C++ API for BLAS and LAPACK

Mark Gates ICL
Piotr Luszczek ICL
Jakub Kurzak ICL
Jack Dongarra ICL
Konstantin Arturov Intel
Cris Cecka NVIDIA
Chip Freitag AMD

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

January 30, 2018
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.

<table>
<thead>
<tr>
<th>Revision</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>06-2017</td>
<td>first publication</td>
</tr>
<tr>
<td>01-2018</td>
<td>new cover</td>
</tr>
</tbody>
</table>

@techreport{gates2017cpp,
  author={Gates, Mark and Luszczek, Piotr and Kurzak, Jakub and Dongarra, Jack and Arturov, Konstantin and Cecka, Cris and Freitag, Chip},
  title={{SLATE} Working Note 2: C++ {API} for {BLAS} and {LAPACK}},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2017},
  month={June},
  number={ICL-UT-17-03},
  note={revision 06-2017} }
## Contents

1 Introduction and Motivation

2 Standards and Trends
   2.1 Programming Language Fortran .................................................. 2
      2.1.1 FORTRAN 77 ................................................................. 2
      2.1.2 BLAST XBLAS .............................................................. 4
      2.1.3 Fortran 90 ................................................................. 4
   2.2 Programming Language C .......................................................... 5
      2.2.1 Netlib CBLAS ............................................................... 5
      2.2.2 Intel MKL CBLAS ............................................................ 6
      2.2.3 Netlib lapack_cwrapper .................................................. 6
      2.2.4 LAPACKE ................................................................. 7
      2.2.5 Next-Generation BLAS: “BLAS G2” .................................... 8
   2.3 C++ Programming Language ....................................................... 9
      2.3.1 Boost and uBLAS ............................................................. 9
      2.3.2 Matrix Template Library: MTL 4 ....................................... 11
      2.3.3 Eigen ....................................................................... 13
      2.3.4 Elemental ................................................................. 15
      2.3.5 Intel DAAL ............................................................... 18
      2.3.6 Trilinos ................................................................. 21

3 C++ API Design .............................................................................. 25
   3.1 Stateless Interface ................................................................. 25
   3.2 Supported Storage Types ....................................................... 25
   3.3 Acceptable Language Constructs .............................................. 26
   3.4 Naming conventions ............................................................. 26
   3.5 Real vs. Complex Routines .................................................... 26
   3.6 Use of const Specifier .......................................................... 28
   3.7 Enum constants ....................................................................... 28
   3.8 Workspaces ............................................................................. 29
3.9 Errors ................................................................. 30
3.10 Return values ....................................................... 31
3.11 Complex numbers .................................................. 31
3.12 Object Dimensions as 64-bit Integers ......................... 31
3.13 Matrix Layout ....................................................... 31
3.14 Templated versions ................................................. 32
3.15 Prototype implementation ....................................... 32
The Basic Linear Algebra Subprograms¹ (BLAS) and the Linear Algebra PACKage² (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 Software for Linear Algebra Targeting Exascale (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 is specified in the routine name—`sgemm` for single, `dgemm` for double, `cgemm` for complex-single, and `zgemm` for complex-double. 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 to run on top of the existing BLAS and LAPACK libraries.

We start with a survey of traditional BLAS and LAPACK libraries, with 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.

¹http://www.netlib.org/blas/
²http://www.netlib.org/lapack/
CHAPTER 2

Standards and Trends

2.1 Programming Language Fortran

2.1.1 FORTRAN 77

The original FORTRAN \footnote{FORTRAN refers to FORTRAN 77 and earlier standards. The capitalized spelling has since been abandoned, and first-letter-capitalized spelling is now preferred and used uniformly throughout the standard documents.} BLAS first proposed level-1 BLAS routines for vector 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 the FORTRAN 77 naming scheme’s six-letter character limit. The prefix denotes the precision, like so:

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

For level-2 BLAS and level-3 BLAS, a two-letter combination denotes the type of matrix, like so:
2.1. PROGRAMMING LANGUAGE FORTRAN  

CHAPTER 2. STANDARDS AND TRENDS

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:

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

So, for example, \texttt{dgemm} would be a double-precision, general matrix-matrix multiply.

The original Fortran interfaces had a number of limitations, listed below.

- **Portability issues calling Fortran**: Since Fortran is case insensitive, compilers variously use \texttt{dgemm}, \texttt{dgemm\_}, and \texttt{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 does not specify how numbers are returned (e.g., on the stack or as an extra hidden argument), so compilers return them in various ways. The f2c and old g77 versions also returned singles as doubles, and this issue remains when using macOS Accelerate, which is based on the f2c version of LAPACK/BLAS.

- **Lacks mixed precision**: Mixed precision (e.g., \( y = Ax \), where \( A \) is single, and \( x \) is double) is important for mixed-precision iterative refinement routines.

- **Lacks mixed real/complex routines**: Mixed real/complex routines (e.g., \( y = Ax \), where \( A \) is complex, and \( x \) is real) occur in some eigenvalue routines.

- **Rigid naming scheme**: Since the precision is encoded in the name, the Fortran interfaces cannot 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, called XBLAS, with suffixes added to the routine name to indicate the extended datatypes. Using \texttt{gemm} as an example, the initial precision (e.g., \texttt{z} in \texttt{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.

The preceding sentence really doesn’t make sense to me as written. What does the suffix \(_a\) represent? You just say they are “letters,” but there’s more to it than that, yeah? Can you give me an example and then explain what the matrix is based on the suffix? Something like: “…\texttt{zgemm}\_a\texttt{ is a complex-double, general matrix-matrix multiply of the matrix }A” or whatever.

For extended precision, a suffix \(_X\) was added that specified it internally used extended precision.

Again, can we have an illustrative example? Just the syntax of what one of these looks like with a suffix of \(_X\).

While these parameters added capabilities to the BLAS, several issues remain:

- **Extended precision was internalized**: Output arguments were in standard precision. For parallel algorithms, the output matrix needed to be in higher precision for reductions. For instance, a parallel \texttt{gemv} would do \texttt{gemv} on each node with the local matrix and then do a parallel reduction to find the final product. To effectively use higher precision, the result of the local \texttt{gemv} had to be in higher precision, with rounding to lower precision only after the parallel reduction. XBLAS did not provide extended precision outputs.

- **Superfluous routines**: Many of the XBLAS routines were superfluous and not useful in writing LAPACK and ScALAPACK routines, thereby making implementation of XBLAS unnecessarily difficult.

- **Limited precision types**: XBLAS had no mechanism for supporting additional precision and did not support half-precision (16-bit); integer or quantized; fixed-point; extended precision (e.g., double-double—two 64-bit quantities representing one value), or quad precision (128-bit).

- **Not widely adopted**: The XBLAS was not widely adopted or implemented, although LAPACK can be built using XBLAS in some routines. The Intel® Math Kernel Library (Intel® MKL) also provides XBLAS implementations.

2.1.3 Fortran 90

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

\(^2\)http://www.netlib.org/blas/blast-forum/blast-forum.html
2.2 Programming Language C

2.2.1 Netlib CBLAS

The BLAS technical forum also introduced CBLAS, a C wrapper for the original Fortran BLAS routines. CBLAS addresses a couple of inconveniences that a user would face when using the Fortran interface directly from C. CBLAS allows for passing of scalar arguments by value, rather than by reference, replaces character parameters with enumerated types, and deals with the compiler’s mangling of the Fortran routine names. CBLAS also supports the row-major matrix layout in addition to the standard column-major layout. Notably, this is handled without actually transposing the matrices, but is accomplished instead by changing the transposition, upper/lower, and dimension arguments. Netlib CBLAS declarations reside in the `cblas.h` header file. This file contains declarations of a handful of types:

```c
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 contains signatures of all the functions:

```c
void cblas_dtrsm(CBLAS_LAYOUT layout, CBLAS_SIDE Side,
                 CBLAS_UPLO Uplo, CBLAS_TRANSPOSE TransA,
                 CBLAS_DIAG Diag, const int M, const int N,
                 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,
                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`, as shown below.

```c
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`, like so:

```c
#define CBLAS_INDEX long

void cblas_isamax(const int N, const float *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 for 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 Intel MKL CBLAS

Intel 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 precision. Also, header files in Intel 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. Like 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 of the types defined in CBLAS and, notably, preserved their integer values, as shown below.

```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
};
```
2.2. PROGRAMMING LANGUAGE C  CHAPTER 2. STANDARDS AND TRENDS

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,
}
```

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

Notably, lapack_cwrapper preserved all of the original application programming interface’s (API’s) semantics, did not introduce support for row-major layout, did not introduce any extra checks (e.g., NaN checks), and did not introduce automatic workspace allocation.

In terms of style, all names were snake case, including those for types, constants, and 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. Like CBLAS, LAPACKE replaces passing scalars by reference with passing scalars by value. LAPACKE also deals with Fortran name mangling in the same manner as CBLAS. Unlike CBLAS and lapack_cwrapper, though, LAPACKE did not replace character parameters with enumerate types.

And, unlike other C APIs for LAPACK, LAPACKE actually uses complex types for complex parameters and introduces `lapack_complex_float` and `lapack_complex_double`, which are set by default to `float _Complex` and `double _Complex`, respectively, relying on the definition of `_Complex` in `complex.h`.

For integers, LAPACKE uses `lapack_int`, which is defined as `int` by default and defined as `long` if the LAPACK_ILP64 flag is set.

Also like CBLAS, the the matrix layout is the first parameter in the LAPACKE calls. Two constants are defined with CBLAS compliant integer values, shown below.
2.2. PROGRAMMING LANGUAGE C  

CHAPTER 2. STANDARDS AND TRENDS

```c
#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: (1) a higher-level interface with names prefixed by LAPACKE_; and (2) a lower-level interface with names prefixed by LAPACKE_ and suffixed by _work.

For example:

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

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 of the input arrays, which can be disabled if LAPACKE is compiled from source, by setting the LAPACK_DISABLE_NAN_CHECK flag; notably, this is not possible with the binary distribution.

2.2.5 Next-Generation BLAS: “BLAS G2”

There is a new, ongoing effort to develop the next generation of BLAS, called “BLAS G2.” This BLAS G2 effort, first presented at the Batched, Reproducible, and Reduced Precision BLAS workshop, introduces a new naming scheme for the lower-level BLAS routines. This new scheme, which is more flexible than the single-prefix character scheme used in the original BLAS and XBLAS, uses suffixes for data types:

- 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 \texttt{r64} for traditional \texttt{dgemm}) or have mixed precisions (e.g., \texttt{blas\_gemm\_r32r32r64}, which has two single-precision matrices \([A\) and \(B]\) and a double-precision matrix \([C]\)). The new scheme also defines extensions for having different input and output matrices (e.g., \(C_{\text{in}}\) and \(C_{\text{out}}\)) and has reproducible accumulators that give the same answer regardless of the runtime choices in parallelism or evaluation order.

These additions to the scheme provide a mechanism to name the various routines. However, not all names that fit the mechanism would be implemented, and a set of recommended routines for implementation will also be defined.

It is expected that users could use higher-level interfaces in C++ and Fortran that could overload the basic operations. To mitigate this problem, for example, C++ \texttt{blas::gemm} would call the correct lower-level routine depending on the precision of the arguments it is given.

I reworked the preceding paragraph pretty hardcore. It wasn't clear what the example was (whether it was a problem or how the new BLAS G2 would solve it). Take a look and make sure I inferred correctly. Rewrite as necessary.

2.3 C++ Programming Language

A number of C++ linear algebra libraries also exist. Most of these provide actual implementations of BLAS-like functionality in C++ rather than being simple wrappers like CBLAS and LAPACKE. Some of these libraries can also call the high-performance, vendor-optimized (traditional) BLAS. The following subsections describe some of these C++ libraries.

2.3.1 Boost and 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, level-2, and level-3 BLAS functionality for dense, banded, and sparse matrices. This functionality is implemented using expression templates with lazy evaluation. Basic expressions on whole matrices are easy to specify. Example \texttt{gemm} calls include:

```cpp
// 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;
```

Here, \texttt{noalias} prevents the creation of a temporary result. While using \texttt{noalias} in this case is a bit dubious, since \(C\) is on the RHS, the result appears to be correct. It can also access submatrices, both contiguous ranges and slices, with stride between rows and columns. However, the syntax is rather cumbersome:

```cpp
noalias( project( C, range(0, i), range(0, j) ) )
   = alpha * prod( project( A, range(0, i), range(0, k) ),
```

\(^4\)http://www.boost.org/doc/libs/1.64.0/libs/numeric/ublas/doc/index.html

---

Please define RHS.

When you say “It can also access...” what do you mean? BLAS
Because the code is templated, any combination of precisions and real and complex values will work. The uBLAS interface mostly conforms to C++ STL containers and iterators. Triangular, symmetric, and Hermitian matrices are stored in a packed configuration, thereby saving significant space but also making operations slower. For example, uBLAS implements `spmv` rather than `symv`. It can also use full matrices for triangular solves to do both `trmm` and `tpmm`.

However, uBLAS is not multi-threaded, nor does it interface fully with vendor BLAS, although there is a way to get the matrix multiply to call MKL, and there is an experimental binding to work with Automatically Tuned Linear Algebra Software (ATLAS).

There does not appear to be a way to wrap existing matrices and vectors (i.e., existing matrices and vectors 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× slower than sequential Intel MKL for $n = 500$ `dgemm` on a machine running a Linux operating system, an Intel Sandy Bridge CPU, Intel icpc and GNU `g++` compilers, `-O3` and `-DNDEBUG` flags, and cold cache.

Below is an example of a blocked Cholesky algorithm.

```cpp
#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>

using namespace boost::numeric::ublas;

template< typename T, typename Layout >
typedef void 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) ) ));
      // inplace_solve doesn't compile ... don't know why
```


2.3. C++ PROGRAMMING LANGUAGE

CHAPTER 2. STANDARDS AND TRENDS

```cpp
// 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() );
```

2.3.2 Matrix Template Library: MTL 4

Matrix Template Library 4\(^6\) (MTL 4) is a C++ library that supports dense, banded, and sparse matrices. For dense matrices, it supports row-major (default), column-major, and Morton recursive layouts. MTL 4 uses parts of Boost, and the default installation even places MTL in a subfolder of Boost. For sparse matrices, MTL 4 supports compressed row storage (CRS)/compressed sparse row (CSR), compressed column storage (CCS)/compressed sparse column (CSC), coordinate, and ELLPACK formats.

Many of the functions are global functions rather than member functions. For instance, MTL 4 uses `num_rows(A)` instead of `A.num_rows()`.

MTL 4 has extensive documentation with numerous example codes. Still, the documentation is somewhat difficult to follow, and it can be difficult to find procedures on how to do certain things or find out what features are explicitly supported.

MTL 4 has native C++ implementations for BLAS operations like matrix-multiply, so it is not limited to the four precisions of traditional BLAS. By defining `MTL_HAS_BLAS`, it will interface with traditional BLAS routines for `gemm`. Upon 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.

As far as obtaining MTL 4, it has an MIT open-source license, as well as a commercial Supercomputing Edition with parallel and distributed support.

Compared to uBLAS, MTL 4’s syntax for accessing sub-matrices is nicer. An example is shown below.

```cpp
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.

Here are some example calls:

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

\(^6\)http://new.simunova.com/index.html#en-mtl4-index-html
MTL 4 uses “move semantics” to more efficiently return matrices from functions (i.e., it does a shallow copy when returning matrices). Aliasing arguments can be an issue, however, and if MTL 4 detects some aliasing methods, it will throw an exception (e.g., in \( A = A \times B \)). However, if there is partial overlap, aliasing will not be detected, and this must be resolved by the user by adding a temporary. It should be noted that traditional BLAS will not detect aliasing either. MTL 4 also throws exceptions if matrix sizes are incompatible. The user can disable exceptions by defining `NDEBUG`.

MTL 4 has triangular-vector solves (trsv) available in `upper_trisolve` and `lower_trisolve`, but it does not appear to support a triangular-matrix solve (trsm). This is an impediment to even a simple blocked Cholesky implementation. However, MTL 4 provides a recursive Cholesky implementation example. It supports recursive algorithms by providing an `mtl::recursator` that divides a matrix into quadrants (\( A_{11}, A_{12}, A_{21}, \) and \( A_{22} \)), named `north_west`, `north_east`, `south_west`, and `south_east`, respectively.

MTL 4 also supports symmetric eigenvalue problems, but it is otherwise unclear if it supports operations on symmetric matrices like `symm`, `syrk`, `syr2k`, etc. Outside of the symmetric eigenvalue problem, there is little mention of symmetric matrices, but there is an `mtl::symmetric` tag. MTL 4 interfaces with UMFPACK for sparse non-symmetric systems.

MTL also includes the following matrix solver capabilities:

- LU, with and without pivoting;
- QR orthogonalization;
- eigenvalue problems (QR iteration);
- SVD; and
- ILU(0), IC(0), IMF(s) incomplete LU, Cholesky, and multi-frontal sparse solvers.

MTL 4 optionally supports some modern C++ 11 features, including:

- 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 }};` and
- for loops using range: `for (int i : irange(size(v))) { ... }`.

Similar to uBLAS, benchmarks showed MTL 4 is around \( 14 \times \) slower than sequential Intel MKL for \( n = 500 \) `dgemm` on a machine running a Linux operating system, an Intel Sandy Bridge CPU, Intel icpc and GNU g++ compilers, -O3 and `-DNDEBUG` flags, and cold cache.

Below is an example of a blocked-Cholesky algorithm lacking trsm.
2.3 Eigen

Like uBLAS and MTL4, Eigen\(^7\), another C++ template library for linear algebra, is based on C++ expression templates. Eigen seems to be a more mature product than uBLAS and MTL 4.

In addition to BLAS-type expressions, the Eigen library includes: (1) linear solvers (e.g., LU with partial pivoting or full pivoting, Cholesky, Cholesky with pivoting [for semidefinite], QR, QR with column pivoting (rank revealing), and QR with full pivoting); (2) eignensolvers (e.g., Hermitian ["Self Adjoint"], generalized Hermitian \(Ax = \lambda Bx\) where \(B\) is HPD, and non-symmetric); and (3) SVD solvers (e.g., two-sided Jacobi and bidiagonalization).

Eigen does not include a symmetric-indefinite solver (e.g., Bunch-Kaufman pivoting, Rook pivoting, or Aasen's algorithm).

Eigen's block syntax is more succinct than other libraries:

uBLAS:  \[\text{project}( A, \text{range}( i, i+mb ), \text{range}( j, j+nb ))\]

MTL4: \[\text{sub_matrix}( A, i, i+mb, j, j+nb )\]

Eigen: \[A.\text{block}( i, j, mb, nb )\]

\(^7\)http://eigen.tuxfamily.org/
However, when using member functions in a template context, the syntax requires extra “template” keywords, which are annoying and clutter the code:

Eigen:  

```cpp
A.template block( i, j, mb, nb )
```

Eigen also provides both triangular and Hermitian (self-adjoint) views on matrices, though 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 4, aliasing can be an issue in Eigen. Component-wise operations, where the $C(i,j)$ output entry depends only on 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 in uBLAS, Eigen assumes matrix-multiply uses an alias and generates a temporary unless the user adds the `.noalias()` call.

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

Eigen has a single class that covers both matrices and vectors. This single class also covers 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, like $x \cdot y$ (in Matlab notation), and an easy conversion between matrix and array classes, as shown below.

```
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, with an assert in debug mode. Other errors like aliasing are (sometimes) detected, and execution is then aborted with an assert. If desired, these errors can be redefined to throw C++ exceptions.

Unlike uBLAS and the open-source MTL 4 release, Eigen supports multi-threading through the Open Multi-Processing (OpenMP) API. Eigen’s performance is also better than uBLAS, though it is still outperformed by the vendor-optimized code in Intel’s MKL. For single-threaded `dgemm`, Intel’s MKL is about 2× faster than Eigen for $n = 500$ (compared to MKL being 14×–15× faster than uBLAS and MTL 4), while for multi-threaded runs, MKL is 2×–4× faster than Eigen on a machine running a Linux operating system, an Intel Sandy Bridge CPU, GNU g++ compiler, -O3 and -DNDEBUG flags, and cold cache. Performance is notably worse with the Intel icpc compiler.

However, Eigen can directly call BLAS and LAPACK functions by setting `EIGEN_USE_MKL_ALL`, `EIGEN_USE_BLAS`, or `EIGEN_USE_LAPACKE`. With these options, Eigen’s performance ranges from nearly the same as Intel’s MKL to 2× slower.

Below is an example of a Cholesky factorization using the Eigen template.
```cpp
// Compute the Cholesky factorization \( A = L \times L^\top \).
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
    if (info != 0) {
        info += j;
        break;
    }
    if (j + nb < n) {
        // Compute the current block column.
        A.template block( j, j, nb, n - j ) -=
            A.template selfadjointView< Eigen::Lower >().
            rankUpdate( A.template block( j, 0, nb, j ), -1.0 );
        lapack_potrf( 'l', nb, &A(j, j), lda, &info );
        if (info != 0) {
            info += j;
            break;
        }
        if (j + j + nb < n) {
            // Compute the current block column.
            A.template block( j + j + nb, j, n - (j + j + nb), nb ) -=
                A.template selfadjointView< Eigen::Lower >().
                rankUpdate( A.template block( j + j + nb, 0, nb, j ), -1.0 );
            lapack_potrf( 'l', nb, &A(j + j + nb, j), lda, &info );
            if (info != 0) {
                info += j;
                break;
            }
            if (j + j + j + nb < n) {
                // Compute the current block column.
                A.template block( j + j + j + nb, j, n - (j + j + j + nb), (j + j) ) -=
                    A.template selfadjointView< Eigen::Lower >().
                    rankUpdate( A.template block( j + j + j + nb, 0, (j + j), (j + j) ), -1.0 );
                lapack_potrf( 'l', (j + j), &A(j + j + j + nb, j), lda, &info );
                if (info != 0) {
                    info += j;
                    break;
                }
            }
        }
    }
}
return info;
```

### 2.3.4 Elemental

Elemental\(^8\) is an MPI-based, distributed-memory linear algebra library that includes a C++ interface for BLAS (excluding band and packed formats) and a selection of LAPACK routines. Elemental’s BLAS interface is in the `El::blas` namespace, and functions are named after the traditional BLAS routines, with the exception of the precision prefix, in Pascal case. For the four standard precisions (single, double, complex-single, and complex-double), Elemental calls an optimized BLAS library. It also offers a templated C++ reference implementation for arbitrary numeric datatypes like `int` or `double-double`.

Hermitian and symmetric routines are extended to all precisions. For example, Herk (\( C = \alpha A A^\top + \beta C, C \) is Hermitian) and Syrk (\( C = \alpha A A^\top + \beta C, C \) is symmetric) are both available for real and complex data types. Dot products are also defined for both real and complex. This allows for templated code to use the same name for all data types.

Arguments in the Elemental wrappers are similar to the traditional BLAS and LAPACK arguments, including options, dimensions, leading dimensions, and scalars. Dimensions use `int`, and there is experimental support for 64-bit integers. Options are a single character, corresponding to the traditional BLAS options; this differs from CBLAS, which uses enums for options. For instance, a NoTrans, Trans matrix-matrix multiply (\( C = \alpha A B^\top + \beta C \)) is expressed as:

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

\(^8\)http://libelemental.org/
Elemental wraps a handful of LAPACK routines, with most of these dealing with eigenvalue and singular value problems. Instead of functions using the LAPACK acronym names (e.g., syevr), Elemental uses descriptive English names (e.g., HermitianEig).

In LAPACK, eigenvalue routines have a job parameter that specifies whether to compute just eigenvalues or to also compute 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, thereby avoiding the need to specify the job parameter and unused dummy arguments. See below.

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

Elemental also provides wrappers around certain functionalities provided in MPI, the Scalable Linear Algebra PACKage (ScalAPACK), Basic Linear Algebra Communication Subprograms (BLACS), Parallel Basic Linear Algebra Subprograms (PBLAS), libFLAME, and the Parallel Multiple Relatively Robust Representations (PMRRR) library.

Elemental throws the (SingularMatrixException and NonHPDMatrixException) C++ exceptions 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 data type only. It uses a column-major LAPACK matrix layout, with a leading dimension that may be explicitly specified as an option—unlike most other C++ libraries reviewed here.

A Matrix can also be constructed as a view to an existing memory buffer, as shown below.

```cpp
1 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 they are in CBLAS. In particular, Elemental has an Orientation enum instead of Transpose, with values El::NORMAL, El::TRANSPOSE, and El::ADJOINT corresponding to NoTrans, Trans, and ConjTrans, respectively. In addition to standard BLAS routines, Elemental provides the following routines, among others.
Adjoint  \quad \text{out-of-place conjugate transpose, } B = A^H \\
Axpy \quad \text{add matrices, } Y = \alpha X + Y \\
Broadcast \quad \text{parallel broadcast} \\
DiagonalScale \quad X = \text{op}(D) X \\
Dot \quad \text{matrix Hilbert-Schmidt inner product, } \text{vec}(A)^H \text{vec}(B) \\
Hadamard \quad \text{element-wise product, } C = A \circ B \\
QuasiTrsm \quad \text{Schur-form quasi-triangular solve} \\
Reduce \quad \text{parallel reduction} \\
Transpose \quad \text{out-of-place transpose, } B = A^T \\
Trrk \quad \text{Rank-} k \text{ update limited to triangular portion (e.g., useful for syrk-like update } \\
\quad C = \alpha AB + \beta C \text{ when } AB \text{ is known to be symmetric; cf. syrkh in cuBLAS} \\
\quad \text{and gemmt in Intel MKL) } \\
TwoSidedTrmm \quad A = L^H AL \\
TwoSidedTrsm \quad A = L^{-1} AL^{-H} \\

In addition to standard LAPACK algorithms, Elemental provides pivoted Cholesky, non-pivoting 
LU, and complete-pivoting LU. It also has a number of other matrix factorizations and 
applications like 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), shown below.

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

Because C++ cannot take a non-\texttt{const} reference of a temporary, the output submatrix of each 
call must be a local variable. For example, one cannot write:

```cpp
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) ) );
```

Instead, one must make the local variable \texttt{Ajj}:

```cpp
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 );
```

Elemental could resolve this issue by adding an overloaded version of Herk and other routines 
using a C++ 11 \texttt{rvalue} reference (\&\&) for the output matrix. Thanks to Vincent Picaud for 
pointing this out.

Below is an example of a Cholesky factorization in Elemental.

```cpp
#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;
```
2.3.5 Intel DAAL

The Intel® Data Analytics Acceleration Library (Intel® DAAL)\(^9\) provides highly optimized algorithmic building blocks for data analysis, including: preprocessing, transformation, analysis, modeling, validation, etc. Intel DAAL contains routines for, e.g.: principal component analysis, linear regression, classification, clustering, etc. Intel 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. It is also designed for distributed processing using popular data analytics platforms: Hadoop, Spark, R, and Matlab. Intel DAAL can access data from memory, files, and SQL databases.

Intel 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:

```cpp
#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:

```cpp
#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>::xsysrk(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:

```cpp
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_d,syrk, (uplo, trans, p, n, alpha, a, lda, beta, ata, ldata));
    }
};
```

and for single precision:

```cpp
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_s,syrk, (uplo, trans, p, n, alpha, a, lda, beta, ata, ldata));
    }
};
```

The call passes through a couple of macro definitions

```cpp
#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(axx512 == cpu)  
  {  
    __DAAL_MKLFN(axx512_,f_pref,f_name) f_args;  
  }

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

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

Calls to LAPACK are handled in a similar manner. Intel 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.h"

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, Intel DAAL calls BLAS and LAPACK through static member functions of the Blas and Lapack class templates. Also, Intel DAAL uses the legacy BLAS calling convention (Fortran), where parameters are passed by reference, and there is no parameter to specify the layout (column-major or row-major). Finally, Intel 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\(^{10}\) 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.

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);
}
```

\(^{10}\)https://trilinos.org/
An example code that invokes dense solver routines for a system of linear equations given by a square matrix is as follows:

```cpp
ANTEDESSE (Teuchos::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
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
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:

```
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:

```
#include "Teuchos_SerialDenseMatrix.hpp"
#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();
}
2.3. C++ PROGRMMING LANGUAGE  CHAPTER 2. STANDARDS AND TRENDS

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.GETS('N', n, nrhs, A, ldA, piv, B, ldB, &info);
}
```
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:

```cpp
#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:

```cpp
#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:

<table>
<thead>
<tr>
<th>C++ name</th>
<th>real</th>
<th>complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>blas::hemv</td>
<td>[sd]symv</td>
<td>[cz]hemv</td>
</tr>
<tr>
<td>blas::symv</td>
<td>[sd]symv</td>
<td>[cz]symv</td>
</tr>
<tr>
<td>blas::her</td>
<td>[sd]syr</td>
<td>[cz]her</td>
</tr>
<tr>
<td>blas::syr</td>
<td>[sd]syr</td>
<td>[cz]syr</td>
</tr>
<tr>
<td>blas::her2</td>
<td>[sd]syr2</td>
<td>[cz]her2</td>
</tr>
<tr>
<td>blas::syr2</td>
<td>[sd]syr2</td>
<td>[cz]syr2</td>
</tr>
<tr>
<td>blas::herk</td>
<td>[sd]syrk</td>
<td>[cz]herk</td>
</tr>
<tr>
<td>blas::syrk</td>
<td>[sd]syrk</td>
<td>[cz]syrk</td>
</tr>
<tr>
<td>blas::her2k</td>
<td>[sd]syr2k</td>
<td>[cz]her2k</td>
</tr>
<tr>
<td>blas::syr2k</td>
<td>[sd]syr2k</td>
<td>[cz]syr2k</td>
</tr>
<tr>
<td>blas::hemm</td>
<td>[sd]symm</td>
<td>[cz]hemm</td>
</tr>
<tr>
<td>blas::symb</td>
<td>[sd]symm</td>
<td>[cz]symm</td>
</tr>
<tr>
<td>blas::dot</td>
<td>[sd]dot</td>
<td>[cz]dotc</td>
</tr>
<tr>
<td>blas::dotc</td>
<td>[sd]dot</td>
<td>[cz]dotc</td>
</tr>
<tr>
<td>blas::dotu</td>
<td>[sd]dot</td>
<td>[cz]dotu</td>
</tr>
</tbody>
</table>

†[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):

<table>
<thead>
<tr>
<th>C++ name</th>
<th>real</th>
<th>complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>lapack::hesv</td>
<td>[sd]sysv</td>
<td>[cz]hesv</td>
</tr>
<tr>
<td>lapack::sysv</td>
<td>[sd]sysv</td>
<td>[cz]sysv</td>
</tr>
<tr>
<td>lapack::unmqr</td>
<td>[sd]ormqr</td>
<td>[cz]unmqr</td>
</tr>
<tr>
<td>lapack::ormqr</td>
<td>[sd]ormqr</td>
<td>—</td>
</tr>
<tr>
<td>lapack::ungqr</td>
<td>[sd]orgqr</td>
<td>[cz]ungqr</td>
</tr>
<tr>
<td>lapack::orgqr</td>
<td>[sd]orgqr</td>
<td>—</td>
</tr>
</tbody>
</table>

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
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):

```plaintext
1 enum class Layout : char { ColMajor = 'C', RowMajor = 'R' };
2 enum class Op : char { NoTrans = 'N', Trans = 'T', ConjTrans = 'C' };
3 enum class Uplo : char { Upper = 'U', Lower = 'L' };
4 enum class Diag : char { NonUnit = 'N', Unit = 'U' };
5 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:

```plaintext
1 TRANSA = 'N' or 'n', op( A ) = A.
2 TRANSA = 'T' or 't', op( A ) = A\textsuperscript{T}.
3 TRANSA = 'C' or 'c', op( A ) = A\textsuperscript{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:

<table>
<thead>
<tr>
<th>C++ API</th>
<th>LAPACK</th>
<th>matrix type</th>
</tr>
</thead>
<tbody>
<tr>
<td>gescl</td>
<td>lascl with type=G</td>
<td>general</td>
</tr>
<tr>
<td>trscl( uplo )</td>
<td>lascl with type=uplo</td>
<td>triangular or Hermitian</td>
</tr>
<tr>
<td>gbscl</td>
<td>lascl with type=Z</td>
<td>general band</td>
</tr>
<tr>
<td>hbscl( uplo )</td>
<td>lascl with type=B (Lower) or Q (Upper)</td>
<td>Hermitian band</td>
</tr>
<tr>
<td>hsscl</td>
<td>lascl with type=H</td>
<td>Hessenberg</td>
</tr>
<tr>
<td>gecpy</td>
<td>lacpy with uplo=G</td>
<td>general</td>
</tr>
<tr>
<td>trcpy( uplo )</td>
<td>lacpy with same uplo</td>
<td>triangular or Hermitian</td>
</tr>
<tr>
<td>geset</td>
<td>laset with uplo=G</td>
<td>general</td>
</tr>
<tr>
<td>trset( uplo )</td>
<td>laset with same uplo</td>
<td>triangular or Hermitian</td>
</tr>
</tbody>
</table>

**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 cases². 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 libxsmm³ 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.

¹See explanation in Batched BLAS document why xerbla is a hideous monstrosity for parallel codes or multiple libraries.
³https://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 stdint 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. 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 $O(n)$ 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 $m \leftrightarrow n$, 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 $m \leftrightarrow n$ 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 as 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.

```plaintext
namespace blas {

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

```
3.15. PROTOTYPE IMPLEMENTATION

CHAPTER 3. C++ API DESIGN

10 // ----------------------------------------------------------------------------
11 // Fortran name mangling depends on compiler, generally one of:
12 // UPPER
13 // lower
14 // lower ## _
15 #ifndef BLAS_FORTRAN_NAME
16 #define BLAS_FORTRAN_NAME( lower, UPPER ) lower ## _
17 #endif
18
19 // blas_int is the integer type of the underlying Fortran BLAS library.
20 // BLAS wrappers take int64_t and check for overflow before casting to blas_int.
21 #ifndef BLAS_ILP64
22 typedef long long blas_int;
23 #else
24 typedef int blas_int;
25 #endif
26
27 // ----------------------------------------------------------------------------
28 // enum class Layout : char { ColMajor='C', RowMajor='R' };
29 enum class Op : char { NoTrans='N', Trans='T', ConjTrans='C' };
30 enum class Uplo : char { Upper='U', Lower='L' };
31 enum class Diag : char { NonUnit='N', Unit='U' };
32 enum class Side : char { Left='L', Right='R' };
33
34 // ----------------------------------------------------------------------------
35 class Error : public std::exception
36 {
37 public:
38   Error(): std::exception() {}
39   Error( const char* msg ): std::exception(), msg_( msg ) {}
40   virtual const char* what() { return msg_.c_str(); }
41 private:
42   std::string msg_;  
43};
44
45 // internal helper function; throws Error if cond is true
46 // called by blas_throw_if macro
47 inline void throw_if( bool cond, const char* condstr )  
48 {
49   if ( cond ) {  
50     throw Error( condstr );
51   }
52 }
53
54 // internal macro to get string #cond; throws Error if cond is true
55 #define blas_throw_if( cond ) \  
56   throw_if( cond, #cond )
57
58 // Fortran prototypes
59 // sgemm, dgemm, cgemm omitted for brevity
60 #define BLAS_zgemm BLAS_FORTRAN_NAME( zgemm, ZGEMM )
61 extern "C"
62 void BLAS_zgemm( char const* transA, char const* transB, 
63   blas_int const* m, blas_int const* n, blas_int const* k, 
64   std::complex<double> const* alpha, 
65   std::complex<double> const* A, blas_int const* lda, 
66   std::complex<double> const* B, blas_int const* ldb, 
67   std::complex<double> const* beta, 
68   std::complex<double>* C, blas_int const* ldc );
69
// 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>* C, blas_int ldc )
{
    BLAS_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
    blas_throw_if ( layout != Layout::RowMajor && layout != Layout::ColMajor );
    blas_throw_if ( transA != Op::NoTrans && transA != Op::Trans && transA != Op::ConjTrans );
    blas_throw_if ( transB != Op::NoTrans && transB != Op::Trans && transB != Op::ConjTrans );
    blas_throw_if ( m < 0 );
    blas_throw_if ( n < 0 );
    blas_throw_if ( k < 0 );
    blas_throw_if ( lda < Am );
    blas_throw_if ( ldb < Bm );
    blas_throw_if ( ldc < Cm );

    // check for overflow in native BLAS integer type, if smaller than int64_t
    if ( sizeof(int64_t) > sizeof(blas_int) ) {
        blas_throw_if ( m > std::numeric_limits<blas_int>::max() );
        blas_throw_if ( n > std::numeric_limits<blas_int>::max() );
        blas_throw_if ( k > std::numeric_limits<blas_int>::max() );
        blas_throw_if ( lda > std::numeric_limits<blas_int>::max() );
        blas_throw_if ( ldb > std::numeric_limits<blas_int>::max() );
        blas_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,
            (blas_int) lda, (blas_int) ldb,
            A, (blas_int) lda, B, (blas_int) ldb,
            alpha,
            C, (blas_int) ldc );
    } else {
        gemm_( (char) transA, (char) transB,
            (blas_int) m, (blas_int) n, (blas_int) k,
            (blas_int) lda, (blas_int) ldb,
            A, (blas_int) lda, B, (blas_int) ldb,
            alpha,
            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

} // 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, blas_throw_if similar to blas.hh

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

extern "C"
void LAPACK_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, cpostrf omitted for brevity
inline void potrf_( char uplo, int_type n,
                    std::complex<double>* A, int_type lda,
                    int_type* info )
{
  LAPACK_zpotrf( &uplo, &n, A, &lda, info );
}

} // namespace lapack

#endif // ifndef LAPACK_HH
// templated wrapper checks arguments, handles row-major to col-major translation

```cpp
template< typename T >
int64_t potrf( Layout layout, Uplo uplo, int64_t n, T* A, int64_t lda )
{
    // check arguments
    blas_throw_if( layout != Layout::RowMajor && layout != Layout::ColMajor );
    blas_throw_if( uplo != Uplo::Upper && uplo != Uplo::Lower );
    blas_throw_if( n < 0 );
    blas_throw_if( lda < n );

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

3.15. PROTOTYPE IMPLEMENTATION
CHAPTER 3. C++ API DESIGN

36