

uLAPACK: Linear Algebra Package for uBLAS

Tim Bailey

Up-to-date for Version 1.0 of the Source Code
April 16, 2005

1 Introduction

The uBLAS library is a C++ template class library that provides BLAS level 1, 2, 3 functionality. It implements the dual goals of notational convenience via operator overloading and efficient code generation via expression templates. This uLAPACK library is designed to augment uBLAS with LAPACK functionality. It currently implements:

- LU decomposition.
- Inverse and determinant for general square matrices.
- Cholesky decomposition.
- Inverse and determinant for positive-definite matrices.

The list of “things to do” include:

- Reciprocal condition numbers.
- Linear system solvers.
- SVD.

2 System Requirements

To use the uLAPACK library, you need to have Boost uBLAS and Netlib LAPACK or CLAPACK installed. The Boost library can be downloaded from

<http://www.boost.org/>

Source versions of the Fortran LAPACK library, and its C translation CLAPACK, can be downloaded from

<http://www.netlib.org/lapack/>

3 Quick Guide

You want to implement linear algebra operations in C++ with the notational convenience of MATLAB. With uLAPACK, this can be reasonably achieved. For example, a numerically stable form of the Kalman filter update step can be implemented as:¹

```
// Cholesky factorised form of the Kalman update step
void kalman_update(Vector &x, Matrix &P,
    const Vector &innov, const Matrix &R, const Matrix &H)
{
    Matrix PHt = prod(P, trans(H));
    Matrix S = prod(H, PHt) + R; // innovation covariance
    Matrix Sci = inv(chol(S));
    Matrix Wc = prod(PHt, Sci);
    Matrix W = prod(Wc, trans(Sci)); // Kalman gain

    x += prod(W, innov);
    P -= prod(Wc, trans(Wc));
}
```

The uLAPACK library provides a number of different versions of each operation type. For example, to invert a positive-definite matrix, one has the following options:

- Return a value. e.g.,

```
Matrix xi = inv_pd(x);
```

- Compute in-place. e.g.,

```
Matrix xi(x);
inv_pd_inplace(xi);
```

- Instantiate an object. e.g.,

```
Cholesky<Matrix> c(x);
Matrix xi = c.inv();
```

The return-value versions are usually the most convenient, particularly if operations are nested as in the Kalman filter example above. The in-place versions are typically most efficient, as they avoid creating unnecessary temporary objects. The class-based forms may be more efficient if several different operations are performed on a matrix (e.g., inverse and determinant). The class caches the decomposed matrix information to save recomputation. The class-based forms are also a little safer than some of the stand-alone functions in terms of preventing misuse.

Simple test code demonstrating the Cholesky-based and LU-based functions, respectively, can be found in files

```
test_cholesky.cpp
test_lu.cpp
```

¹In this example, the input parameters are the state mean, the state covariance, the innovation, the observation noise, and the observation model, respectively.

4 Reference

Each of the stand-alone functions in Sections 4.1 to 4.3 are templates of the form

```
template<class F, class A>
```

where `class F` is the matrix format,

```
ublas::row_major  
ublas::column_major
```

and `class A` is the matrix storage type,

```
ublas::bounded_array  
ublas::unbounded_array  
std::vector  
etc
```

To simplify reading of these interfaces, we neglect the template part of the function declaration and also replace the rather convoluted matrix type

```
ublas::matrix<double,F,A>
```

with the abbreviation `Matrix`.

4.1 Cholesky Decomposition Operations <cholesky.hpp>

The Cholesky-based operations are for manipulation of symmetric positive-definite matrices. The three basic operations currently implemented are: Cholesky decomposition, inversion and determinant. The key properties of these functions are:

- The result of matrix inversion is guaranteed to be symmetric.
- All functions throw an exception if the input matrix is not square (apart from `det_chol()`, which does no checking at all).
- All functions throw an exception if the input matrix is not positive-definite, apart from `det_chol()` and `chol_checked_inplace()`. The latter returns `false` for non-positive-definite matrices, which some algorithms may use as a runtime diagnostic.
- All stand-alone functions check for invalid input except `det_chol()` and `inv_chol_inplace()`. These two unsafe functions are provided for efficiency reasons.
- The class version provides no unsafe operations.

4.1.1 Return-Value Forms

Cholesky decomposition

```
Matrix chol(const Matrix &m, const bool upper=true);
```

Returns the Cholesky factored matrix for `m`, where `m` is positive-definite. The return value is upper or lower triangular depending on the second argument, which defaults to upper.

Inverse of a positive-definite matrix

```
Matrix inv_pd(const Matrix &m);
```

Returns the inverse matrix of `m`, where `m` is positive-definite.

Determinant of a Cholesky matrix

```
double det_chol(const Matrix &m);
```

Returns the determinant of `m`, where `m` is assumed to be in Cholesky factored form. Warning: this function does not check that `m` is in the correct form.

Square-root of the determinant of a positive-definite matrix

```
double det_pd_sqrt(const Matrix &m);
```

Returns the square-root of the determinant of `m`, where `m` is positive-definite.

Determinant of a positive-definite matrix

```
double det_pd(const Matrix &m);
```

Returns the determinant of `m`, where `m` is positive-definite.

4.1.2 In-Place Forms

Cholesky decomposition with checking

```
bool chol_checked_inplace(Matrix &m, const bool upper=true);
```

Converts positive-definite matrix `m` to a Cholesky factored matrix. The result is upper or lower triangular depending on the second argument, which defaults to upper. Returns `true` if successful, `false` if input matrix is not positive-definite.

Cholesky decomposition without checking

```
void chol_inplace(Matrix &m, const bool upper=true);
```

Converts positive-definite matrix `m` to a Cholesky factored matrix. The result is upper or lower triangular depending on the second argument, which defaults to upper.

Convert Cholesky matrix to inverse of positive-definite matrix

```
void inv_chol_inplace(Matrix &m, const bool upper=true);
```

Converts matrix `m` in-place, where `m` is a Cholesky matrix. Warning: the result of this operation is *not* the inverse of the Cholesky matrix. It is the inverse of the positive-definite matrix that `m` was factored from. Also, the second input argument *must* match the actual upper/lower triangular state of `m`. This function does not check.

This function is provided solely for when efficiency is a premium. If the Cholesky factored matrix has already been found, it saves having to compute the decomposition again (as in `inv_pd_inplace()`).

Invert positive-definite matrix

```
void inv_pd_inplace(Matrix &m);
```

Invert matrix `m`, where `m` is a positive-definite matrix.

4.1.3 Cholesky Class

The class form of the Cholesky-based operations accepts a positive-definite matrix in its constructor. The constructor precomputes the Cholesky factored matrix and caches it. Thus, calls to any of the member-functions do not incur the overhead of recomputing the decomposition.

```
template<class M>
class Cholesky {
public:
    Cholesky(const M &m, const bool upper=true);

    bool is_posdef() const;
    const M& chol() const;
    M inv() const;
    double det_sqrt() const;
    double det() const;
};
```

4.2 LU Decomposition Operations <lufactor.hpp>

The LU-operations are for manipulation of general matrices. LU decomposition may be performed on arbitrary matrices. LU-based inversion and determinant may only be performed on square matrices. The key properties of these functions are:

- All functions throw an exception if passed invalid input.
- The inversion and determinant functions throw an exception if the input matrix is not square.
- Inversion operations throw an exception if the input matrix is singular.

4.2.1 Return-Value Forms

Inverse of a general square matrix

```
Matrix inv(const Matrix &m);
```

Returns the inverse of matrix `m`.

Determinant of a general square matrix

```
double det(const Matrix &m);
```

Returns the determinant of matrix `m`.

4.2.2 In-Place Forms

LU decomposition of a general matrix

```
bool lufactor(Matrix &m, ublas::vector<int> &pivot);
```

Performs LU decomposition of matrix `m`. The factored matrix is stored in `m` and its associated pivot-index vector is stored in `pivot`. The function returns `false` if the input matrix was singular. Warning: this function is considerably more efficient if `m` is column-major. Row-major matrices incur the overhead of two additional matrix copy operations (to and from column-major format). Note that this overhead does not affect inversion or determinant operations, which are equally efficient for either matrix format.

Invert a general square matrix

```
void inv_inplace(Matrix &m);
```

Invert matrix `m`.

4.2.3 LU Class

The class form of LU-based operations accepts an arbitrary matrix in its constructor. However, calling the inversion or determinant member-functions will throw an exception if the matrix is not square. The class constructor precomputes the LU decomposition and caches the resulting matrix and its pivot vector. Thus, calls to any member-function do not incur the decomposition overhead.

```
template<class M>
class LU {
public:
    LU(const M &m);

    bool is_singular() const;
    M inv() const;
    double det() const;
};
```

4.3 Utilities

The uLAPACK library incorporates a number of utility functions.

4.3.1 Symmetry Functions <symmetry.hpp>

Check for symmetry

```
bool is_symmetric(const Matrix &m);
```

Returns `true` if matrix `m` is symmetric.

Make a matrix symmetric

```
void force_symmetry(Matrix &m, const bool upperToLower=true);
```

Copies the upper-triangular part of matrix `m` to its low-triangular part, or vice-versa as specified by the second input argument.

4.3.2 Scalar-Fill Functions <scalar_fill.hpp>

```
void scalar_fill(Matrix &m, double x);
```

Set every element of matrix `m` to the value `x`. Several variations of this fill operation are available. Calling

```
scalar_fill(m, x);
```

fills the entire matrix. Calling

```
scalar_fill<TRI>(m, x);
```

fills a specified triangular part of the matrix, where `TRI` can be `ublas::upper`, `ublas::lower`, `ublas::strict_upper`, or `ublas::strict_lower`.

4.3.3 Matrix Printing <printmatrix.hpp>

```
void print_matrix(const Matrix &m);
```

Prints matrix `m` to the screen with matrix-like formatting.

4.3.4 Error Message Macro <errormacros.hpp>

```
#define ERROR_INFO(message)
```

Adds file-name and line-number information to an error message string. Typical use is in a `throw` statement:

```
throw MyException(ERROR_INFO("Something failed"));
```

4.4 Exceptions <lapack_exception.hpp>

The functions in this library throw two types of exception: `NumericalError` for numerical faults, such as trying to invert a singular matrix, and `LogicalError` for flawed logic, such as trying to invert a non-square matrix. Both types are derived from `LapackError`, which, in turn, derives from `std::exception`. All uLAPACK exceptions exhibit two member functions: the standard `what()` interface, which returns a character string, and

```
int get_info() const;
```

which returns a LAPACK “info” value. If `get_info()` returns 0 then the error is not generated from an internal LAPACK function call, but from auxiliary checks by uLAPACK.

5 Notes on Using uBLAS

The uBLAS library performs efficient BLAS operations by using expression templates to remove unwanted temporary objects. Documentation on effective use of uBLAS facilities is scanty and the following considers a few issues.

- Nested products can be very inefficient.

```
D = prod(A, prod(B, C));
```

Since `prod()` is transitive (not commutative) the bracket order affects the complexity. To avoid this problem, introduce a temporary matrix.

```
tmp = prod(B, C);  
D = prod(A, tmp);
```

- There exists a faster version of `prod()` called `axpy_prod()`, which implements block algorithms for more efficient locality-of-reference properties.
- The use of `noalias` can increase efficiency for expressions where the left-hand variable does not appear in the right-hand-side of an equation. For example,

```
noalias(x) += prod(W, innov);
```

avoids creating an unnecessary temporary object before adding the right-hand-side to `x`.

- Operations on `symmetric_matrix` may be significantly slower than for ordinary matrix types due to its packed implementation. For this reason, `uLAPACK` uses ordinary matrices even for symmetric operations.

Most of the above information was obtained from

http://www.crystalclearsoftware.com/cgi-bin/boost_wiki/wiki.pl?Effective_UBLAS

which discusses a few aspects of effective `uBLAS` usage.

6 Acknowledgements

This library borrowed ideas from the *uLAPACK* implementation of Alex Brooks and Ian Mahon, and the *rawLAPACK* module of Mike Stevens' *Bayes++* library.