CLAPACK_sgesvd( int matrix_order, char jobu, char jobvt,
clapack_int m, clapack_int n, float* a, clapack_int lda, float* s, float* u,
clapack_int ldu, float* vt, clapack_int ldvt, float* superb );
```

The same “superb” array is useful for all 4 routines, although its precision would vary from float * to double * depending on whether the underlying routine is single or double precision accordingly.

Case 3: Work contains a single sequence of fixed length
The following two routines contain a single sequence of elements of fixed length from the work array that the user may require: dgesvj, sgesvj.

```
      SUBROUTINE DGESVJ( JOBA, JOBU, JOBV, M, N, A, LDA, SVA,
     &                   MV, V, LDV, WORK, LWORK, INFO )
      INTEGER     INFO, LDA, LDV, LWORK, M, MV, N
      CHARACTER*1 JOBA, JOBU, JOBV
      DOUBLE PRECISION A( LDA, * ), SVA( N ), V( LDV, * ), WORK( LWORK )
```

On exit, 6 statistics about the code are saved off at the front of the WORK array. So, we propose a “stat” array of the appropriate precision. For instance:

```
clapack_int CLAPACK_dgesvj( int matrix_order, char joba, char jobu, char
jobv, clapack_int m, clapack_int n, double* a, clapack_int lda, double* sva,
clapack_int mv, double* v, clapack_int ldv, double* stat );
```

Case 4: Work contains two sequences of fixed length
The following two routines contain two sequences of fixed length from the work array that the user may require: dgejsv, sgejsv

```
      SUBROUTINE DGEJSV( JOBA, JOBU, JOBV, JOBR, JOBT, JOBP,
     &                 M, N, A, LDA, SVA, U, LDU, V, LDV,
     &                 WORK, LWORK, IWORK, INFO )
      INTEGER     INFO, LDA, LDU, LDV, LWORK, M, N
      DOUBLE PRECISION A( LDA, * ), SVA( N ), U( LDU, * ), V( LDV, * ),
     &          WORK( LWORK )
      INTEGER     IWORK( * )
      CHARACTER*1 JOBA, JOBP, JOBR, JOBT, JOBU, JOBV.
```

On exit, both integer statistics and floating point statistics are saved off in IWORK and WORK respectively. So, we propose the introduction of two new parameters: stat and istat:

```
clapack_int CLAPACK_dgejsv( int matrix_order, char joba, char jobu, char
jobv, char jobr, char jobt, char jobp, clapack_int m, clapack_int n, const
double* a, clapack_int lda, double* sva, double* u, clapack_int ldu, double*
v, clapack_int ldv, double* stat, clapack_int* istat );
```