# C++ API for BLAS and LAPACK

Mark Gates ICL[1]
Piotr Luszczek ICL
Jakub Kurzak ICL
Jack Dongarra ICL
Konstantin Arturov Intel[2]
Cris Cecka NVIDIA[3]
Chip Freitag AMD[4]

[1]Innovative Computing Laboratory
[2]Intel Corporation
[3]NVIDIA Corporation
[4]Advanced Micro Devices, Inc.

July 2, 2017

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

| **Revision** | **Notes** |
|---|---|
| 06-2017 | first publication |

```
@techreport{gates2017cpp,
  author={Gates, Mark and Luszczek, Piotr and Kurzak, Jakub and Dongarra, Jack,
          and Arturov, Konstantin and Cecka, Cris and Freitag, Chip},
  title={C++ API for BLAS and LAPACK},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2017},
  month={June},
  type={SLATE Working Note},
  number={2},
  note={revision 06-2017}
}
```

# Contents

# CHAPTER 1

# Introduction and Motivation

The Basic Linear Algebra Subroutines[1] (BLAS) and Linear Algebra PACKage[2] (LAPACK) have been around for many decades and serve as *de facto* standards for performance-portable and numerically robust implementations of essential linear algebra functionality. Both are written in Fortran, with C interfaces provided by CBLAS and LAPACKE, respectively.

BLAS and LAPACK will serve as building blocks for the SLATE project. However, their current Fortran and C interfaces are not suitable for SLATE's templated C++ implementation. The primary issue is that the data type – single, double, complex-single, and complex-double – is specified in the routine name: `sgemm`, `dgemm`, `cgemm`, and `zgemm`, respectively. A templated algorithm requires a consistent interface with the same function name to be called for all data types. Therefore, we are proposing a new C++ interface layer on top of the existing BLAS and LAPACK libraries.

We start with a survey of traditional BLAS and LAPACK libraries, both the Fortran and C interfaces. Then we review various C++ linear algebra libraries to see the trends and features available. Finally, Chapter 3 covers our proposed C++ API for BLAS and LAPACK.

[1]http://www.netlib.org/blas/
[2]http://www.netlib.org/lapack/

# CHAPTER 2

# Standards and Trends

## 2.1 Programming Language Fortran

### 2.1.1 FORTRAN 77

The original FORTRAN [1] BLAS first proposed Level 1 BLAS routines for vectors operations with $O(n)$ work on $O(n)$ data. Level 2 BLAS routines were added for matrix-vector operations with $O(n^2)$ work on $O(n^2)$ data. Finally, Level 3 BLAS routines for matrix-matrix operations benefit from the surface-to-volume effect of $O(n^2)$ data to read for $O(n^3)$ work.

Routines are named to fit within FORTRAN 77's 6 letter name limit. The prefix denotes the precision:

| | |
|---|---|
| s | single (float) |
| d | double |
| c | complex-single |
| z | complex-double |

For BLAS-2 and BLAS-3, a two letter combination gives the type of matrix:

[1]FORTRAN refers to FORTRAN 77 and earlier standards. All caps spelling has since been abandoned and first-letter-capitalized spelling is now preferred and used uniformly throughout the standard documents.

| | |
|---|---|
| `ge` | general rectangular |
| `gb` | general band |
| `sy` | symmetric |
| `sp` | symmetric, packed storage |
| `sb` | symmetric band |
| `he` | Hermitian |
| `hp` | Hermitian, packed storage |
| `hb` | Hermitian band |
| `tr` | triangular |
| `tp` | triangular, packed storage |

Finally, the root specifies the operation, such as:

| | |
|---|---|
| `axpy` | $y = \alpha x + y$ |
| `copy` | $y = x$ |
| `scal` | scaling $x = \alpha x$ |
| `mv` | matrix-vector multiply, $y = \alpha Ax + \beta y$ |
| `mm` | matrix-matrix multiply, $C = \alpha AB + \beta C$ |
| `rk` | rank-$k$ update, $C = \alpha AA^T + \beta C$ |
| `r2k` | rank-$2k$ update, $C = \alpha AB^T + \alpha BA^T + \beta C$ |
| `sv` | matrix-vector solve, $Ax = b$, $A$ triangular |
| `sm` | matrix-matrix solve, $AX = B$, $A$ triangular |

and so forth. So, `dgemm` would be a double-precision, general matrix-matrix multiply.

There are a number of limitations of the original Fortran interfaces:

- Portability issues calling Fortran. Since Fortran is case-insensitive, compilers variously use `dgemm`, `dgemm\_`, and `DGEMM` as the actual function name in the binary object file. Typically macros are used to abstract these differences in C/C++.
- Portability issues for routines returning numbers, such as nrm2 and dot (norm and dot product). The Fortran standard doesn't specify how numbers are returned, e.g., on the stack or as an extra hidden argument, so compilers return them in various ways. f2c and old g77 versions also returned singles as doubles (this remains an issue when using MacOS X Accelerate, which is based on the f2c version of LAPACK/BLAS).
- Lacks mixed precision, e.g., $y = Ax$ where $A$ is single and $x$ is double. These are important for mixed-precision iterative refinement routines.
- Lacks mixed real/complex routines, e.g., $y = Ax$ where $A$ is complex and $x$ is real. These occur in some eigenvalue routines.
- Since the precision is encoded in the name, they can't readily be used in precision-independent template code (either C++ or Fortran 90).

### 2.1.2 BLAST XBLAS

The BLAS technical forum[2] (BLAST) added extended and mixed-precision BLAS routines, termed the XBLAS, with suffixes to the routine name indicating the extended datatypes. Using `gemm` as an example, the initial precision (e.g., `z` in `zgemm`) specified the precision of the output matrix $C$ and scalars $(\alpha, \beta)$. For mixed precision a suffix of the form `_a\_b` was added, where `a` and `b` are letters `s, d, c`, or `z` indicating the types of the $A$ and $B$ matrices, respectively. For extended precision, a suffix `_X` was added that specified it internally used extended precision.

While this added capabilities to the BLAS, there remain several issues:

- Extended precision was done only internally: output arguments were in standard precision. For parallel algorithms, the output matrix needs to be in higher precision for reductions. For instance, a parallel `gemv` would do `gemv` on each node with the local matrix, then a parallel reduction to find the final product. To effectively use higher precision, the result of the local `gemv` must be in higher precision, with rounding to lower precision only after the parallel reduction. XBLAS did not provide extended precision outputs.
- Many of the XBLAS routines are superfluous, not being useful in writing LAPACK and ScaLAPACK, making implementation of XBLAS unnecessarily difficult.
- XBLAS had no mechanism to support additional types such as half-precision (16 bit), integer and quantized, fixed point, extended precision such as double-double (two 64-bit quantities representing one value) or quad (128-bit).
- The XBLAS was not widely adopted or implemented. LAPACK can be built using XBLAS in some routines. Intel MKL provides XBLAS implementations.

### 2.1.3 Fortran 90

The BLAST forum also introduced a Fortran 90 interface, which includes precision-independent wrappers around all the routines, and makes certain arguments optional with default values (e.g., assume $\alpha = 1$ or $\beta = 0$ if not given).

## 2.2 Programming Language C

### 2.2.1 Netlib CBLAS

The BLAS technical forum also introduced CBLAS, a C wrapper around the original Fortran BLAS routines. CBLAS addresses a couple of inconveniences when using the Fortran interface directly from C. It allows for passing of scalar arguments by value, rather than by reference, it replaces character parameters by enumerated types, and it deals with mangling of Fortran routine names by the compiler. Also, CBLAS supports the row-major matrix layout, in addition to the standard column-major layout. Notably, this is handled without actually transposing the

[2] http://www.netlib.org/blas/blast-forum/blast-forum.html

matrices, but by changing the transposition, upper/lower, and dimension arguments. Netlib CBLAS declarations reside in the `cblas.h` header file. It contains declarations of a handful of types, i.e.:

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

and signatures of all the functions, e.g.:

```
void cblas_dtrsm(CBLAS_LAYOUT layout, CBLAS_SIDE Side,
                 CBLAS_UPLO Uplo, CBLAS_TRANSPOSE TransA,
                 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_ztrsm(CBLAS_LAYOUT layout, CBLAS_SIDE Side,
                 CBLAS_UPLO Uplo, CBLAS_TRANSPOSE TransA,
                 CBLAS_DIAG Diag, const int M, const int N,
                 const void *alpha, const void *A, const int lda,
                 void *B, const int ldb);
```

Notably, Netlib CBLAS does not introduce a complex type, due to the lack of a standard C complex type at that time. Instead, complex parameters are declared as `void*`. Routines that return a complex value in Fortran are recast as subroutines in the C interface, with the return value being an output parameter added to the end of the argument list, which allows them to also be of type `void*`. Also, the name is suffixed by `_sub`:

```
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);
```

CBLAS contains one function, `i_amax`, in 4 precision flavors, that returns an integer value used for indexing an array. Keeping with C language conventions, it indexes from 0, instead of from 1 as the Fortran `i_amax` does. The type is `int` by default and can be changed to `long` by setting the amusing flag `WeirdNEC`:

```
#ifdef WeirdNEC
   #define CBLAS_INDEX long
#else
   #define CBLAS_INDEX int
#endif

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);
```

In terms of style, CBLAS uses: capital snake case for type names, lower snake case for function names, prefixed with `cblas_`, and Pascal case for constant names. In function signatures, CBLAS uses: lower case for scalars, single capital letter for arrays, and Pascal case for enumerations. Also, CBLAS uses `const` for all read-only input parameters, both scalars and arrays.

To address the issue of Fortran name mangling, CBLAS allows for Fortran routine names to be upper case, lower case, or lower case with an underscore, e.g.: `DGEMM`, `dgemm`, or `dgemm_`. Appropriate renaming is done by C preprocessor macros.

### 2.2.2 MKL CBLAS

MKL CBLAS follows most of the conventions of the Netlib CBLAS with two main exceptions. First, `CBLAS_INDEX` is defined as `size_t`. Second, all integer parameters are of type `MKL_INT`, which can be either 32 bit or 64 bit. Also, header files in MKL are prefixed with `mkl_`, and, therefore, the CBLAS header file is `mkl_cblas.h`.

### 2.2.3 Netlib lapack_cwrapper

The lapack_cwrapper was an initial attempt to develop a C wrapper for LAPACK, similar in nature to the Netlib CBLAS. Similarly to CBLAS, the lapack_cwrapper replaced character parameters with enumerated types, replaced passing of scalars by reference with passing by value, and dealt with Fortran name mangling. The name of the main header file was `lapack.h`.

Enumerated types included all the types defined in CBLAS, and notably, preserved their integer values.

```c
enum lapack_order_type {
            lapack_rowmajor = 101,
            lapack_colmajor = 102 };

enum lapack_trans_type {
            lapack_no_trans   = 111,
            lapack_trans      = 112,
            lapack_conj_trans = 113 };

enum lapack_uplo_type  {
            lapack_upper       = 121,
            lapack_lower       = 122,
            lapack_upper_lower = 123 };

enum lapack_diag_type {
            lapack_non_unit_diag = 131,
            lapack_unit_diag     = 132 };

enum lapack_side_type {
            lapack_left_side  = 141,
            lapack_right_side = 142 };
```

At the same time, many new types were introduced to cover all the other cases of character constants in LAPACK, e.g.:

```c
enum lapack_norm_type {
            lapack_one_norm       = 171,
            lapack_real_one_norm  = 172,
            lapack_two_norm       = 173,
            lapack_frobenius_norm = 174,
            lapack_inf_norm       = 175,
            lapack_real_inf_norm  = 176,
            lapack_max_norm       = 177,
            lapack_real_max_norm  = 178 };

enum lapack_symmetry_type {
            lapack_general                    = 231,
            lapack_symmetric                  = 232,
            lapack_hermitian                  = 233,
            lapack_triangular                 = 234,
            lapack_lower_triangular           = 235,
            lapack_upper_triangular           = 236,
            lapack_lower_symmetric             = 237,
            lapack_upper_symmetric             = 238,
```

Same as CBLAS, lapack_cwrapper used the `void*` type for passing of complex arguments and applied the `const` keyword to all read-only parameters, both scalars and arrays.

Notably, lapack_cwrapper preserved all the semantics of the original API, did not introduce support for row-major layout, did not introduce any extra checks, such as NaN checks, and did not introduce automatic workspace allocation.

In terms of style, all names were snake case: types, constants, functions. In function signatures, lapack_cwrapper used small letters only. Function names were prefixed with `lapack_`. Notably, the name CLAPACK and prefix `clapack_` were not used, to avoid confusion with an incarnation of LAPACK that was expressed in C, by automatically translating the Fortran codes using the F2C tool. The confusing part was that, while being implemented in C, CLAPACK preserved the Fortran calling convention.

### 2.2.4 LAPACKE

LAPACKE is another C language wrapper for LAPACK, originally developed by Intel and later incorporated into LAPACK. Similarly to CBLAS, LAPACKE replaces passing of scalars by reference with passing by value. LAPACKE also deals with Fortran name mangling in the same manner as CBLAS. Unlike CBLAS and lapack_cwrapper, LAPACKE did not replace character parameters with enumerate types.

Unlike all the other C APIs, LAPACKE actually uses complex types for complex parameters. LAPACKE introduces `lapack_complex_float` and `lapack_complex_double`, set by default to `float _Complex` and `double _Complex` (relying on the definition of `_Complex` in `complex.h`). For integers, LAPACKE uses `lapack_int` which is defined as `int` by default, and as `long` if the flag `LAPACK_ILP64` is set.

Similarly to CBLAS, the first parameter in LAPACKE calls is the matrix layout. Two constants are defined, with CBLAS compliant integer values:

```
#define LAPACK_ROW_MAJOR               101
#define LAPACK_COL_MAJOR               102
```

However, unlike in CBLAS, support for row-major layout cannot be implemented by changing the values of transposition and lower/upper arguments. Here, the matrices have to be actually transposed.

LAPACKE offers two interfaces: a higher level interface, with names prefixed by `LAPACKE_`, and a lower level interface, with names prefixed by `LAPACKE_` and suffixed by `_work`, e.g.:

```
lapack_int LAPACKE_zgecon( int matrix_layout, char norm, lapack_int n,
                           const lapack_complex_double* a, lapack_int lda,
                           double anorm, double* rcond );

lapack_int LAPACKE_zgecon_work( int matrix_layout, char norm, lapack_int n,
                                const lapack_complex_double* a, lapack_int lda,
                                double anorm, double* rcond,
                                lapack_complex_double* work, double* rwork );
```

In the case of `matrix_layout=LAPACK_COL_MAJOR`, the lower level interface (`_work` suffix) serves only as a simple wrapper with no extra functionality added. In the case of `matrix_layout=LAPACK_ROW_MAJOR`, the lower level interface performs out-of-place transpositions of all the input arrays and corresponding allocations and deallocations. At the same time, the lower level interface preserves the LAPACK convention of leaving it up to the user to allocate the required workspaces.

The higher level interface (no `_work` suffix) eliminates the requirement for the user to allocate workspaces. Instead, the workspace allocation is done inside the routine, after the appropriate query for the required size.

At the same time, the higher level interface performs NaN checks for all the input arrays, which can be disabled, if LAPACKE is compiled from sources, by setting the `LAPACK_DISABLE_NAN_CHECK` flag (not possible in the case of a binary distribution).

### 2.2.5 Next Generation BLAS "G2"

A new, ongoing effort, first presented at the Batched, Reproducible, and Reduced Precision BLAS workshop[3], is to develop the next generation of BLAS, termed BLAS G2. This introduces a new naming scheme for the lower level BLAS routines that is more flexible than the single prefix character in the original BLAS and XBLAS. It uses suffixes for data types, for example:

| | |
|---|---|
| `r16` | half (16-bit float) |
| `r32` | single |
| `r64` | double |
| `c32` | complex-single |
| `c64` | complex-double |
| `r64x2` | extended double-double |

Arguments can either share the same precision (e.g., all `r64` for traditional `dgemm`), or can have mixed precisions, such as `blas_gemm_r32r32r64`, which has single-precision matrices $A$ and $B$, and double-precision matrix $C$. It also defines extensions such as having different input and output matrices, $C_{\text{in}}$ and $C_{\text{out}}$, having reproducible accumulators that give the same answer regardless of runtime choices in parallelism or evaluation order.

This gives a mechanism to name the various routines. However, not all names that fit the mechanism would be implemented. A set of recommended routines to implement will also be defined.

It is expected that users would use higher level interfaces in C++ and Fortran that overload the basic operations, e.g., C++ `blas::gemm` would call the correct lower-level routine depending on the precision of the arguments it is given.

[3] http://www.netlib.org/utk/people/JackDongarra/WEB-PAGES/Batched-BLAS-2017/

## 2.3 Programming Language C++

A number of C++ linear algebra libraries also exist. Most of these provide actual implementations of BLAS-like functionality in C++, rather than being simply wrappers as CBLAS and LAPACKE are. Some can also call the high-performance vendor-optimized traditional BLAS, at least in some instances.

### 2.3.1 Boost::uBLAS

Boost is a widely used collection of C++ libraries covering many topics. Some of the features developed in Boost have later been adopted into the C++ standard template library (STL). As one library within Boost, uBLAS[4] provides Level 1, 2, 3 BLAS functionality for dense, banded, and sparse matrices. It is implemented using expression templates with lazy evaluation. Basic expressions on whole matrices are easy to specify. Example gemm calls:

```
    // C = alpha A B
    C = alpha * prod( A, B );

    // C = alpha A^H B + beta C
    noalias(C) = alpha * prod( herm(A), B ) + beta * C;
```

where noalias prevents creating a temporary result. (While use of noalias in this case is a bit dubious, since $C$ is on the RHS, the result appears to be correct.) It can access submatrices, both contiguous ranges and slices (with stride between rows and cols). However, the syntax is rather cumbersome:

```
    noalias( project( C, range(0,i), range(0,j) ))
        = alpha * prod( project( A, range(0,i), range(0,k) ),
                        project( B, range(0,k), range(0,j) ) )
        + beta * project( C, range(0,i), range(0,j) );
```

Because code is templated, any combination of precisions, real, and complex values works. Its interface is "mostly" conforming with C++ STL containers and iterators. Triangular, symmetric, and Hermitian matrices are stored packed, saving about half the space but making operations slower. For example, it implements spmv rather than symv. (It can use full matrices for triangular solves, to do both trmm and tpmm.) It is not multi-threaded. It doesn't interface to vendor BLAS, although there is an experimental binding to work with ATLAS. There does not appear to be a way to wrap existing matrices and vectors; i.e., they have to be copied into new uBLAS matrices and vectors. Per the uBLAS FAQ, development has stagnated since 2008, so it is missing the latest C++ features, and is not as fast as other libraries. Benchmarks showed it is $12$–$15\times$ slower than sequential MKL for $n = 500$ dgemm (with Linux, Intel Sandy Bridge, Intel icpc and GNU g++ compilers, -O3 -DNDEBUG flags, cold cache).

Here is an example blocked Cholesky algorithm.

```
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/matrix_proxy.hpp>
#include <boost/numeric/ublas/vector_proxy.hpp>
#include <boost/numeric/ublas/triangular.hpp>

```

[4]http://www.boost.org/doc/libs/1_64_0/libs/numeric/ublas/doc/index.html

```cpp
using namespace boost::numeric::ublas;

template< typename T, typename Layout >
int potrf( matrix< T, Layout >& A )
{
    // Assume uplo == lower. This is a left-looking version.
    // Compute the Cholesky factorization A = L*L^H.
    int n = A.size1(), lda = n, nb = 8, info = 0;
    for (int j = 0; j < n; j += nb) {
        // Update and factorize the current diagonal block and test
        // for non-positive-definiteness.
        int jb = std::min( nb, n-j );
        // herk: A(j:j+jb, j:j+jb) -= A(j:j+jb, 0:j) * A(j:j+jb, 0:j)^H
        noalias( project( A, range(j, j+jb), range(j, j+jb) ))
            -= prod(       project( A, range(j, j+jb), range(0, j) ),
                     herm( project( A, range(j, j+jb), range(0, j) )));
        lapack_potrf( 'l', jb, &A(j, j), lda, &info );
        if (info != 0) {
            info += j;
            break;
        }
        if (j+jb < n) {
            // Compute the current block column.
            // gemm: A(j+jb:n, j:j+jb) -= A(j+jb:n, 0:j) * A(j:j+jb, 0:j)^H
            noalias( project( A, range(j+jb, n), range(j, j+jb) ))
                -= prod(       project( A, range(j+jb, n), range(0, j) ),
                         herm( project( A, range(j, j+jb), range(0, j) )));

            // trsm: A(j+jb:n, j:j+jb) = A(j+jb:n, j:j+jb) / A(j:j+jb, j:j+jb)^H  # lower
            //  ==>  A(j+jb:n, j:j+jb)^H = A(j:j+jb, j:j+jb) \ A(j+jb:n, j:j+jb)^H
            // inplace_solve doesn't compile ... don't know why
            // out-of-place solve will create a temporary. sigh.
            project( A, range(j+jb, n), range(j, j+jb) )
                = solve( project( A, range(j, j+jb), range(j, j+jb) ),
                         project( A, range(j+jb, n), range(j, j+jb) ),
                         lower_tag() );
        }
    }
    return info;
}
```

### 2.3.2 MTL4: Matrix Template Library

MTL4[5] is a C++ library that supports dense, banded, and sparse matrices. For dense matrices, it supports row-major (default), column-major, and a Morton recursive layout. It uses parts of Boost, and in fact the default installation puts MTL as a sub-folder of Boost. For sparse matrices, it supports CRS/CSR (compressed row storage/compressed sparse row), CCS/CSC (compressed column storage/compressed sparse column), coordinate, and ELLPACK formats.

Many functions are global functions rather than member functions, for instance, `num_rows(A)` instead of `A.num_rows()`.

There is extensive documentation with numerous example codes. Still, the documentation is somewhat difficult to follow and hard to find how to do things, or what features are exactly supported.

It has native C++ implementations for BLAS operations such as matrix-multiply, so is not limited

[5] http://www.simunova.com/mtl4

to the 4 precisions of traditional BLAS. By defining `MTL_HAS_BLAS`, it will interface to traditional BLAS routines for `gemm`; searching the code it does not appear that other traditional BLAS routines are called. However, benchmarks did not reveal any difference in `dgemm` performance when `MTL_HAS_BLAS` was defined.

It has an MIT open source license, as well as a commercial Supercomputing Edition with parallel and distributed support.

Compared to uBLAS, the syntax for accessing sub-matrices is nicer:

```
dense2D<T> Asub = sub_matrix( A, i1, i2, j1, j2 );
// or
dense2D<T> Asub = A[ irange(i1, i2) ][ irange(j1, j2) ];
```

Like uBLAS, MTL uses expression templates, providing efficient implementations of BLAS operations in a convenient syntax. The syntax is nicer than uBLAS, avoiding the `noalias()` and `prod()` functions. Example calls:

```
C = alpha*A*B;

// gemm: C = alpha A^T B + beta C
C = alpha * trans(A)*B + beta * C;

// gemv
y = alpha*A*x + beta*y;
```

It uses "move semantics" to make returning matrices from functions efficient; i.e., it does a shallow copy when returning matrices. Aliasing of arguments can be an issue; it detects some aliasing and will throw an exception, e.g., in `A = A*B`. But if there is partial overlap, aliasing will not be detected, and must be resolved by the user adding a temporary. (Traditional BLAS will not detect aliasing, either.) It also throws exceptions if matrix sizes are incompatible. Exceptions are disabled if `NDEBUG` is defined.

MTL has triangular-vector solves (`trsv`) available in `upper_trisolve` and `lower_trisolve`, but does not appear to support triangular-matrix solve (`trsm`). This is an impediment to even a simple blocked Cholesky implementation. However, it provides a recursive Cholesky implementation example. It supports recursive algorithms by providing a `mtl::recursator` that divides a matrix into quadrants ($A_{11}$, $A_{12}$, $A_{21}$, $A_{22}$), named `north_west, north_east, south_west, south_east`, respectively.

It has support for symmetric eigenvalue problems, but otherwise it is unclear if it supports operations on symmetric matrices, such as `symm, syrk, syr2k`, etc. Outside of the symmetric eigenvalue problem, there is little mention of symmetric matrices, but there is an `mtl::symmetric` tag.

It includes some matrix solver capabilities:

- LU, with and without pivoting
- QR orthogonalization
- Eigenvalue problems (QR iteration)
- SVD

- ILU(0), IC(0), IMF(s) incomplete LU, Cholesky, and multifrontal sparse solvers

It interfaces with UMFPACK for sparse non-symmetric systems.

It optionally supports some modern C++11 features:

- move semantics (`std::move, std::forward`)
- static asserts (`static_assert`) for compile-time checks of templates (e.g., that a template type is compatible)
- initializer lists: `dense2D<T> A = {{ 3, 4 }, { 5, 6 }};`
- for loops using range: `for (int i : irange(size(v))) { ... }`

Similar to uBLAS, benchmarks showed it is $\approx 14\times$ slower than sequential MKL for $n = 500$ `dgemm` (with Linux, Intel Sandy Bridge, Intel icpc and GNU g++ compilers, -O3 -DNDEBUG flags, cold cache).

Here is an example blocked Cholesky algorithm, except lacking trsm.

```cpp
#include <boost/numeric/mtl/mtl.hpp>

template< typename T, typename Layout >
int potrf( mtl::dense2D< T, Layout >& A )
{
    // Assume uplo == lower. This is a left-looking version.
    // Compute the Cholesky factorization A = L*L^H.
    int n = num_rows(A), lda = n, nb = 8, info = 0;
    for (int j = 0; j < n; j += nb) {
        // Update and factorize the current diagonal block and test
        // for non-positive-definiteness.
        int jb = std::min( nb, n-j );
        // herk: A(j:j+jb, j:j+jb) -= A(j:j+jb, 0:j) * A(j:j+jb, 0:j)^H
        if (j > 0) {  // throws exception on empty matrices
            sub_matrix( A, j, j+jb, j, j+jb )
                -=           sub_matrix( A, j, j+jb, 0, j ) *
                   adjoint( sub_matrix( A, j, j+jb, 0, j ));
        }
        lapack_potrf( 'l', jb, &A(j, j), lda, &info );
        if (info != 0) {
            info += j;
            break;
        }
        if (j+jb < n) {
            // Compute the current block column.
            // gemm: A(j+jb:n, j:j+jb) -= A(j+jb:n, 0:j) * A(j:j+jb, 0:j)^H
            if (j > 0) {
                sub_matrix( A, j+jb, n, j, j+jb )
                    -=           sub_matrix( A, j+jb, n, 0, j ) *
                       adjoint( sub_matrix( A, j, j+jb, 0, j ));
            }
            // trsm: A(j+jb:n, j:j+jb) = A(j+jb:n, j:j+jb) / A(j:j+jb, j:j+jb)^H  # lower
            //  ==>  A(j+jb:n, j:j+jb)^H = A(j:j+jb, j:j+jb) \ A(j+jb:n, j:j+jb)^H
            // This solve doesn't compile: ambiguous (perhaps a bug in their API).
            // Also, only trsv is supported, not trsm.
            // lower_trisolve( sub_matrix( A, j, j+jb, j, j+jb ),
            //                 sub_matrix( A, j+jb, n, j, j+jb ),
            //                 sub_matrix( A, j+jb, n, j, j+jb ) );
        }
    }
    return info;
}
```

### 2.3.3 Eigen

Like uBLAS and MTL4, Eigen[6] is based on C++ Expression Templates. It seems to be a more mature product than uBLAS and MTL4. In addition to BLAS-type expressions, it includes:

- Linear solvers
  - LU with partial pivoting or full pivoting
  - Cholesky, Cholesky with pivoting (for semidefinite)
  - QR, QR with column pivoting (rank revealing), QR with full pivoting
- Eigensolvers
  - Hermitian (“Self Adjoint”)
  - Generalized Hermitian ($Ax = \lambda Bx$ where $B$ is HPD)
  - Nonsymmetric
- SVD
  - 2-sided Jacobi
  - bidiagonalization

It does not have a symmetric indefinite solver such as Bunch-Kaufman pivoting, Rook pivoting, or Aasen’s algorithm.

The syntax for blocks is more succinct than other libraries:
uBLAS: `project( A, range( i, i+mb ), range( j, j+nb ))`
MTL4: `sub_matrix( A, i, i+mb, j, j+nb )`
Eigen: `A.block( i, j, mb, nb )`
However, when using member functions in a template context, it requires extra “template” keywords, which are annoying and clutter the code:
Eigen: `A.template block( i, j, mb, nb )`

It provides both triangular and Hermitian (self-adjoint) views on matrices. It does not appear to offer complex-symmetric views, which are less frequently used but do occur in some applications.

As with uBLAS and MTL, aliasing can be an issue. Component-wise operations where the $C(i, j)$ output entry depends on only the $C(i, j)$ input entry of $C$ and other matrices, are unaffected by aliasing. Some operations like transpose have an in-place version available, and Eigen detects obvious cases of aliasing in debug mode. Like uBLAS, matrix-multiply is assumed to alias, so generates a temporary unless the user adds `.noalias()`.

[6] http://eigen.tuxfamily.org/

Therefore, while it makes simple expressions like `C = A*B` simple, more complex expressions are quickly bogged down by extra function calls (block, triangularView, selfadjointView, solveInPlace, noalias) and C++ syntax.

Eigen has a single class covering both Matrices and Vectors, which also covers both compile-time fixed size (good for small matrices) and runtime dynamic sizes. Either rows or columns can be fixed at compile-time. Default storage is column-wise, but that is a template parameter. It also has an Array class for component-wise operations such as $x. * y$ (in Matlab notation), and an easy conversion between Matrix and Array classes:

```
    VectorXd x(n), y(n);
    double   r = x.transpose() * y;      // dot product
    VectorXd w = x * y;                  // assertion error: invalid matrix product
    VectorXd z = x.array() * y.array();  // component-wise product
```

Incompatible matrix dimensions in matrix-multiply are detected at runtime in debug mode with an assert. Other errors such as aliasing are (sometimes) also detected and execution aborted with an assert. These can be redefined to throw C++ exceptions, if desired.

Eigen supports multi-threading through OpenMP, unlike uBLAS and the open source MTL release. Its performance is better, as well, though still less than vendor-optimized code in MKL. For single-threaded `dgemm`, it is about $2\times$ slower than MKL for size $500$ (compared to $14$–$15\times$ for uBLAS and MTL), while for multi-threaded, it is $2$–$4\times$ slower than MKL (with Linux, 16-core Intel Sandy Bridge, GNU g++ compiler, -O3 -DNDEBUG flags, cold cache). Performance is noticeably worse with Intel icpc.

However, Eigen can directly call BLAS and LAPACK functions, by setting `EIGEN_USE_MKL_ALL`, `EIGEN_USE_BLAS`, or `EIGEN_USE_LAPACKE`. With one of these options, Eigen is between nearly the same as MKL to $2\times$ slower.

Here is an example of Cholesky.

```
#include <Eigen>

template< typename T, int Rows, int Cols, int Layout >
int potrf( Eigen::Matrix< T, Rows, Cols, Layout >& A )
{
    // Assume uplo == lower. This is a left-looking version.
    // Compute the Cholesky factorization A = L*L^H.
    int n = A.rows(), lda = n, nb = 8, info = 0;
    for (int j = 0; j < n; j += nb) {
        // Update and factorize the current diagonal block and test
        // for non-positive-definiteness.
        int jb = std::min( nb, n-j );
        // herk: A(j:j+jb, j:j+jb) -= A(j:j+jb, 0:j) * A(j:j+jb, 0:j)^H
        A.template block( j, j, jb, jb )
         .template selfadjointView< Eigen::Lower >()
         .rankUpdate( A.template block( j, 0, jb, j ), -1.0 );
        lapack_potrf( 'l', jb, &A(j, j), lda, &info );
        if (info != 0) {
            info += j;
            break;
        }
        if (j+jb < n) {
            // Compute the current block column.
            // gemm: A(j+jb:n, j:j+jb) -= A(j+jb:n, 0:j) * A(j:j+jb, 0:j)^H
            A.template block( j+jb, j, n - (j + jb), jb ) -=
                A.template block( j+jb, 0, n - (j + jb), j ) *
                A.template block( j, 0, jb, j ).adjoint();
            // trsm: A(j+jb:n, j:j+jb) = A(j+jb:n, j:j+jb) * A(j:j+jb, j:j+jb)^{-H}  # lower
            A.template block( j, j, jb, jb )
             .template triangularView< Eigen::Lower >().adjoint()
             .template solveInPlace< Eigen::OnTheRight >(
                A.template block( j+jb, j, n - (j + jb), jb ) );
        }
    }
    return info;
}
```

### 2.3.4 Elemental

Elemental[7] is an MPI-based, distributed memory linear algebra library. It includes a C++ interface to BLAS (excluding band and packed formats) and a selection of LAPACK routines. Its BLAS interface is in the `El::blas` namespace, and functions are named after the traditional BLAS routines, minus the precision prefix, in Pascal case. For the standard 4 precisions (single, double, complex-single, complex-double), it calls an optimized BLAS library. It also offers a templated C++ reference implementation for arbitrary numeric datatypes such as **`int`** or **`double-double`**.

Hermitian and symmetric routines are extended to all precisions, e.g., Herk ($C = \alpha AA^H + \beta C$, $C$ is Hermitian) and Syrk ($C = \alpha AA^T + \beta C$, $C$ is symmetric) are both available for both real and complex datatypes. Dot products are also defined for both real and complex. This allows for templated code to use the same name for all datatypes.

Arguments in its wrappers are similar to the traditional BLAS and LAPACK, including options, dimensions, leading dimensions, and scalars. Dimensions use **`int`**; there is experimental support for 64-bit integers. Options are a single char, corresponding to the traditional BLAS, contrasting to CBLAS, which uses enums for options. For instance, a `NoTrans`, `Trans` matrix-matrix multiply ($C = \alpha AB^T + \beta C$) is:

```
    El::blas::Gemm( 'N', 'T', m, n, k, alpha, A, lda, B, ldb, beta, C, ldc );
```

Elemental wraps a handful of LAPACK routines, mainly dealing with eigenvalue and singular value problems. Instead of functions using the LAPACK acronym names (e.g., `syevr`), it uses descriptive English names (`HermitianEig`).

In LAPACK, eigenvalue routines have a job parameter specifying whether to compute eigenvalues only or also eigenvectors. Some routines also have range parameters to specify computing only a portion of the eigen/singular value spectrum. In Elemental's wrappers, these different jobs are provided by overloaded functions, avoiding the need to specify the job parameter and unused dummy arguments:

```
// factor A = Z lambda Z^H, eigenvalues lambda and eigenvectors Z
HermitianEig( uplo, n, A, lda, lambda,                 tol=0 )  // lambda only
HermitianEig( uplo, n, A, lda, lambda, Z, ldz,         tol=0 )  // lambda and Z
HermitianEig( uplo, n, A, lda, lambda,         il, iu, tol=0 )  // il-th to iu-th lambda
HermitianEig( uplo, n, A, lda, lambda, Z, ldz, il, iu, tol=0 )  // il-th to iu-th lambda and Z
HermitianEig( uplo, n, A, lda, lambda,         vl, vu, tol=0 )  // lambda in (vl, vu]
HermitianEig( uplo, n, A, lda, lambda, Z, ldz, vl, vu, tol=0 )  // lambda in (vl, vu] and Z
```

[7] http://libelemental.org/

Elemental also provides wrappers around certain functionality in the MPI, ScaLAPACK, BLACS, PBLAS, libFLAME, and PMRRR libraries.

It throws C++ exceptions (`SingularMatrixException` and `NonHPDMatrixException`) for runtime numerical issues.

Elemental defines a dense matrix class, `Matrix` a distributed-memory matrix class, `DistMatrix`, and sparse matrix classes, `SparseMatrix` and `DistSparseMatrix`. The `Matrix` class is templated on datatype only. It uses column-major LAPACK matrix layout, with a leading dimension that may optionally be explicitly specified, unlike most other C++ libraries reviewed here. A Matrix can also be constructed as a view to an existing memory buffer:

```
    Matrix<double> A( m, n, data, lda );
```

Numerous BLAS, BLAS-like, LAPACK, and other algorithms are defined for Elemental's matrix types. In contrast to the lightweight wrappers described above, the dimensions are implicitly known from matrix objects, rather than being passed explicitly. Options are specified by enums instead of by character values; however, the enums are named differently than in CBLAS. Particularly, it has an `Orientation` enum instead of `Transpose`, with values `El::NORMAL`, `El::TRANSPOSE`, and `El::ADJOINT`, corresponding to `NoTrans`, `Trans`, and `ConjTrans`. In addition to standard BLAS routines, Elemental provides (among others):

| | |
|---|---|
| Adjoint | out-of-place conjugate transpose, $B = A^H$ |
| Axpy | add matrices, $Y = \alpha X + Y$ |
| Broadcast | parallel broadcast |
| DiagonalScale | $X = \text{op}(D)X$ |
| Dot | matrix Hilbert-Schmidt inner product, $\text{vec}(A)^H\text{vec}(B)$ |
| Hadamard | element-wise product, $C = A \circ B$ |
| QuasiTrsm | Schur-form quasi-triangular solve |
| Reduce | parallel reduction |
| Transpose | out-of-place transpose, $B = A^T$ |
| Trrk | Rank-$k$ update limited to triangular portion (e.g., useful for syrk-like update $C = \alpha AB + \beta C$ when $AB$ is known to be symmetric; cf. syrkx in cuBLAS and gemmt in Intel MKL) |
| TwoSidedTrmm | $A = L^H AL$ |
| TwoSidedTrsm | $A = L^{-1}AL^{-H}$ |

In addition to standard LAPACK algorithms, Elemental provides pivoted Cholesky, no-pivoting LU, and complete pivoting LU. It also has a number of other matrix factorizations and applications such as pseudospectra, polar decomposition, matrix square root, and matrix sign function.

The syntax for accessing submatrices is very concise, using the `IR( low, hi )` Integer Range class, which provides the half-open range [low, hi):

```
    Matrix<double> A( m, n );
    auto Asub = A( IR(j, j+jb), IR(j, n) );
```

Because C++ can't take a non-const reference of a temporary, the output submatrix of each call must be a local variable, i.e., one can't write:

```
        El::Herk( El::LOWER, El::NORMAL,
                  -1.0, A( IR(j,j+jb), IR(0,j) ),
                   1.0, A( IR(j,j+jb), IR(j,j+jb) ) );
```

but must instead make the local variable Ajj:

```
        auto Ajj = A( IR(j,j+jb), IR(j,j+jb) );
        El::Herk( El::LOWER, El::NORMAL,
                  -1.0, A( IR(j,j+jb), IR(0,j) ),
                   1.0, Ajj );
```

Here is an example of Cholesky.

```
#include <El.h>

// throws NonHPDMatrixException
template< typename T >
void potrf( El::Matrix<T>& A )
{
    assert( A.Height() == A.Width() );
    int n = A.Height();
    int nb = 8;

    using El::IR;
    // Assume uplo == lower. This is a left-looking version.
    // Compute the Cholesky factorization A = L*L^H.
    for (int j = 0; j < n; j += nb) {
        // Update and factorize the current diagonal block and test
        // for non-positive-definiteness.
        int jb = std::min( nb, n-j );
        // herk: A(j:j+jb, j:j+jb) -= A(j:j+jb, 0:j) * A(j:j+jb, 0:j)^H
        auto Ajj = A( IR(j,j+jb), IR(j,j+jb) );
        El::Herk( El::LOWER, El::NORMAL,
                  -1.0, A( IR(j,j+jb), IR(0,j) ),
                   1.0, Ajj );
        El::Cholesky( El::LOWER, Ajj );
        if (j+jb < n) {
            // Compute the current block column.
            // gemm: A(j+jb:n, j:j+jb) -= A(j+jb:n, 0:j) * A(j:j+jb, 0:j)^H
            auto Acol = A( IR(j+jb,n), IR(j,j+jb) );
            El::Gemm( El::NORMAL, El::ADJOINT,
                      -1.0, A( IR(j+jb,n), IR(0,j) ),
                            A( IR(j,j+jb), IR(0,j) ),
                       1.0, Acol );
            // trsm: A(j+jb:n, j:j+jb) = A(j+jb:n, j:j+jb) * A(j:j+jb, j:j+jb)^{-H}  # lower
            El::Trsm( El::RIGHT, El::LOWER, El::ADJOINT, El::UNIT,
                      1.0, A( IR(j,j+jb), IR(j,j+jb) ),
                           Acol );
        }
    }
}
```

### 2.3.5 Intel DAAL

The Intel *Data Analytics Acceleration Library* (DAAL)[8] provides highly optimized algorithmic building blocks for data analysis, including: preprocessing, transformation, analysis, modeling, validation, etc. DAAL contains routines for, e.g.: principal component analysis, linear regression, classification, clustering, etc. DAAL is designed to handle data that is too big to fit in memory, and instead comes in chunks, which can be referred to as an “out-of-core” mode of operation.

[8]https://software.intel.com/en-us/intel-daal

It is also designed for distributed processing using popular data analytics platforms: Hadoop, Spark, R, and Matlab. DAAL can access data from memory, files, and SQL databases.

DAAL calls BLAS through wrappers, defined as static members of the `Blas` class template. For example, a call to the SYRK function, in the `computeXtX` method of the `ImplicitALSTrainKernelCommon` class, looks like this:

```
#include "service_blas.h"
template <typename algorithmFPType, CpuType cpu>
void computeXtX(size_t *nRows, size_t *nCols, algorithmFPType *beta,
                algorithmFPType *x, size_t *ldx,
                algorithmFPType *xtx, size_t *ldxtx)
{
    char uplo = 'U';
    char trans = 'N';
    algorithmFPType alpha = 1.0;
    Blas<algorithmFPType, cpu>::xsyrk(&uplo, &trans,
                                      (DAAL_INT *)nCols, (DAAL_INT *)nRows,
                                      &alpha, x, (DAAL_INT *)ldx,
                                      beta, xtx, (DAAL_INT *)ldxtx);
}
```

The `service_blas.h` header file contains the definition of the `Blas` class template:

```
#include "service_blas_mkl.h"
template<typename fpType, CpuType cpu, template<typename, CpuType> class _impl=mkl::MklBlas>
struct Blas
{
    typedef typename _impl<fpType,cpu>::SizeType SizeType;
    static void xsyrk(char *uplo, char *trans, SizeType *p, SizeType *n,
                      fpType *alpha, fpType *a, SizeType *lda,
                      fpType *beta, fpType *ata, SizeType *ldata)
    {
        _impl<fpType,cpu>::xsyrk(uplo, trans, p, n, alpha, a, lda, beta, ata, ldata);
    }
```

This relies in turn on the `mkl::MklBlas` class template, defined in `service_blas_mkl.h`, which contains partial specializations of the BLAS routines for double precision:

```
template<CpuType cpu>
struct MklBlas<double, cpu>
{
    typedef DAAL_INT SizeType;
    static void xsyrk(char *uplo, char *trans, DAAL_INT *p, DAAL_INT *n,
                      double *alpha, double *a, DAAL_INT *lda,
                      double *beta, double *ata, DAAL_INT *ldata)
    {
        __DAAL_MKLFN_CALL(blas_, dsyrk, (uplo, trans, p, n, alpha, a, lda, beta, ata, ldata));
    }
```

and for single precision:

```
template<CpuType cpu>
struct MklBlas<float, cpu>
{
    typedef DAAL_INT SizeType;
    static void xsyrk(char *uplo, char *trans, DAAL_INT *p, DAAL_INT *n,
                      float *alpha, float *a, DAAL_INT *lda,
                      float *beta, float *ata, DAAL_INT *ldata)
    {
        __DAAL_MKLFN_CALL(blas_, ssyrk, (uplo, trans, p, n, alpha, a, lda, beta, ata, ldata));
    }
```

The call passes through a couple of macro definitions

```
#define __DAAL_MKLFN_CALL(f_pref,f_name,f_args)  __DAAL_MKLFN_CALL1(f_pref,f_name,f_args)

#define __DAAL_MKLFN_CALL1(f_pref,f_name,f_args)                   \
    if(avx512 == cpu)                                              \
    {                                                              \
        __DAAL_MKLFN(avx512_,f_pref,f_name) f_args;                \
    }                                                              \

#define __DAAL_MKLFN(f_cpu,f_pref,f_name)        __DAAL_CONCAT4(fpk_,f_pref,f_cpu,f_name)

#if !defined(__DAAL_CONCAT4)
    #define __DAAL_CONCAT4(a,b,c,d) __DAAL_CONCAT41(a,b,c,d)
    #define __DAAL_CONCAT41(a,b,c,d) a##b##c##d
#endif
```

before reaching the actual reference to an MKL function, e.g.: `avx512_blas_syrk()`.

Calls to LAPACK are handled in a similar manner. DAAL calls LAPACK through wrappers, defined as static members of the `Lapack` class template. For example, a call to the `POTRF` function, in the `solve` method of the `ImplicitALSTrainKernelBase` class, looks like this:

```
#include "service_lapack.h"
template <typename algorithmFPType, CpuType cpu>
void ImplicitALSTrainKernelBase<algorithmFPType, cpu>::solve(
    size_t *nCols,
    algorithmFPType *a, size_t *lda,
    algorithmFPType *b, size_t *ldb)
{
    char uplo = 'U';
    DAAL_INT iOne = 1;
    DAAL_INT info = 0;
    Lapack<algorithmFPType, cpu>::xxpotrf(&uplo, (DAAL_INT *)nCols,
                                          a, (DAAL_INT *)lda, &info);
```

The `service_lapack.h` header file contains the definition of the `Lapack` class template:

```
#include "service_lapack_mkl.h"
template<typename fpType, CpuType cpu, template<typename, CpuType> class _impl=mkl::MklLapack>
struct Lapack
{
    typedef typename _impl<fpType,cpu>::SizeType SizeType;
    static void xxpotrf(char *uplo, SizeType *p,
                        fpType *ata, SizeType *ldata, SizeType *info)
    {
        _impl<fpType,cpu>::xxpotrf(uplo, p, ata, ldata, info);
    }
```

This relies in turn on the `mkl::MklLapack` class template, defined in `service_lapack_mkl.h`, which contains partial specializations of the LAPACK routines for double precision:

```
template<CpuType cpu>
struct MklLapack<double, cpu>
{
    typedef DAAL_INT SizeType;
    static void xpotrf(char *uplo, DAAL_INT *p, double *ata, DAAL_INT *ldata, DAAL_INT *info)
    {
        __DAAL_MKLFN_CALL(lapack_, dpotrf, (uplo, p, ata, ldata, info));
    }
```

and for single precision:

```
template<CpuType cpu>
struct MklLapack<float, cpu>
```

```
{
    typedef DAAL_INT SizeType;
    static void xpotrf(char *uplo, DAAL_INT *p, float *ata, DAAL_INT *ldata, DAAL_INT *info)
    {
        __DAAL_MKLFN_CALL(lapack_, spotrf, (uplo, p, ata, ldata, info));
    }
```

In summary, DAAL calls BLAS and LAPACK through static member functions of the `Blas` and `Lapack` class templates. Also, DAAL uses the legacy BLAS calling convention (Fortran), were parameters are passed by reference, and there is no parameter to specify the layout (column-major or row-major). Finally, DAAL contains templates only for the BLAS and LAPACK functions that it actually uses. It contains specializations only for single and double precision.

One potential problem with making the datatype a class template parameter is supporting mixed or extended precision – the class has only one datatype, and it is unclear how to extend it to multiple datatypes.

### 2.3.6 Trilinos

Trilinos[9] is a collection of open-source software libraries, called packages, linked together by a common infrastructure, and intended to be used as building blocks for the development of scientific applications. Trilinos was developed at Sandia National Laboratories from a core group of existing algorithms and utilities. Trilinos supports distributed-memory parallel computation through the Message Passing Interface (MPI) and has growing support for shared-memory parallel computation, and also GPUs. This happens by the means of the Kokkos package, which provides a common C++ interface over various parallel programming models, including OpenMP, POSIX Threads, and CUDA.

Trilinos provides two sets of wrappers that interface with BLAS and LAPACK. The more generic interface is contained in the Teuchos package, while a much more concrete implementation is included in the Epetra package. One worthwhile feature of both of these interfaces is that the actual BLAS or LAPACK function call is nearly identical between the two. The only difference is the instantiation of the library object. That object serves as a pseudo namespace for all the subsequent calls to the wrapper functions. See the examples below for more details.

Another shared aspect of both packages is that only the column-major order of matrix elements is supported and no provisions are made for row-major layout.

#### Teuchos

The main package within Trilinos that provides BLAS and LAPACK interface is called Teuchos. More precisely, there are two subpackages that constitute an interface: `Teuchos::BLAS` and `Teuchos::LAPACK`. These two subpackages constitute a rather thin layer on top of the existing linear algebra libraries, especially when compared with the rest of features and software services that Teuchos provides such as memory management, message passing, OS portability and so on.

[9]https://trilinos.org/

The interface is heavily templated. The first two template parameters refer to the numeric data type for matrix/vector elements and the integral type for dimensions. In addition, traits are used throughout Teuchos in a manner similar to the string character traits in the standard C++ library. `MagnitudeType` corresponds to magnitude of scalars with a corresponding trait method `squareroot` that enforces non-negative arguments through the type system. `ScalarType` is used for scalars and its trait methods include `magnitude` and `conjugate`.

In addition to a generic interface and wrappers around low-level BLAS and LAPACK, Teuchos also contains reference implementations of a majority of BLAS routines. The implementations are vector-oriented and unlikely to yield efficient code, but are useful for instantiation of Teuchos for more exotic data types that are not necessarily supported by hardware.

An example code that calls Level 1 BLAS looks like this:

```cpp
#include "Teuchos_BLAS.hpp"
int example(int n, double alpha, double *x, int incx) {
  // instantiate BLAS class for integer dimensions and double-precision numerics
  Teuchos::BLAS<int, double> blas;
  blas.SCAL( n, alpha, x, incx );
  return blas.IAMAX( n, x, incx );
}
```

An example code that invokes dense solver routines for a system of linear equations given by a square matrix is as follows:

```cpp
#include "Teuchos_LAPACK.hpp"
void example(int n, int nrhs, double *A, int ldA, int *piv, double *B, int& info) {
  Teuchos::LAPACK<int, double> lapack;
  lapack.GETRF(n, n, A, ldA, piv, &info);
  lapack.GETRS('N', n, nrhs, A, ldA, piv, B, ldB, &info);
}
```

Note the use of character integral types instead of enumerated types for standard LAPACK enumeration parameters. Also, the error handling requires explicit use of an integral type commonly referred to as `info`.

The LAPACK routines available in the `Teuchos::LAPACK` class are called through member functions that are not inlined:

```cpp
// File Teuchos_LAPACK.hpp
namespace Teuchos {
  template<typename OrdinalType, typename ScalarType>
  class LAPACK
  {
  public:
   void POTRF(const char UPLO, const OrdinalType n,
   ScalarType* A, const OrdinalType lda, OrdinalType* info) const;
  }
}
```

This separates declaration from the implementation and adds additional overhead of non-virtual member call:

```cpp
// File Teuchos_LAPACK.cpp
namespace Teuchos {
void LAPACK<int, float>::POTRF(const char UPLO, const int n,
  float* A, const int lda, int* info) const {
    SPOTRF_F77(CHAR_MACRO(UPLO), &n, A, &lda, info);
  }
}
```

Note that the implementation contains the resolution of name-mangling scheme generated by the FORTRAN 77 compiler. This creates an implicit coupling at link time between Teuchos and LAPACK implementation that depends on the naming scheme. As a result, multiple implementations of the Teuchos LAPACK wrapper must exist for every naming scheme of interest to the user on the target platform.

It solves the problem with the compiler-generated object code growth because the `Teuchos::LAPACK` class is templated with dimension and storage template types for LAPACK matrices, vectors, and scalar. The `Teuchos::LAPACK` class has to be instantiated explicitly. This could be optimized by using static methods but it is only supported in the newer C++ standards. To reduce the overhead of constructing an object of the `Teuchos::LAPACK` class for every calling scope, the user may choose to keep a global object for all calls. But because the constructor is empty and defined in the header file, code inlining would likely eliminate the construction overhead. The similar argument applies to the object destruction with the caveat that the destructor was made virtual which might trigger creation of the *vtable*. This is despite the fact that it is hard to imagine the need for a virtual destructor because deriving from the base `Teuchos::LAPACK` class is unlikely due to the lack of internal state and the fact that the LAPACK interface is stable in syntax and semantics, with only occasional additions of new routines. However, Teuchos contains an additional abstract interface layer that derives from the base `Teuchos::LAPACK` class to accommodate various matrix and vector objects. More concretely, the band, dense, QR, and SPD (symmetric positive definite) solvers derive from the base class to call the specific LAPACK routines' wrappers:

```cpp
namespace Teuchos {
  template<typename OrdinalType, typename ScalarType> class SerialBandDenseSolver
  : public CompObject,
    public Object,
    public BLAS<OrdinalType, ScalarType>,
    public LAPACK<OrdinalType, ScalarType> ;
  template<typename OrdinalType, typename ScalarType> class SerialDenseSolver
  : public CompObject,
    public Object,
    public BLAS<OrdinalType, ScalarType>,
    public LAPACK<OrdinalType, ScalarType> ;
  template<typename OrdinalType, typename ScalarType> class SerialQRDenseSolver
  : public CompObject,
    public Object,
    public BLAS<OrdinalType, ScalarType>,
    public LAPACK<OrdinalType, ScalarType> ;
  template<typename OrdinalType, typename ScalarType> class SerialSpdDenseSolver
  : public CompObject,
    public Object,
    public BLAS<OrdinalType, ScalarType>,
    public LAPACK<OrdinalType, ScalarType> ;
}
```

These derived classes contain generic methods for factorization, solving-with-factors, and inversion implemented by `factor()`, `solve()`, and `invert()` methods, respectively. Additional methods may include equilibration, error estimation, and conditioning estimation.

For completeness, it should be mentioned that Teuchos includes additional objects and functions that could be used to perform linear algebra operations. This additional interface layer is above the level of abstraction that is the aim of this document. An example code that calls a linear solve is shown below:

```cpp
#include "Teuchos_SerialDenseMatrix.hpp"
```

```cpp
#include "Teuchos_SerialDenseSolver.hpp"
#include "Teuchos_RCP.hpp" // reference-counted pointer
#include "Teuchos_Version.hpp"

void example(int n) {
  Teuchos::SerialDenseMatrix<int,double> A(n, n);
  Teuchos::SerialDenseMatrix<int,double> X(n,1), B(n,1);
  Teuchos::SerialDenseSolver<int,double> solver;
  solver.setMatrix( Teuchos::rcp( &A, false ) );
  solver.setVectors( Teuchos::rcp( &X, false ), Teuchos::rcp( &B, false ) );

  A.random();
  X.putScalar(1.0); // set X to all 1's
  B.multiply( Teuchos::NO_TRANS, Teuchos::NO_TRANS, 1.0, A, X, 0.0 );
  X.putScalar(0.0); // set X to all 0's

  info = solver.factor();
  info = solver.solve();
}
```

### Epetra

Epetra abbreviates “essential Petra” – the foundational functionality of Trilinos that aims, above all, for portability across hardware platforms and compiler versions. As such, Epetra shuns the use of templates and thus its code is much closer to hardware and implementation artifacts.

Complex-valued matrix elements are not supported by either `Epetra_BLAS` or `Epetra_LAPACK` – only single and double precision real interface are provided.

An example

```cpp
#include <Epetra_BLAS.h>
void example(int n, float *fx, double *dx, int inc, float& fsum, double& dsum) {
  Epetra_BLAS() blas;
  fsum = blas.ASUM(n, fx, inc);
  dsum = blas.ASUM(n, dx, inc);
}
```

An example code that invokes dense solver routines for a system of linear equations given by a square matrix is as follows:

```cpp
#include <Epetra_LAPACK.h>
void example(int n, int nrhs, double *A, int ldA, int *piv, double *B, int& info) {
  Epetra_LAPACK() lapack;
  lapack.GETRF(n, n, A, ldA, piv, &info);
  lapack.GETRS('N', n, nrhs, A, ldA, piv, B, ldB, &info);
}
```

# CHAPTER 3

# C++ API Design

## 3.1 Stateless Interface

The interface shall be stateless, with any implementation-specific setting handled outside of this interface. Initialization and library cleanup will be performed with calls that are specific to the BLAS and LAPACK implementations if any such operations are required.

**Rationale:** It is possible to include the state in the layer of the C++ interface which could then be manipulated with calls not available in the original BLAS and LAPACK libraries. However, this was decided against as this creates confusion when the same call with the same call arguments behaves differently due to the hidden state. The only way for the user to ensure consistent behavior for every call would be to switch the internal state to the desired setting. And there would still remain the issue of threaded and asynchronous calls that could alter the internal state in-between the state reset and, for example, the factorization call.

## 3.2 Supported BLAS and LAPACK Storage Types

In order to support templated algorithms, BLAS and LAPACK need to have precision-independent names, for instance `gemm` instead of `sgemm, dgemm, cgemm, zgemm`. This will also provide future compatibility with mixed and extended precisions, where the arguments have different precisions, as proposed by the Next Generation BLAS (Section 2.2.5). A further goal is to make function calls consistent across all data types, resolving any differences that currently exist.

Our C++ API defines a set of overloaded wrappers that call the traditional vendor-optimized

BLAS and LAPACK routines. Our initial implementation focuses on full matrices (“ge”, “sy”, “he”, “tr” prefixes). It is readily extendable to band (“gb”, “sb”, “hb”) and packed (“sp”, “hp”, “tp”) matrices.

## 3.3 Acceptable Constructs and C++ Language Standard

The C++ language standard has a long history, which results in practical considerations that we try to adapt in this document. In short, the very latest version of the standard is rarely implemented across the majority of compilers and supporting tools. Consequently, it is wise to restrict the range of constructs and limit the syntax in a working code to a subset of one of the standard versions. Accordingly, we will use only the features from the C++11 standard due to its wide acceptance by the software we use and on the hardware platforms we target.

## 3.4 Naming conventions

C++ interfaces to BLAS routines and associated constants are in the `blas` namespace. They are made available by including the `blas.hh` header:

```
#include <blas.hh>

using namespace blas;
```

C++ interfaces to LAPACK routines are in the `lapack` namespace. They are made available by including the `lapack.hh` header:

```
#include <lapack.hh>

using namespace lapack;
```

Most C++ routines are named the same as in traditional BLAS and LAPACK, sans precision, and all lowercase, e.g., `blas::gemm`, `lapack::posv`. Arguments are named the same as in BLAS and LAPACK. In general, matrices are uppercase, e.g., `A`, `B`, vectors are lowercase, e.g., `x`, `y`, scalars are lower case Greek letters spelled out in English, e.g., `alpha`, `beta`, following common math notation.

**Rationale:** Lowercase namespace convention was chosen per usage in standard libraries (`std namespace`), Boost (`boost namespace`), and other common use cases such as the Google style guide. For C++-only headers, the file extension `.hh` was chosen to distinguish it from C-only `.h` headers. This goes against some HPC libraries such as Kokkos and Trilinos that capitalize the first letter, but this naming does not fit any of the standards that are followed in our software.

## 3.5 Real vs. Complex Routines: the Case for Unified Syntax

Some routines in the traditional BLAS have different names for real and complex matrices, for instance `herk` for complex Hermitian matrices and `syrk` for real symmetric matrices. This prevents templating algorithms for both real and complex matrices, so in these cases, both names are extended to apply to both real and complex matrices. For real matrices, `herk` and `syrk` are synonyms, both meaning $C = \alpha AA^H + \beta C = \alpha AA^T + \beta C$, where $C$ is symmetric. For complex matrices, `herk` means $C = \alpha AA^H + \beta C$, where $C$ is complex Hermitian, while `syrk` means $C = \alpha AA^T + \beta C$, where $C$ is complex symmetric. Some complex-symmetric routines such as `csymv` and `csyr` are not in the traditional BLAS standard, but are provided by LAPACK. Some complex-symmetric routines are missing from BLAS and LAPACK, such as `[cz]syr2`, which can be performed using `[cz]syr2k`, albeit suboptimally. We provide all these routines in C++ BLAS, for consistency. LAPACK routines prefixed with `sy` and `he` are handled similarly.

The dot product has different names in real and complex. We extend `dot` to mean `dotc` in complex, and extend `dotc` and `dotu` both to mean `dot` in real.

Additionally in LAPACK, the `un` prefix denotes a complex unitary matrix and the `or` prefix denotes a real orthogonal matrix. For these cases, we extend the `un`-prefixed names to real matrices. The term “orthogonal” is not applicable to complex matrices, so `or`-prefixed routines apply only to real matrices.

Mapping of C++ generic name to traditional BLAS names:

| C++ name | real | complex |
|---|---|---|
| `blas::hemv` | `[sd]symv` | `[cz]hemv` |
| `blas::symv` | `[sd]symv` | `[cz]symv` † |
| `blas::her` | `[sd]syr` | `[cz]her` |
| `blas::syr` | `[sd]syr` | `[cz]syr` † |
| `blas::her2` | `[sd]syr2` | `[cz]her2` |
| `blas::syr2` | `[sd]syr2` | `[cz]syr2` ‡ |
| `blas::herk` | `[sd]syrk` | `[cz]herk` |
| `blas::syrk` | `[sd]syrk` | `[cz]syrk` |
| `blas::her2k` | `[sd]syr2k` | `[cz]her2k` |
| `blas::syr2k` | `[sd]syr2k` | `[cz]syr2k` |
| `blas::hemm` | `[sd]symm` | `[cz]hemm` |
| `blas::symm` | `[sd]symm` | `[cz]symm` |
| `blas::dot` | `[sd]dot` | `[cz]dotc` |
| `blas::dotc` | `[sd]dot` | `[cz]dotc` |
| `blas::dotu` | `[sd]dot` | `[cz]dotu` |

†`[cz]symv` and `[cz]syr` provided by LAPACK instead of BLAS.
‡`[cz]syr2` not available; can substitute `[cz]syr2k` with $k = 1$.

Mapping of C++ generic name to traditional LAPACK names (incomplete list):

| C++ name | real | complex |
|---|---|---|
| `lapack::hesv` | `[sd]sysv` | `[cz]hesv` |
| `lapack::sysv` | `[sd]sysv` | `[cz]sysv` |
| `lapack::unmqr` | `[sd]ormqr` | `[cz]unmqr` |
| `lapack::ormqr` | `[sd]ormqr` | — |
| `lapack::ungqr` | `[sd]orgqr` | `[cz]ungqr` |
| `lapack::orgqr` | `[sd]orgqr` | — |

Where applicable, options that apply conjugate-transpose in complex are interpreted to apply transpose in real. For instance, in LAPACK’s `zlarfb`, `trans` takes `NoTrans` and `ConjTrans` but not

Trans, while in dlarfb it takes NoTrans and Trans but not ConjTrans. We extend this to allow ConjTrans in the real case to mean Trans. This is already true for BLAS routines such as dgemm, where ConjTrans and Trans have the same meaning.

In LAPACK, for non-symmetric eigenvalues, dgeev takes a split complex representation with two double-precision vectors for eigenvalues, one vector for real components, one for imaginary components, while zgeev takes single vector of complex values. In C++, geev follows the complex routine in taking a single vector of complex values in both the real and complex cases.

Other instances where there are differences between real and complex matrices will be resolved to provide a consistent interface across all data types.

## 3.6 Use of const Specifier

Array arguments (matrices and vectors) that are read-only are declared **const** in the interface. Dimension-related and scalar arguments are passed by value, so are not declared **const** as there is no benefit at the call site.

## 3.7 Enum constants

As in CBLAS, options such as transpose, uplo (upper-lower), etc. are provided by enums. Strongly typed C++11 enums are used. Constants have similar names to those in CBLAS, minus the Cblas prefix, but the value is left unspecified and implementation dependent. Enums and constants are Title Case.

Enums for BLAS (values for example only; see implementation note below):

```
enum class Layout : char { ColMajor='C', RowMajor='R' };
enum class Op     : char { NoTrans ='N', Trans   ='T', ConjTrans='C' };
enum class Uplo   : char { Upper   ='U', Lower   ='L' };
enum class Diag   : char { NonUnit ='N', Unit    ='U' };
enum class Side   : char { Left    ='L', Right   ='R' };
```

Note CBLAS_ORDER was renamed CBLAS_LAYOUT around LAPACK 3.6.0.

In most cases, the name of the enum is also similar to the name in CBLAS. However, for transpose, because Transpose::NoTrans could easily be misread as *transposed*, rather than *not transposed*, the enum is named Op, which is already frequently used in the documentation, such as for zgemm:

```
    TRANSA = 'N' or 'n',  op( A ) = A.
    TRANSA = 'T' or 't',  op( A ) = A^T.
    TRANSA = 'C' or 'c',  op( A ) = A^H.
```

In some cases, BLAS and LAPACK take identical options such as uplo. For consistency within each library, typedef aliases for the five BLAS enums above are provided, such as blas::Uplo and lapack::Uplo.

For some routines, LAPACK supports a wider set of values for an **enum** category than provided by BLAS. For instance, in BLAS, uplo = Lower or Upper, while in LAPACK, laset and lacpy take `uplo = Lower, Upper`, or `General`; and `lascl` takes 8 different matrix types. Instead of having an extended **`enum`**, the C++ API consistently uses the standard prefixes (`ge, he, tr`, etc.) to indicate the matrix type, rather than using the `la` auxiliary prefix and differentiating matrix types based on an argument. Thus, these new names are introduced, with their mapping to LAPACK names:

| C++ API | LAPACK | matrix type |
|---|---|---|
| `gescl` | `lascl` with type=G | general |
| `trscl( uplo )` | `lascl` with type=uplo | triangular or Hermitian |
| `gbscl` | `lascl` with type=Z | general band |
| `hbscl( uplo )` | `lascl` with type=B (Lower) or Q (Upper) | Hermitian band |
| `hsscl` | `lascl` with type=H | Hessenberg |
| `gecpy` | `lacpy` with uplo=G | general |
| `trcpy( uplo )` | `lacpy` with same uplo | triangular or Hermitian |
| `geset` | `laset` with uplo=G | general |
| `trset( uplo )` | `laset` with same uplo | triangular or Hermitian |

**Implementation note:** 3 potential implementations are readily apparent. Enumeration values could be:

1. Default values (0, 1, ...). This is used by cuBLAS.
2. Same value as in CBLAS, e.g., `NoTrans = 111`.
3. Character values used in Fortran, e.g., `NoTrans = 'n'` (as shown above).

If the C++ API calls Fortran BLAS, the first two options require a switch, **`if`**`-then`, or lookup table to determine the equivalent character constant (e.g., `NoTrans=111` maps to `'n'`). The third option is trivially converted using a cast, and is easier to understand if printed out for debugging.

If the C++ API calls CBLAS, obviously option 2 is the easiest.

If the C++ API calls some other BLAS library such as cuBLAS or clBLAS, a **`switch`**, **`if`**`-then`, or lookup table is probably required in all three cases.

We leave the enumeration values unspecified and implementation-dependent.

**Rationale:** In C++, the old style enumeration type, that was borrowed from C, is of integral type without exact size specified. This may cause problems for binary interfaces when the C compiler used the default **`int`** representation and C++ compiler use a different storage size. We do not face this issue here as we only target C++ as the calling language and C or Fortran as the likely implementation language.

## 3.8 Workspaces

Many LAPACK routines take workspaces, with both minimum and optimal sizes. These are typically of size $O(n \times n_b)$, for a matrix of dimension $n$ and an optimal block size $n_b$. Notable exceptions are eigenvalue and singular value routines, which often take workspaces of size $O(n^2)$.

As memory allocation is typically a minor amount of time, the C++ LAPACK interface allocates optimal workspace sizes internally, removing workspaces from the interface. Traditional BLAS routines do not take workspaces.

If this becomes a performance bottleneck, workspaces could be added as optional arguments, with a default value of `nullptr` indicating that the wrapper should allocate workspace, without breaking code written with the C++ LAPACK API.

**Rationale:** As needed, there is a possibility of adding an overloaded function call that takes a user-defined memory allocator as an argument. This may serve memory-constrained implementations that insist on controlled memory usage.

## 3.9 Errors

Traditional BLAS routines call `xerbla` when an error occurs, such as `lda < m`. All errors that BLAS detects are bugs. LAPACK likewise calls `xerbla` for invalid parameters (which are bugs), but not for runtime numerical errors like a singular matrix in `getrf` or an indefinite matrix in `potrf`. The default implementation of `xerbla` aborts execution.[1]

Instead, we adopt C++ exceptions for errors, such as invalid arguments. Two new exceptions are introduced: `blas::error` and `lapack::error`, which are subclasses of `std::exception`. The `what()` member function yields a description of the error.

For runtime numerical errors, the traditional info value is returned. Zero indicates success. Note these are often not fatal errors: an application may want to know whether a matrix is positive definite, and the easiest, fastest test is to attempt Cholesky factorization.

We do not implement NaN or Inf checks. These add $O(n^2)$ work and memory traffic, with little added benefit. Ideally, a robust BLAS library would ensure that NaN and Inf values are propagated, meaning that if there is a NaN or Inf in the input, there is one in the output. (Aside from optimizations when `alpha=0` or `beta=0`. In `gemm`, for instance, if `beta=0`, then it is specifically documented in the reference BLAS that C need not be initialized.) The current reference BLAS implementation does not always propagate NaN and Inf; see the Next Generation BLAS (Section 2.2.5) for examples and proposed new routines that are guaranteed to propagate NaN and Inf values.

**Rationale:** Occasionally, users express concern about the overhead of error checks. For even modestly sized matrices, error checks take negligible time. However, for very small matrices, with $n < 20$ or so, there can be noticeable overhead. Intel introduced `MKL_DIRECT_CALL` to disable error checks in these cases[2]. However, libraries compiled for specific sizes, either via templating or JIT compilation, provide an even larger performance boost for these small sizes. For instance, see Intel's libxsmm[3] for extra small matrix-multiply, or batched BLAS for sets of small matrices. Thus users with such small matrices are encouraged to use special purpose interfaces, rather than trying to optimize overheads in a general purpose interface.

[1]See explanation in Batched BLAS document why xerbla is a hideous monstrosity for parallel codes or multiple libraries.

[2]https://software.intel.com/en-us/articles/improve-intel-mkl-performance-for-small-problems-the-use-of-mkl-direct-call

[3]https://github.com/hfp/libxsmm

## 3.10 Return values

Most C++ BLAS routines are void. The exceptions are `asum`, `nrm2`, `dot*`, and `iamax`, which return their result, as in the traditional Fortran interface. `dot` returns a complex value in the complex case (unlike CBLAS, where the complex result is an output argument). This makes the interface consistent across real and complex data types.

Most C++ LAPACK routines return an integer status code, corresponding to positive info values in LAPACK, indicating numerical errors such as a singular matrix in `getrf`. Zero indicates success. LAPACK norm functions return their result.

## 3.11 Complex numbers

C++ `std::complex` is used. Unlike CBLAS, complex scalars are passed by value, the same as real scalars. This avoids inconsistencies that would prevent templated code from calling BLAS. For type safety, arguments are specified as `std::complex`, rather than as `void*` as CBLAS uses.

## 3.12 Object Dimensions as 64-bit Integers

The interface will require 64-bit integers to specify object sizes using `cstdint` header and `int64_t` integral data type.

In the recent years, 32-bit software has been in decline with both vendors and open source projects dropping support for 32-bit versions and opting exclusively for 64-bit only. This has also taken place in HPC software and there are examples of 64-bit vendors, for example Intel with MKL, MathWorks with MATLAB. In fact, 32-bit version is more of a legacy issue with the increasing memory sizes and the demand of larger models that require large matrices and vectors.

BLAS and LAPACK libraries can easily address this because sizing dense matrices and vectors has negligible cost. Even on a 32-bit systems, an overhead of using 64-bit integers is not an issue with the exception of storage for pivots, which arises in LU and pivoted QR as well as accompanying routines that operate on these pivots such as `laswp`. The overhead for those could be $\mathcal{O}(n)$ where $n$ is the number of swapped rows.

## 3.13 Matrix Layout

Traditional Fortran BLAS assumes column-major matrices. CBLAS added support for row-major matrices. In many cases, this can be accomplished with essentially no overhead by swapping matrices, dimensions, upper-lower, and transposes, and then calling the column-major routine. For instance, `cblas_dgemv` simply changes `trans=NoTrans` into `Trans`, or `trans=Trans` into `NoTrans`, swaps `m <=> n`, and calls (column-major) `dgemv`. However, some routines require a little extra effort for complex matrices. For `cblas_zgemv`, `trans=ConjTrans` can be changed to

`NoTrans`, but then the matrix isn't conjugated. This can be resolved by conjugating $y$ and a copy of $x$, calling `zgemv` with `m <=> n` swapped and `trans=NoTrans`, then conjugating $y$ again. Several other Level 2 BLAS routines have similar solutions. So, with minimal overhead, row-major matrices can be supported in BLAS.

We propose the same mechanism for the C++ BLAS API, either by calling CBLAS and relying on the row-major support in CBLAS, or by reimplementing similar solutions in C++ and calling the Fortran BLAS.

We also build the same option into the C++ LAPACK API, for future support. However, initially it would be unimplemented, causing an exception to be thrown. This is because for some routines such `getrf` there can be substantial overhead in calling the traditional Fortran LAPACK implementation, because a transpose is required. Other routines such as matrix norms, QR, LQ, SVD, and operations on symmetric matrices can readily be translated to LAPACK calls with essentially no overhead, without physically transposing the matrix in memory.

Row-major layout is specified the same as in CBLAS, using the `blas::Layout` or `lapack::Layout` enum as the first parameter of C++ BLAS and LAPACK functions. (Perhaps it should be moved to the end to make it an optional argument with default value ColMajor.)

## 3.14 Templated versions

As a future extension, in addition to overloaded wrappers around traditional BLAS routines, generic templated versions that work for any data type could be provided. For instance, these would support half precision, double-double or quad precision, and integer types. The data types need only basic arithmetic operations (+ - * /) and functions (e.g., conj, sqrt, abs, real, imag) to be defined. Initially, such templated versions could be based on the reference BLAS, but these can be optimized using well-known techniques such as blocking and vectorization.

## 3.15 Prototype implementation

To make our proposal concrete, we include a prototype implementation of wrappers for `blas::gemm` matrix-matrix multiply and `lapack::potrf` Cholesky factorization. For brevity, only the `complex<double>` datatype is shown; code for other precisions is analogous. The only compile-time parameters are the Fortran name-mangling convention (here assumed to be lowercase with appended underscore, "_") and `BLAS_ILP64`, which indicates whether it will be linked with an ILP64 (64-bit integer) BLAS/LAPACK library version.

### blas.hh

```
#ifndef BLAS_HH
#define BLAS_HH

#include <cstdint>
#include <exception>
#include <complex>
```

```
#include <string>

namespace blas {

// -----------------------------------------------------------------------------
// Fortran name mangling depends on compiler, generally one of:
//     UPPER
//     lower
//     lower ## _
#ifndef FORTRAN_NAME
#define FORTRAN_NAME( lower, UPPER ) lower ## _
#endif

// -----------------------------------------------------------------------------
// blas_int is the integer type of the underlying Fortran BLAS library.
// BLAS wrappers take int64_t and check for overflow before casting to blas_int.
#ifdef BLAS_ILP64
typedef long long blas_int;
#else
typedef int blas_int;
#endif

// -----------------------------------------------------------------------------
enum class Layout : char { ColMajor='C', RowMajor='R' };
enum class Op     : char { NoTrans ='N', Trans   ='T', ConjTrans='C' };
enum class Uplo   : char { Upper   ='U', Lower   ='L' };
enum class Diag   : char { NonUnit ='N', Unit    ='U' };
enum class Side   : char { Left    ='L', Right   ='R' };

// -----------------------------------------------------------------------------
class Error: public std::exception
{
public:
    Error(): std::exception() {}
    Error( const char* msg ): std::exception(), msg_( msg ) {}
    virtual const char* what() { return msg_.c_str(); }
private:
    std::string msg_;
};

// -----------------------------------------------------------------------------
// internal helper function; throws Error if cond is true
// called by throw_if_ macro
inline void throw_if__( bool cond, const char* condstr )
{
    if (cond) {
        throw Error( condstr );
    }
}

// internal macro to get string #cond; throws Error if cond is true
#define throw_if_( cond ) \
        throw_if__( cond, #cond )

// -----------------------------------------------------------------------------
// Fortran prototypes
// sgemm, dgemm, cgemm omitted for brevity
#define f77_zgemm FORTRAN_NAME( zgemm, ZGEMM )

extern "C"
void f77_zgemm( char const* transA, char const* transB,
                blas_int const* m, blas_int const* n, blas_int const* k,
                std::complex<double> const* alpha,
                std::complex<double> const* A, blas_int const* lda,
                std::complex<double> const* B, blas_int const* ldb,
                std::complex<double> const* beta,
                std::complex<double>*       C, blas_int const* ldc );

// ---------------------------------------------------------------------------------
// lightweight overloaded wrappers: converts C to Fortran calling convention.
// calls to sgemm, dgemm, cgemm omitted for brevity
inline void gemm_( char transA, char transB,
                   blas_int m, blas_int n, blas_int k,
                   std::complex<double> alpha,
                   std::complex<double> const* A, blas_int lda,
                   std::complex<double> const* B, blas_int ldb,
                   std::complex<double> beta,
                   std::complex<double>*       C, blas_int ldc )
{
    f77_zgemm( &transA, &transB, &m, &n, &k,
               &alpha, A, &lda, B, &ldb, &beta, C, &ldc );
}

// ---------------------------------------------------------------------------------
// templated wrapper checks arguments, handles row-major to col-major translation
template< typename T >
void gemm( Layout layout, Op transA, Op transB,
           int64_t m, int64_t n, int64_t k,
           T alpha,
           T const* A, int64_t lda,
           T const* B, int64_t ldb,
           T beta,
           T*       C, int64_t ldc )
{
    // determine minimum size of leading dimensions
    int64_t Am, Bm, Cm;
    if (layout == Layout::ColMajor) {
        Am = (transA == Op::NoTrans ? m : k);
        Bm = (transB == Op::NoTrans ? k : n);
        Cm = m;
    }
    else {
        // RowMajor
        Am = (transA == Op::NoTrans ? k : m);
        Bm = (transB == Op::NoTrans ? n : k);
        Cm = n;
    }

    // check arguments
    throw_if_( layout != Layout::RowMajor && layout != Layout::ColMajor );
    throw_if_( transA != Op::NoTrans && transA != Op::Trans && transA != Op::ConjTrans );
    throw_if_( transB != Op::NoTrans && transB != Op::Trans && transB != Op::ConjTrans );
    throw_if_( m < 0 );
    throw_if_( n < 0 );
    throw_if_( k < 0 );
    throw_if_( lda < Am );
    throw_if_( ldb < Bm );
    throw_if_( ldc < Cm );

    // check for overflow in native BLAS integer type, if smaller than int64_t
    if (sizeof(int64_t) > sizeof(blas_int)) {
        throw_if_( m   > std::numeric_limits<blas_int>::max() );
        throw_if_( n   > std::numeric_limits<blas_int>::max() );
        throw_if_( k   > std::numeric_limits<blas_int>::max() );
        throw_if_( lda > std::numeric_limits<blas_int>::max() );
        throw_if_( ldb > std::numeric_limits<blas_int>::max() );
        throw_if_( ldc > std::numeric_limits<blas_int>::max() );
    }

    if (layout == Layout::ColMajor) {
        gemm_( (char) transA, (char) transB,
               (blas_int) m, (blas_int) n, (blas_int) k,
               alpha,
               A, (blas_int) lda,
               B, (blas_int) ldb,
               beta,
               C, (blas_int) ldc );
    }
    else {
        // RowMajor: swap (transA, transB), (m, n), and (A, B)
        gemm_( (char) transB, (char) transA,
               (blas_int) n, (blas_int) m, (blas_int) k,
               alpha,
               B, (blas_int) ldb,
               A, (blas_int) lda,
               beta,
               C, (blas_int) ldc );
    }
}

}  // namespace blas

#endif        //  #ifndef BLAS_HH
```

## lapack.hh

```
#ifndef LAPACK_HH
#define LAPACK_HH

#include <cstdint>
#include <exception>
#include <complex>

#include "blas.hh"

namespace lapack {

// assume same int_type as BLAS
typedef blas::int_type int_type;

// alias types from BLAS
typedef blas::Layout Layout;
typedef blas::Op     Op;
typedef blas::Uplo   Uplo;
typedef blas::Diag   Diag;
typedef blas::Side   Side;

// omitted for brevity: lapack::Error, throw_if_ similar to blas.hh

// --------------------------------------------------------------------------------
// Fortran prototypes
// spotrf, dpotrf, cpotrf omitted for brevity
#define f77_zpotrf FORTRAN_NAME( zpotrf, ZPOTRF )

extern "C"
void f77_zpotrf( char const* uplo, int_type const* n,
                 std::complex<double>* A, int_type const* lda,
                 int_type* info );

// --------------------------------------------------------------------------------
// lightweight overloaded wrappers: converts C to Fortran calling convention.
// calls to spotrf, dpotrf, cpotrf omitted for brevity
inline void potrf_( char uplo, int_type n,
                    std::complex<double>* A, int_type lda,
                    int_type* info )
{
    f77_zpotrf( &uplo, &n, A, &lda, info );
}

// ---------------------------------------------------------------------------
// templated wrapper checks arguments, handles row-major to col-major translation
template< typename T >
int64_t potrf( Layout layout, Uplo uplo, int64_t n, T* A, int64_t lda )
{
    // check arguments
    throw_if_( layout != Layout::RowMajor && layout != Layout::ColMajor );
    throw_if_( uplo != Uplo::Upper && uplo != Uplo::Lower );
    throw_if_( n < 0 );
    throw_if_( lda < n );

    // check for overflow in native BLAS integer type, if smaller than int64_t
    if (sizeof(int64_t) > sizeof(int_type)) {
        throw_if_( n   > std::numeric_limits<int_type>::max() );
        throw_if_( lda > std::numeric_limits<int_type>::max() );
    }

    int_type info = 0;
    if (layout == Layout::ColMajor) {
        potrf_( (char) uplo, n, A, lda, &info );
    }
    else {
        // RowMajor: change upper <=> lower; no need to conjugate
        Uplo uplo_swap = (uplo == Uplo::Lower ? Uplo::Upper : Uplo::Lower);
        potrf_( (char) uplo_swap, (int_type) n, A, (int_type) lda, &info );
    }
    return info;
}

}  // namespace lapack

#endif        //  #ifndef LAPACK_HH
```