# LAPACK++ V. 0.9

Users' Guide

Janunary 1994

February 2, 1994

**Jack Dongarra**
**Roldan Pozo**
**David Walker**

Oak Ridge National Laboratory
University of Tennessee, Knoxville

**Note: This document is a working draft. Its contents are incomplete and subject to change. Updates will be available from `netlib@onrl.gov`.**

# Contents

# 1 Introduction

LAPACK++ is an object-oriented C++ extension to the Fortran LAPACK [1] library for numerical linear algebra. This package includes state-of-the-art numerical algorithms for the more common linear algebra problems encountered in scientific and engineering applications. It is based on the widely used EISPACK [5] and LINPACK [2] libraries for solving linear equations, linear least squares, and eigenvalue problems for dense and banded systems.

Traditionally, such libraries have been available only in Fortran; however, with an increasing number of programmers using C and C++ for scientific software development, there is a need to have high-quality numerical libraries to support these platforms as well. LAPACK++ can provide the speed and efficiency competitive with native Fortran codes (see section 2.4), while allowing programmers to capitalize on the software engineering benefits of object oriented programming. Replacing the Fortran 77 interface of LAPACK with an object-oriented framework simplifies the coding style and allows for a more flexible and extendible software platform. The design goals of LAPACK++ include

- Maintain performance competitive with Fortran.
- Provide a simple interface that hides implementation details of various matrix storage schemes and their corresponding factorization structures.
- Provide a universal interface and open system design for integration into user-defined data structures and third-party matrix packages.
- Replace static work array limitations of Fortran with more flexible and type-safe dynamic memory allocation schemes.
- Provide an efficient indexing scheme for matrix elements that has minimal overhead and can be optimized for in most application code loops.
- Utilize function and operator overloading in C++ to simplify and reduce the number of interface entry points to LAPACK.
- Utilize exception error handling in C++ for intelligent managing of error situations without cluttering up application codes.
- Provide the capability to access submatrices by **reference**, rather than by value, and perform factorizations "in place". This is vital for implementing blocked algorithms efficiently.
- Provide more meaningful naming conventions for variables and function names. (Names no longer limited to six alphanumeric characters.)

LAPACK++ also provides an object-oriented interface to the Basic Linear Algebra Subprograms (BLAS) [3], [4] (see section 2.3), allowing programmers to utilize these optimized computational kernels in their own C++ applications.

The current LAPACK software consists of over 1,000 routines and 600,000 lines of Fortran 77 source code. These routines have been highly optimized for vector, superscalar, and cached-memory computer architectures. To provide a C++ extension within a reasonable time frame, we propose to use automated language translators to convert most of the source code into C. Preliminary versions of LAPACK++ will use a C version generated by `f2c` and will utilize assembly-coded BLAS kernels whenever available. The matrix and various support classes of LAPACK++ will then provide the object-oriented framework for these algorithms.

# 2 Overview

The underlying philosophy of the LAPACK++ is to provide an interface which is relatively simple, yet powerful enough to express all complex and subtle tasks within LAPACK, including those which optimize performance and/or storage. Following the framework of LAPACK, the C++ extension contains **driver routines** for solving standard types of problems, **computational routines** to perform a distinct computational task, and **auxiliary routines** to perform a certain subtask or common low-level computation. Each driver routine typically calls a sequence of computational routines. Taken as a whole, the computational routines can perform a wider range of tasks than are covered by the driver routines. Currently, dense and band matrices are supported but not general sparse matrices.

Utilizing function overloading and object inheritance in C++, the procedural interface to the LAPACK is simple: two fundamental drivers and their variants, `LA_Lin_Solve()` and `LA_Eign_Solve()`, replace several hundred subroutines in the original Fortran version.

## 2.1 Contents of LAPACK++

- Algorithms
  - LU Factorization
  - Cholesky ($LL^T$) Factorization
  - QR Factorization
  - Singular Value Decomposition (SVD)
  - Eigenvalue problems (currently what LAPACK supports)
- Storage Classes
  - rectangular matrices
  - symmetric and symmetric positive definite (SPD)
  - banded matrices
  - tri/bidiagonal matrices
- Element Data Types
  - float, double, complex, double complex
  - external data types via templates (section 3)

## 2.2 A simple code example

To provide a first glimpse at how LAPACK++ simplifies the user interface, this section presents a few simple code fragments. The examples are incomplete and are meant to merely illustrate the interface style. The next few sections will further discuss the details of matrix classes and their operations.

The first example illustrates a code fragment to solve a linear system $Ax = b$ using LU factorization:

```
#include <lapack++.h>                           // 1

LaGenMatDouble A(N,N);                          // 2
LaVectorDouble x(N), b(N);                      // 3
...

LaLinSolve(A,x,b);                              // 4
```

Line (1) includes all of the LAPACK++ object and function declarations. Line (2) declares $A$ to be a square $N \times N$ coefficient matrix, while line (3) declares the right-hand-side and solution vectors. Finally, the `LaLinSolve()` function in line (4) calls the underlying LAPACK driver routine `SGESV()` for solving linear equations.

Consider now solving a similar system with a tridiagonal coefficient matrix:

```
#include <lapack++.h>

LaTridiagMatDouble A(N,N);
LaVectorDouble x(N), b(N);
...

LaLinSolveDouble(A,x,b);
```

The only code modfication is in the declaration of $A$. In this case `LaLinSolve()` calls the driver routine `SGTSV()` for banded linear systems. The `LaLinSolve()` function has been **overloaded** to perform different tasks depending on the type of the input matrix $A$. If the matrix types are known at compile time, as in this example, then there is no runtime overhead associated with this.

## 2.3 Basic Linear Algebra Subroutines (BLAS++)

The Basic Linear Algebra Subprograms (BLAS) [3] has been the key to obtaining good performance on a wide variety of computer architectures. The BLAS define a common interface for low-level operations often found in computational kernels. These operations, such as matrix/matrix multiply and triangular solves, typically comprise of most of the computatioal workload found in dense and banded linear algebra algorithms. The Level 3 BLAS obtains good performance on a wide variety of architectures by keeping data used most often in the closest level of memory hierarhcy (registers, cache, etc.).

The BLAS++ interface simplifies many of the calling sequences to the traditional f77 BLAS interface, by using the LAPACK++ matrix classes. These routines are called within the LAPACK++ algorithms, or can be called directly at the user-level within applications.

There are two levels of the BLAS++ interface. The first is a direct interface, as shown in Table 7, are essentially inlined to call BLAS directly in eliminating any overhead. The other, more elegant interface, overloads the binary opertors * and + for simple expressions such as `C=A*B`. Having these two interfaces gives the users the choice between simplicity and performance in their application codes. See Appendix B for a list of BLAS++ interface routines.

## 2.4 Performance

Figure 1, for example, illustrates the performance (Megaflop) rating of the simple code

```
LaBlasMatMultDouble(C, A, B);
```

for square matrices of various sizes on the IBM RS/6000 Model 550 workstation. This particular implementation used GNU g++ v. 2.3.1 and utlized the BLAS-3 routines from the native ESSL library. The performance results are nearly identical with those of optimized Fortran calling the same library. This is accomplished by *inlining* the LAPACK++ BLAS kernels directly into the unerlying function call. For example, `LaMatMult(C, A, B)` is expanded inline into

```
inline LaGenMat& LaMatMult(LaGenMat &C, LaGenMat &A, LaGenMat &B)
{
    char t = 'N';
    int m = A.size(0), k = A.size(1), n = B.size(1);
    double alpha = 1.0, beta = 0.0;
```

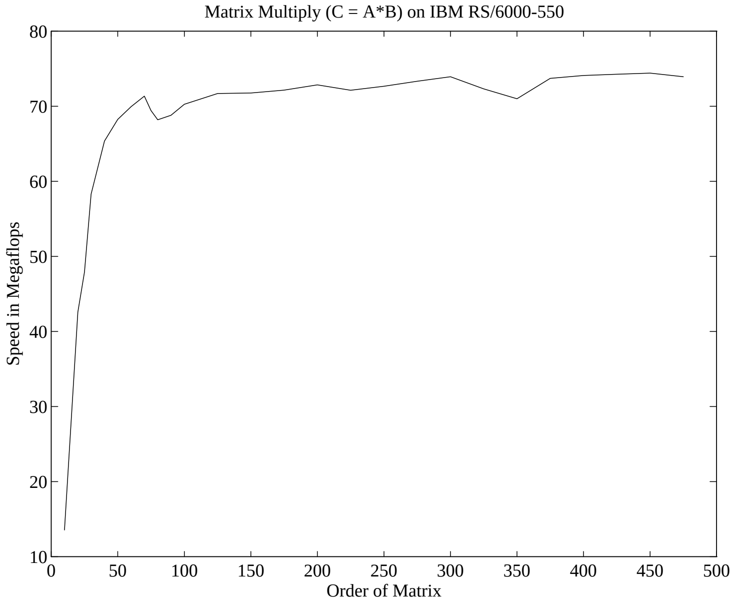


Figure 1: Performance of matrix multiply in LAPACK++ on the IBM RS/6000 Model 550 workstation. GNU g++ v. 2.3.1 was used together with the ESSL BLAS-3 routine `dgemm`.

```
    int lda = A.gdim(0), ldb = B.gdim(0), ldc = C.gdim(0);

    dgemm_(&t, &t, &m, &n, &k, &alpha, A, &lda, B, &ldb, &beta, C, &ldc);

    return C;
}
```

to call the underlying DGEMM BLAS call. This occurs at *compile* time, without any runtime overhead. The performance numbers are very near the machine peak and illustrate that using C++ with optimized computational kernels provides an elegant high-level interface without sacrificing performance.

The performance difference between optimized BLAS called from Fortran and the BLAS++ called from C++ is barely measureable. Figure 2 illustrates performance characteristics of the LU factorization of for square matrices of various sizes on the IBM RS/6000 Model 550 workstation. This particular implementation used GNU g++ v. 2.3.1 and utlized the BLAS-3 routines from the native ESSL library. The performance results are nearly identical with those of optimized Fortran calling the same library. This is accomplished by *inlining* the LAPACK++ BLAS kernels directly into the unerlying function call.

### 2.5 Exception Handling

Although exception handling has been officially accepted into the language by the ANSI committee (November 1990) it is not yet supported by all C++ compilers. LAPACK++ will use C++ exception handling as soon as it becomes widely available. Currently, Version 0.9 uses a macro mechanism to simulate `throw` expressions so that the library code will work correctly when exception handling becomes supported:

```
#define throw throw_
inline void throw(const char *message)
```

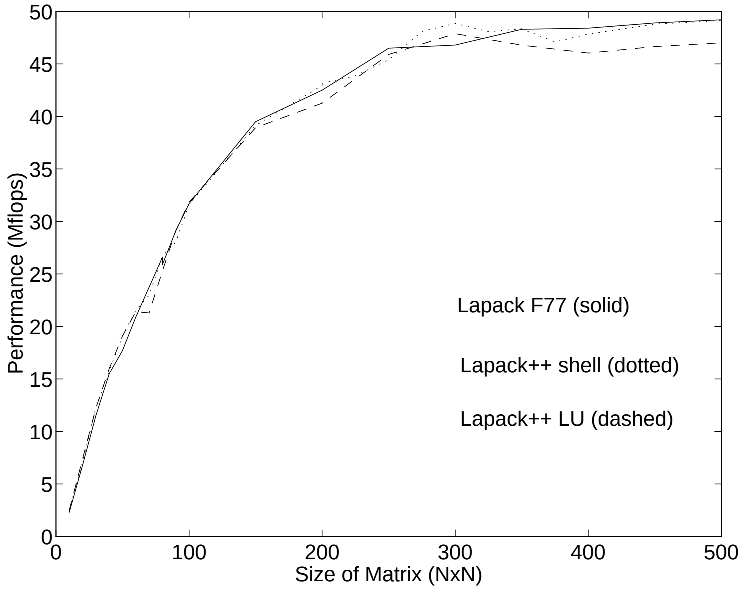


Figure 2: Performance of LAPACK++ LU factorization on a IBM RS/6000 Model 550 workstation, using GNU g++ v. 2.3.1 and BLAS routines from the IBM ESSL library. The results are nearly identical to the Fortran LAPACK performance. The Lapack++ shell is call the Fortan `dgetrf()`, the Lapack++ LU implements the right-looking LU algorithm in C++.

```
{
   cerr << "Exception: " << message << endl;
   exit(1);
}
```

Transfer of control does not correspond to a "try" block, but rather exits the program with the proper diagnostic message. This is a similar behavior to the real exception handling if there was no explicit user-supplied try block.

# 3 LAPACK++ Matrix Objects

The fundamental objects in LAPACK++ are numerical vectors and matrices; however, LAPACK++ is **not** a general-purpose array package. Rather, LAPACK++ is a self-contained interface consistsing of only the minimal number of classes to support the functionality of the LAPACK algorithms and data strucutres.

LAPACK++ matrices can be referenced, assigned, and used in mathematical expressions as naturally as if they were an integral part of C++; the matrix element $a_{ij}$, for example, is referenced as `A(i,j)`. In keeping with the indexing convention of C++, matrix subscripts begin at zero. Thus, `A(0,0)` denotes the first element of `A`. Internally, LAPACK++ matrices are typically stored in column-order for compatibility with Fortran subroutines and libraries. **Notice that the prototype version 0.9 of LAPACK++ does not yet support templates; future versions will.**

Various kinds of matrices are supported: banded, symmetric, Hermitian, packed, triangular, tridiagonal, bidiagonal, and nonsymmetric. Rather than have an unstructured collection of matrix classes, LAPACK++ maintains a class hierarchy (figure 3) to exploit commonality in the derivation of the fundamental matrix types. This limits much of the code redundancy and allows for an open-ended design which can be extended as new matrix storage structures are introduced.

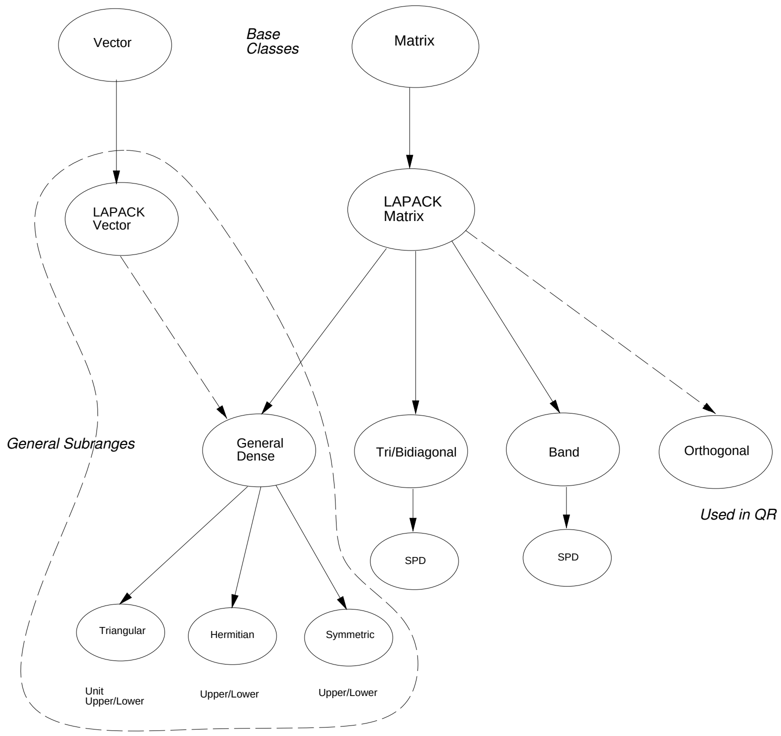


Figure 3: The Matrix Hierarchy of LAPACK++. Rectangular and Vector classes allow indexing of any rectangular submatrix by reference.

## 3.1 Naming Conventions

Matrix classes andother related data types specific to LAPACK++ begin with the prefix "`La`" to avoid naming conflicts with user-defined or third-party matrix packages. The list of valid names is a subset of the nomenclature show in figure 4.

$$\texttt{La}\left\{\begin{array}{c}\texttt{Gen}\\ \texttt{Orthog}\\ \left[\begin{array}{l}\texttt{Symm}\\ \texttt{Hermitian}\\ \texttt{SPD}\end{array}\right]\left\{\begin{array}{l}\texttt{Band}\\ \texttt{Tridiag}\\ \texttt{Bidiag}\\ \texttt{Packed}\end{array}\right\}\\ {[}\texttt{Unit}{]}\left\{\begin{array}{c}\texttt{Upper}\\ \texttt{Lower}\end{array}\right\}\{\texttt{Triang}\}\end{array}\right\}\texttt{Mat}\left\{\begin{array}{l}\texttt{Float}\\ \texttt{Double}\\ \texttt{Complex}\end{array}\right\}$$

Figure 4: LAPACK++ matrix nomenclature. Items in square brackets are optional.

## 3.2 Fundamental Matrix Operations

In its most general formulation, a matrix is a data structure representing an ordered collection of elements that can be accessed by an integer pair. In addition to their most common representation as rectangular arrays, matrices can also be represented as linked lists (unstructured sparse), in packed formats (triangular), as a series of Householder transformations (orthogonal), or stored by diagonals. Nevertheless, these various representations exhibit some commonality. The following operations are available for all matrix types:

- **Declarations.** Matrices can be constructed dynamically by specifying their size as a pair of non-negative integers, as in

  ```
  LaGenMatDouble A(M,N)
  ```

- **Indexing.** Individual elements of the matrix can be accessed by an integer pair $(i, j)$ where $0 \le i < M$ and $0 \le j < N$. The indexing syntax is for a matrix `A` is the natural

  ```
  A(i,j)
  ```

  notation, and can also be used as a destination for an assignment as in `A(0,3) = 3.14`. A runtime error is generated if the index is out of the matrix bounds or outside of specified storage scheme (for example, accesing outside the non-zero portion of a banded matrix). This index range checking can be turned off by defining `LA_NO_BOUNDS_CHECK`.

- **Assignment.** The basic matrix assignment is by copying, and is denoted by the `=` operator, as in

  ```
  A = B
  ```

  The sizes of `A` and `B` must conform.

- **Referencing.** The unnecessary copying of matrix data elements has been avoided by employing the `reference()` method, as in

  ```
  A.ref(B)
  ```

  to assign the data space of $B$ to the matrix $A$. The matrix elements are shared by both $A$ and $B$, and changes to the data elements of one matrix will effect the other.

Furthermore, all matrix objects employ a mechanism to intelligently free their data space when leaving their scope of existence. By utilizing a reference counting scheme that keeps track of the number of aliases utilizing all or part of its data space, a matrix object can safely recycle unused data space, so that memory leaks are avoided.

## 3.3 General Rectangular Matrices

One of the fundamental matrix types in LAPACK++ is a nonsymmetric rectangular matrix. The corresponding LAPACK++ names are given as LaGenMat
for Lapack General Matrix. The *Type* can be `Float`, `Double`, or `Complex`. Matrices in this category have the added property that submatrices can be efficiently accessed and referenced in matrix expressions. This is a necessity for describing block-structured algorithms.

### 3.3.1 Declarations

General LAPACK++ matrices may be declared (constructed) in various ways:

```
#include <lapack++.h>
float d[4] = {1.0, 2.0, 3.0, 4.0};

LaGenMatDouble    A(200,100)              // 1
LaGenMatComplex   B;                      // 2
LaGenMatDouble    D(A);                   // 3
LaGenMatFloat     E(d, 2, 2)              // 4
```

Line (1) declares `A` to be a rectangular 200x100 matrix. The elements are uninitalized and should be set to some value, with all of its elements initialized to 0.0. Line (2) declares `B` to be an empty (uninitialized) matrix of complex numbers. Until B becomes initialized, any attempt to reference its elements will result in a run time error. Line (3) illustrates the copy constructor; `D` is a copy of `A`. Finally, line (4) demonstrates how one can initialize a 2x2 matrix with the data from a standard C++ vector. The values are initalized in column-major form, so that the first column of `E` contains $\{1.0, 2.0\}$, and the second column contains $\{3.0, 4.0\}$.

### 3.3.2 Submatrices

Blocked linear algebra algorithms utilize submatrices as their basic unit of computation. It is crucial that submatrix operations be highly optimized. Because of this, LAPACK++ provides mechanisms for accessing rectangular subregions of a general matrix. These regions are accessed by *reference*, that is, without copying data, and can be used in any matrix expression.

Submatrices in LAPACK++ are denoted by specifying a subscript range through the `LaIndex()` function. For example, the 3x3 matrix in the upper left corner of `A` is denoted as

```
A( LaIndex(0,2), LaIndex(0,2) )
```

This references $A_{ij}, i = 0,1,2\ j = 0,1,2$, and is equivalent to the `A(0:2,0:2)` colon notation used elsewhere. Submatrix expressions may be also be used as a destination for assignment, as in

```
A( LaIndex(0,2), LaIndex(0,2) ) = 0.0;
```

which sets the 3x3 submatrix of A to zero. The index notation has an optional third argument denoting the stride value,

```
LaIndex(start, end, increment)
```

If the `increment` value is not specified it is assumed to be one. The expression `LaIndex(s, i, e)` is equivalent to the index sequence

$$s,\ \ s+i,\ \ s+2i,\ \ \ldots\ \ s+\lfloor\frac{e-s}{i}\rfloor i$$

Of course, the internal representation of an index is not expanded to a full vector, but kept in its compact (start,increment,end) format. The increment values may be negative and allow one to traverse a subscript

| Storage | Triangular matrix $A$ | Storage in array A |
|---|---|---|
| `Upper` | $\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ & a_{11} & a_{12} & a_{13} \\ & & a_{22} & a_{23} \\ & & & a_{33} \end{pmatrix}$ | $\begin{matrix} a_{00} & a_{01} & a_{02} & a_{03} \\ * & a_{11} & a_{12} & a_{13} \\ * & * & a_{22} & a_{23} \\ * & * & * & a_{33} \end{matrix}$ |
| `Lower` | $\begin{pmatrix} a_{00} & & & \\ a_{10} & a_{11} & & \\ a_{20} & a_{21} & a_{22} & \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$ | $\begin{matrix} a_{00} & * & * & * \\ a_{10} & a_{11} & * & * \\ a_{20} & a_{21} & a_{22} & * \\ a_{30} & a_{31} & a_{32} & a_{33} \end{matrix}$ |

Figure 5: Internal storage pattern for an example 5x5 triangular matrix.

range in the opposite direction, such as in `(10,-1,7)` which denotes the sequence $\{10, 9, 8, 7\}$. Indices can be named and used in expressions, as in the following submatrix assignments,

```
LaGenMatDouble A(10,10), B, C;                    // 1
LaIndex I(1,9,2),                                 // 2
LaIndex J(1,3,2);                                 // 3


B = A(I,I);                                       // 4
B(2,3) = 3.1;                                     // 5
C = B(LaIndex(2,2,4), J);                         // 6
```

In lines (2) and (3) we declare indices `I` $= \{1, 3, 5, 7, 9\}$, and `J` $=\{1, 3\}$. Line (4) sets `B` to the specified 5x5 submatrix of `A`. The submatrix B can used in any matrix expression, including accessing its individual elements, in line (5). Note that `B(2,3)` is the memory location as `A(5,7)`. Line (6) assigns the 2x2 submatrix of B to C. Note that C can also be thought of as `A( LaIndex(5,2,9), LaIndex(3,2,7))`.

Although LAPACK++ submatrix expressions allow one to access non-contiguous row or columns, many of the LAPACK routines only allow submatrices with unit stride in the column direction. Calling an LAPACK++ routine with a non-contiguous submatrix columns *may* cause data to be copied into a contiguous submatrix and can optionally generate a runtime warning to advise the programmer that data copying has taken place.

## 3.4 Triangular Matrices

**Note: This section is still under development.**

The triangular matrix types consist of

$$\left\{ \begin{array}{l} \mathit{Upper} \\ \mathit{Lower} \\ \mathit{Unit} \end{array} \right\} \{Triangular\} \left\{ \begin{array}{l} \textit{single precision} \\ \textit{double precision} \\ \textit{double precision complex} \end{array} \right\}$$

## 3.5 Symmetric, Hermitian and SPD Matrices

Matrices in this category are classified by their symmetry properties. Symmetric matrices have $a_{ij} = a_{ji}$; Hermitian matrices have $a_{ij} = a^*_{ji}$, where $^*$ denotes complex conjugation; symmetric positive definite matrix, $A$, has the added property that $x^T Ax > 0$ for any $x$.

| Storage | Hermitian matrix $A$ | Storage in array A |
|---|---|---|
| `Upper` | $\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ \bar{a}_{01} & a_{11} & a_{12} & a_{13} \\ \bar{a}_{02} & \bar{a}_{12} & a_{22} & a_{23} \\ \bar{a}_{03} & \bar{a}_{13} & \bar{a}_{23} & a_{33} \end{pmatrix}$ | $\begin{matrix} a_{00} & a_{01} & a_{02} & a_{03} \\ * & a_{11} & a_{12} & a_{13} \\ * & * & a_{22} & a_{23} \\ * & * & * & a_{33} \end{matrix}$ |
| `Lower` | $\begin{pmatrix} a_{00} & \bar{a}_{10} & \bar{a}_{20} & \bar{a}_{30} \\ a_{10} & a_{11} & \bar{a}_{21} & \bar{a}_{31} \\ a_{20} & a_{21} & a_{22} & \bar{a}_{32} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$ | $\begin{matrix} a_{00} & * & * & * \\ a_{10} & a_{11} & * & * \\ a_{20} & a_{21} & a_{22} & * \\ a_{30} & a_{31} & a_{32} & a_{33} \end{matrix}$ |

Figure 6: Storage pattern for Symmetric and Hermitian matrices.

Matrix types in each class of this group consists of two constituents: the mathematical characteristics (e.g. symmetric, Hermitian), and the underlying matrix element type (e.g., float, dcomplex). The possible matrix types include

$$\left\{ \begin{array}{l} \textit{symmetric} \\ \textit{Hermitian} \\ \textit{SPD} \end{array} \right\} \left\{ \begin{array}{l} \textit{single precision} \\ \textit{double precision} \\ \textit{single precision complex} \\ \textit{double precision complex} \end{array} \right\}$$

All possible combinations are allowed, except for redundancies such as the equivalence beetween real Hermitian and real symmetric matrices. The corresponding LAPACK++ matrix classes are given as

| | |
|---|---|
| `LaSymmMatDouble` | Rectangular symmetric |
| `LaSPDMatDouble` | Rectangular SPD |
| `LaHermitianMatComplex` | Rectangular Hermitian |

Internally, symmetric and Hermitian matrices may be stored in an `Upper` or `Lower` format as shown in figure 6.

### 3.6 Banded Matrices

Square matrices whose sparsity pattern has non-zeros close to the diagonal can be efficiently represented as banded matrices. An $N \times N$ banded matrix is specified by its size $N$, the number of subdiagonals $kl$, and the number of superdiagonals $ku$. In practice, this storage scheme should be used only when $kl$, and $ku$ are much smaller than $N$. These matrices can be declared as

```
LaBandedMatDouble B(N, kl, ku);
LaBandedMatDouble A;
```

A reference to element $B_{ij}$ is expressed as `B(i,j)`, and if the macro `LA_NO_BOUNDS_CHECK` is undefined, an error is generated if `(i,j)` lies outside the bands. Banded matrices may be copied and assigned as

```
A.ref(B);          // shallow assignment
A = B;             // copy
```

The $i$th diagonal of `B` is accessed as `B(i)`, and `B(-i)` is the $i$th subdiagonal ($i > 0$). Thus `B(0)` is the main diagonal, The result is a vector with (possible non-unit stride),

```
LaVectorDouble d = B(-2);     // 2nd subdiagonal of B
```

| Band matrix $A$ | Band storage in array AB |
|---|---|
| $\begin{pmatrix} a_{00} & a_{01} & & & \\ a_{10} & a_{11} & a_{12} & & \\ a_{20} & a_{21} & a_{22} & a_{23} & \\ & a_{31} & a_{32} & a_{33} & a_{34} \\ & & a_{42} & a_{43} & a_{44} \end{pmatrix}$ | $\begin{matrix} * & a_{01} & a_{12} & a_{23} & a_{34} \\ a_{00} & a_{11} & a_{22} & a_{33} & a_{44} \\ a_{10} & a_{21} & a_{32} & a_{43} & * \\ a_{20} & a_{31} & a_{42} & * & * \end{matrix}$ |

Figure 7: Storage pattern for banded 5x5 matrix with 2 superdiagonals and 1 subdiagonal.

whose size is the length of the particular diagonal, not the matrix size $N$. Note that this diagonal indexing expression can be used as any `LaVector`. For example, it is perfectly legal to write

```
B(-2)(LaIndex(0,3)) = 3.1;
```

to set the first four elements of the second subdiagonal of B to the value of 3.1.

Accessing out of the matrix band generates a run-time error, unless `LA_NO_BOUNDS_CHECK` is set.

### 3.6.1 Triangular Banded Matrices

Triangular banded matrices are declared, stored, and accessed in a similar format, except that $kl = 0$ for upper triangular, and $ku = 0$ for lower triangular matrices:

```
LaUpperTriangBandedMatDouble  U(N, kl);
LaLowerTriangBandedMatDouble  L(N, ku);
```

Diagonals and individual elements are accessed the same way as general banded matrices. Triangular banded matrices can also be aliases for the upper or lower region of general banded matrices in the following exmample

```
LaBandedMatDouble B(N, kl, ku) = 0.0;
LaUpperTriangBandedMatDouble U;
LaLowerTriangBandedMatDouble L;

U.ref(B);
L.ref(B);

U(0,0) = 1.0;           // B(0,0) and L(0,0) also set to 1.0
                        //    (diagonal is shared by L and U.)
```

### 3.6.2 Symmetric and Hermitian Banded Matrices

Symmetric and Hermitian banded matrices with $kd$ subdiagonals and superdiagonals are specified as

```
LaSymmBandedMatDouble S(N, kd);
LaHermBandedMatComplex C(N, kd);
```

Diagonals and individual elements are accessed the same way as above. If `LA_NO_BOUNDS_CHECK` is undefined, accessing out of the range of diagonal bands generates a run-time error.

### 3.7 Tridiagonal Matrices

Although bidiagonal and tridiagonal matrices are special cases of the more general banded matrix structure, their occurrence is so common in numerical codes that it is advantageous to treat them as a special case. Unlike general banded matrices, tridiagonal matrices are stored by diagonals rather than columns. A tridiagonal matrix of order $N$ is stored in three one-dimensional arrays, one of length $N$ containing the diagonal elements and two of length $N-1$ containing the subdiagonal and superdiagonal elements in elements 0 through $N-2$. This ensures that diagonal elements are in consecutive memory locations.

Tridiagonal NxN matrices are constructed as

```
LaTridiagMatDouble B(N,N);
LaTridiagMatDouble A;
```

reserving a space of 3N (see [1]) elements, or from user data as

```
LaTridiagMatDouble T(N, d, dl, du)
```

where `*d`, `*dl`, and `*du` are contiguous vectors that point to the diagonal, subdiagonal, and superdiagonals, respectively. Note that `*d` must point to N elements, but `*du` and `*dl` can point to only N-1.

Matrix elements can be accessed as `T(i,j)`, where $|(i-j)| \leq 2$, (i.e. elements on the diagonal, sub or superdiagonal) otherwise there is a bound-check error generated at run time.

Since the largest rectangular submatrix of a tridiagonal matrix has three elements, index ranges are of limited use and are therefore not implemented with tridiagonal matrices.

# 4 Driver Routines

## 4.1 Linear Equations

This section provides LAPACK++ routines for solving linear systems

$$Ax = b.$$

where $A$ is the **coefficient matrix**, $b$ is the **right hand side**, and $x$ is the **solution**. $A$ is assumed to be square matrix of order $n$, although underlying computational routines allow for $A$ to be rectangular. For several right hand sides, we write

$$AX = B$$

where the columns of $B$ are individual right hand sides, and the columns of $X$ are the corresponding solutions. The task is to find $X$, given $A$ and $B$. The coefficient matrix $A$ can be of the types show in figure 4. Note that for real (non-complex) matrices, *symmetric* and *Hermitian* are equivalent.

The basic syntax for a linear equation driver in LAPACK++ is given by

```
LaLinSolve(A, X, B);
```

The matrices `A` and `B` are input, and `X` is the output. `A` is an $M \times N$ matrix of one of the above types. Letting $nrhs$ denote the number of right hand sides in eq. 4.1, `X` and `B` are both rectangular matrices of size $N \times nrhs$.

This version requires intermediate storage of $\approx M * (N + nrhs$ elements. Section 5 describes how to use factorization classes to reduce this storage requirement at the expense of overwriting $A$ and $B$.

In cases where no additional information is supplied, the LAPACK++ routines will attempt to follow an intelligent course of action. For example, if `LaLinSolve(A,X,B)` is called with a non-square MxN matrix, the solution returned will be the linear least square that minimizes $||Ax - b||_2$ using QR factorization. Or, if A is SPD, then the Cholesky factorization will be used. Alternatively, one can directly specify the exact factorization method, such as `LU_factor(F, A)`. In this case, if A is non-square, only the factors return represent only a partial factorization of the upper square portion of A.

Error conditions in performing the `LaLinSolve()` operations can be retrieved via the `LaLinSolveInfo()` function, which returns information about the last called `LaLinSolve()`. A zero value denotes a successful completion. A negative value of $-i$ denotes that the $i$th argument was somehow invalid or inappropriate. A positive value of $i$ denotes that in the LU decomposition, $U(i,i) = 0$; the factorization has been completed but the factor $U$ is exactly singular, so the solution could not be computed. In this case, the value returned by `LaLinSolve()` is a null (0x0) matrix.

## 4.2 Memory Optimizations: Factorizing in place

When using large matrices that consume a significant portion of available memory, it may be beneficial to remove the requirement of storing intermediate factorizaton representations at the expense of destroying the contents of the input matrix A. For most matrix factorizations we require temporary data structures roughly equal to the size of the original input matrix. (For general banded matrices, one may need even slight more, see section 3.6.) For example, for a square $N \times N$ dense nonsymmetric factorization, the temporary memory requirement can be reduced from $N \times (N + nrhs + 1)$ elements to $N \times 1$ Such memory-efficient factorizations are accomplished with the `LaLinSolveIP()` routine:

```
LaLinSolveIP(A, X, B);
```

Here the contents of $A$ are overwritten (with the respective factorization ), and $B$ is overwritten by the soultion. It is also explicitly returned by the function so that it matches the return type of `LaLinSolve()`. In the line above, both `X` and `B` refer to the same memory locations.

Table 1: LAPACK++ Drivers for Linear Equations: Rectangular Matrices.

General

```
LaGenMat<t> A(M,N);
LaGenMat<t> B(M,nrhs), X(M,nrhs);
LaGenFact<t> F;

LaLinSolve(A, X, B);
LaLinSolveInPlace(A, X, B);
LaLinSolve(F, X, B);
```

Symmetric

```
LaUpperSymmMat<t> A(N,N);
LaLowerSymmMat<t> A(N,N);
LaGenMat<t> B(N, nrhs), X(N, nrhs);
LaSymmFact<t> F;

LaLinSolve(A, X, B);
LaLinSolveInPlace(A, X, B);
LaLinSolve(F, X, B);
```

Symmetric Positive Definite

```
LaUpperSPDMat<t> A(N,N);
LaLowerSPDMat<t> A(N,N);
LaGenMat<t> B(N, nrhs), X(N, nrhs);
LaSPDFact<t> F;

LaLinSolve(A, X, B);
LaLinSolveInPlace(A, X, B);
LaLinSolve(F, X, B);
```

Complex Symmetric

```
LaUpperSymmMat<t> A(N,N);
LaLowerSymmMat<t> A(N,N);
LaGenMat<t> B(N, nrhs), X(N, nrhs);
LaSymmFact<t> F;

LaLinSolve(A, X, B);
LaLinSolveInPlace(A, X, B);
LaLinSolve(F, X, B);
```

Table 2: LAPACK++ Drivers for Linear Equations: Tridiagonal Matrices.

General

```
LaTriadMat<t> A(N,N);
LaGenMat<t> B(N,nrhs), X(N,nrhs);
LaGenFact<t> F;

LaLinSolve(A, X, B);
LaLinSolveInPlace(A, X, B);
LaLinSolve(F, X, B);
```

Symmetric Positive Definite

```
LaUpperTriadSPDMat<t> A(N,N);
LaLowerTriadSPDMat<t> A(N,N);
LaGenMat<t> B(N, nrhs), X(N, nrhs);
LaSPDFact<t> F;

LaLinSolve(A, X, B);
LaLinSolveInPlace(A, X, B);
LaLinSolve(F, X, B);
```

Table 3: LAPACK++ Drivers for Linear Equations: Banded Matrices.

General

```
LaBandedMat<t> A(N,N);
LaGenMat<t> B(N,nrhs), X(N,nrhs);
LaBandedFact<t> F;

LaLinSolve(A, X, B);
LaLinSolve(F, X, B);
```

Symmetric Positive Definite

```
LaUpperBandedSPDMat<t> A(N,N);
LaLowerBandedSPDMat<t> A(N,N);
LaGenMat<t> B(N, nrhs), X(N, nrhs);
LaSPDFact<t> F;

LaLinSolve(A, X, B);
LaLinSolve(F, X, B);
```

# 5 Factorization Classes

Factorization classes are used to describe the various types of matrix decompositions: LU, Cholesky ($LL^T$), QR, and singular-value decompositions (SVD). The driver routines of LAPACK++ typically choose an appropriate factorization, but the advanced user can express specific factorization algorithms and their variants for finer control of their application or meeting stiff memory storage and performance requirements.

In an object-oriented paradigm it is natural to encapsulate the factored representation of a matrix in a single object. An LU factorization, for example, returns the upper and unit-lower triangular factors, $L$ and $U$, as well the pivoting information that describes how the rows were permuted during the factorizaion process. The representation of the $L$ and $U$ factors is incomplete without this information. Rather than store and manage these components seperately, a factoriation class is used as in

```
LaGenMatDouble A, B, X;
LaGenFactDouble F;

LaLUFactor(F, A);
LaSolve(F, X, B);
```

More importantly, the various factorization components can be extracted from `F` as in,

```
LaUnitLowerTriangMatDouble L;
LaUpperTriangMatDouble U;
LaGenMatDouble Y;

L = F.L();
U = F.U();

Y = LaSolve(L, B);
X = LaSolve(U, Y);
```

Here we solve $AX = B$, by first solving the lower triangular system, $LY = B$, and the upper triangular system, $UX = Y$. (The pivot information is stored in both L and U.) The `LaGenFact` object can also be used directly in the triangular forward and backsolve phase to solve $(LU)X = B$ by calling

```
LaSolve(F, X, B);
```

## 5.1 Optimizations: Factoring in Place

By default, the matrix factorization does not alter the contents of the original matrix or overwrite it. Nevertheless, the ability to "factor in place" is crucial when dealing with realistic memory constraints on large matrices, and is a necessity to implement blocked linear algebra algorithms efficiently.

Here we utilize the "in place" versions of computational routines (see section 4.2), which overwrite the matrix A to conserve space:

```
LaGenMatDouble A;
LaGenFactDouble F;

LaLUFactorIP(F, A);
```

# 6 Programming Examples

To illustrate what programming with LAPACK++ looks like to scientific and engineering code developers, this section provides a few code examples. These examples are presented here for illustrative purposes, yet we have tried to use realistic examples that accurately display a level of sophistication encourted in realistic applications.

## 6.1 Polynomial data fitting

This code example solves the liner least squares problem of fitting $N$ data points $(x_i, y_i)$ to a $d$th degree polynomial equation

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_d x^d$$

using QR factorization. Given the two vectors $x$ and $y$ it returns the vector of coefficients $\mathtt{a} = \{a_0, a_1, a_2, \ldots a_{d-1}\}$. It is assumed that $N \gg d$. The solution arises from solving the overdetermined Vandermonde system $Xa = y$:

$$\begin{bmatrix} 1 & x_0^1 & x_0^2 & \ldots & x_0^d \\ 1 & x_1^1 & x_1^2 & \ldots & x_1^d \\ \vdots & & & & \vdots \\ 1 & x_{N-1}^1 & x_{N-1}^2 & \ldots & x_{N-1}^d \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix}$$

in the least squares sense, i.e., minimizing $||Xa - y||_2$. The resulting code is shown in figure 8.

| *Filename* | *Function* | *Description* |
| --- | --- | --- |
| | | |
| blas++.h | Vec * Vec | Vector * Vector |
| | Vec + Vec | Vector + Vector |
| | Vec - Vec | Vector - Vector |
| | Mat * Vec | General Matrix * Vector |
| | Mat * Vec | Banded Matrix * Vector |
| | Mat * Vec | Symmetric Matrix * Vector |
| | Mat * Vec | Symmetric Banded Matrix * Vector |
| | Mat * Vec | SPD Matrix * Vector |
| | Mat * Vec | Lower Triang Matrix * Vector |
| | Mat * Vec | Upper Triang Matrix * Vector |
| | Mat * Vec | Unit Lower Triang Matrix * Vector |
| | Mat * Vec | Unit Upper Triang Matrix * Vector |
| | Mat + Mat | General Matrix + General Matrix |
| | Mat - Mat | General Matrix - General Matrix |
| | Mat * Mat | General Matrix * General Matrix |
| | Mat * Mat | Symmetric Matrix * General Matrix |
| | Mat * Mat | Lower Triang Matrix * General Matrix |
| | Mat * Mat | Upper Triang Matrix * General Matrix |
| | Mat * Mat | Unit Lower Triang Matrix * General Matrix |
| | Mat * Mat | Unit Upper Triang Matrix * General Matrix |
| | Norm_Inf(Vec) | Infinity Norm of a Vector |
| | Norm_Inf(Mat) | Infinity Norm of a Matrix |
| | Mach_eps_float() | Returns Machine Epsilon - float |
| | Mach_eps_double() | Returns Machine Epsilon - double |

# 7 Appendix A: Direct Interfaces to LAPACK and BLAS Routines

| *Filename* | *Function* | *Description* |
|---|---|---|
| | | |
| blas1++.h | Blas_Norm1(Vec) | One Norm of a Vector |
| | Blas_Norm2(Vec) | $nrm2 \leftarrow \|x\|_2$ |
| | Blas_Add_Mult(double,Vec,Vec) | $y \leftarrow \alpha x + y$ |
| | Blas_Add_Mult_IP(double,Vec,Vec) | $y \leftarrow \alpha y + = x$ |
| | Blas_Copy(Vec,Vec) | $y \leftarrow x$ |
| | Blas_Dot_Prod(Vec,Vec) | $dot \leftarrow \alpha + x^T y$ |
| | Blas_Apply_Plane_Rot(Vec,Vec) | Apply Plane Rotation |
| | Blas_Gen_Plane_Rot(double x 4) | Generate Plane Rotation |
| | Blas_Scale(double,Vec) | $x \leftarrow \alpha x$ |
| | Blas_Swap(Vec,Vec) | $x \leftrightarrow y$ |
| | Blas_Index_Max(Vec) | Returns Maximum Vector Element |
| | | |
| blas2++.h | Blas_Mat_Vec_Mult(GenMat,Vec) | Gen Mat * Vec |
| | Blas_Mat_Vec_Mult(BandMat,Vec) | Band Mat * Vec |
| | Blas_Mat_Vec_Mult(SymmMat,Vec) | Symm Mat * Vec |
| | Blas_Mat_Vec_Mult(SymmBandMat,Vec) | Symm Band Mat * Vec |
| | Blas_Mat_Vec_Mult(SpdMat,Vec) | SPD Mat * Vec |
| | Blas_Mat_Vec_Mult(LowerTriangMat,Vec) | Lower Triang Mat * Vec |
| | Blas_Mat_Vec_Mult(UpperTriangMat,Vec) | Upper Triang Mat * Vec |
| | Blas_Mat_Vec_Mult(UnitLowerTriangMat,Vec) | Unit Lower Triang Mat * Vec |
| | Blas_Mat_Vec_Mult(UnitUpperTriangMat,Vec) | Unit Upper Triang Mat * Vec |
| | Blas_Mat_Vec_Solve(LowerTriangMat,Vec) | Lower Triang Mat/Vec Solve |
| | Blas_Mat_Vec_Solve(UpperTriangMat,Vec) | Upper Triang Mat/Vec Solve |
| | Blas_Mat_Vec_Solve(UnitLowerTriangMat,Vec) | Unit Lower Triang Mat/Vec Solve |
| | Blas_Mat_Vec_Solve(UnitUpperTriangMat,Vec) | Unit Upper Triang Mat/Vec Solve |
| | Blas_R1_Update(GenMat,Vec,Vec) | $A \leftarrow \alpha x y^T + A$ |
| | Blas_R1_Update(SymmMat,Vec,Vec) | $A \leftarrow \alpha x x^T + A$ |
| | Blas_R1_Update(SpdMat,Vec,Vec) | $A \leftarrow \alpha x x^T + A$ |
| | Blas_R2_Update(SymmMat,Vec,Vec) | $A \leftarrow \alpha x y^T + \alpha y x^T + A$ |
| | Blas_R2_Update(SpdMat,Vec,Vec) | $A \leftarrow \alpha x y^T + \alpha y x^T + A$ |
| | | |
| blas3++.h | Blas_Mat_Mat_Mult(GenMat,GenMat) | Gen Mat * Gen Mat |
| | Blas_Mat_Trans_Mat_Mult(GenMatT,GenMat) | Gen Mat Transpose * Gen Mat |
| | Blas_Mat_Mat_Trans_Mult(GenMat,GenMatT) | Gen Mat * Gen Mat Transpose |
| | Blas_Mat_Mat_Mult(LowerTriangMat,GenMat) | Lower Triang Mat * Gen Mat |
| | Blas_Mat_Mat_Mult(UpperTriangMat,GenMat) | Upper Triang Mat * Gen Mat |
| | Blas_Mat_Mat_Mult(UnitLowerTriangMat,GenMat) | Unit Lower Triang Mat * Gen Mat |
| | Blas_Mat_Mat_Mult(UnitUpperTriangMat,GenMat) | Unit Upper Triang Mat * Gen Mat |
| | Blas_Mat_Mat_Mult(SymmMat,GenMat,GenMat) | Symm Mat * Gen Mat |
| | Blas_Mat_Mat_Solve(LowerTriangMat,GenMat) | Lower Triang Mat/Gen Mat Solve |
| | Blas_Mat_Mat_Solve(UpperTriangMat,GenMat) | Upper Triang Mat/Gen Mat Solve |
| | Blas_Mat_Mat_Solve(UnitLowerTriangMat,GenMat) | Unit Lower Triang Mat/Gen Mat Solve |
| | Blas_Mat_Mat_Solve(UnitUpperTriangMat,GenMat) | Unit Upper Triang Mat/Gen Mat Solve |
| | Blas_R1_Update(SymmMat,GenMat) | $C \leftarrow \alpha A A^T + \beta C$ |
| | Blas_R2_Update(SymmMat,GenMat,GenMat) | $C \leftarrow \alpha A B^T + \alpha B A^T + \beta C$ |

# 8 Appendix B: Example LAPACK++ Wrappers

This section provides examples of the actual LAPACK++ source code for some of the key routines.

**Note: This section is mainly for us, althougth other C++ developers may want to build interfaces from their objects so that integrate with LAPACK++. This type of activity should be encouraged so that the library will be compatible with most commerical C++ package and gain wider acceptance.**

## 8.1 LU Factorization

```
#include <lapack++.h>

LU_factor(LaGenMat<double> &A, LaGenFact<double> &F)
{

    int M = A.size(0), N=A.size(1);
    int info;
    int factor_in_place = 0;

    if (M==0 || N==0)
    {
        info = -1;      // is input matrix invalid?
        return info;
    }

    if (F.LU == A) // did we specify to factor in place?
    {
        A.detach();     // mark A so we can't reference it after this
                        // function call;
    }
    else if (F.LU.size(0)<M || F.LU.size(1)<N)  //  is F too small?
    {
        F.LU.resize(M,N);
        F.LU.copy(A);           // make an actual copy
    }

    if (F.ipiv.size() < M) // is pivot vector large enough?
        F.ipiv.resize(M);

    F.transpose_ = A.transpose_;    // save status if we wish to solve
                                    // A' X = B


    SGETRF(&M, &N, &F.LU(0,0), & F.LU.dim[0], &ipiv(0), &info);

    if (info >= 0) return info;     // successful exit;

    else
        return -3; // internal error (1 more than the number of input
                   // paramters.)

}
```

## 8.2 LU Solve

```
#include <lapack++.h"

LU_solve(LaGenFact<double> &F, LaGenMat<double> &X, LaGenMat<double> &B)
{

    int N = B.size(0), Nrhs=B.size(1);
    int info;
    int factor_in_place = 0;

    if (F.LU.size(0) < N || F.LU.size(1) < N)
        return -1;      // factor is too small for this B matrix.

    if (M==0 || N==0)
        return -3;      // B matrix is degenerate.


    if (X == B || &X != &B) // did we specify to solve in place with
                            // different aliases? (i.e. we didn't
                            // call function with (F, B, B).
    {
        B.detach();     // mark B so we can't reference it after this
                        // function call.
    }
    else
        X.copy(B);      // don't solve in place, but copy B.


    char trans = ( F.tranpose_ ? 'T' : 'N' );
    int lda = X.dim(0);

    SGETRS(&trans, &N, &Nrhs, &F.LU(0,0), & F.LU.dim[0], &ipiv(0),
            &X(0,0), &lda, &info);


    if (info == 0) return info;     // successful exit;

    else
        return -4; // internal error (1 more than the number of input
                   // paramters.)

}
```

## 8.3 Condition Number Estimate

```
#include <lapack++.h>

int rcond(LaGenFact<double> &F, double normA, char *s, double *cond)
{

    int N = F.LU.size(0);
    int info;
    char normletter;

    if (F.LU.size(1) < N)
        return -1;      // factor must be at least square

    if (normA < 0)
        return -2;      // all matrix norms are positive

    if (*s == 'i' || *s == 'I')
        normletter = 'I';
    else if (*s == '1' || *s == 'O')
        normletter = '1';
    else
        return -3;      // bad string to specify norm


    LaVector<double> work(2*N);
    LaVector<int>   iwork(N);
    int lda = F.LU.dim[0];

    SGECON_( &normletter, &N, &F.LU(0,0), &lda, cond, &work(0,0),
        &iwork(0,0), &info);

    if (info == 0) return info;

    else
        return -4; // internal error (1 more than number of arguments)
}



int rcond(LaGenMat<double> A, double *cond)
{
    LaGenFact<double>  F;
    int info;

    if (A.size(0) == 0 || A.size(1) == 0)
        return -1;

    LU_factor(F,A);

    if (rcond(F, norm(A, '1'), cond) == 0)
        return 0;
    else
        return -3; // internal error (1 more than number of arguments)
}
```

## References


[1] E. Anderson, Z. Bai, C. Bischof, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK: A portable linear algebra library for high-performance computers*, Computer Science Dept. Technical Report CS-90-105, University of Tennessee, Knoxville, 1990. (LAPACK Working Note 20).

[2] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, PA, 1979.

[3] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft., 16 (1990), pp. 1–17.

[4] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, *Basic Linear Algebra Subprograms for FORTRAN usage*, ACM Trans. Math. Soft ., 5 (1979), pp. 308–323.

[5] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide*, vol. 6 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2 ed., 1976.

[6] B. J. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

```
LaVectorDouble poly_fit(LaVectorDouble x, LaVectorDouble y, int d)
{
    int N = min(x.size(), y.size());

    LaGenMatDouble P(N,d);
    LaVectorDouble a(d);

    for (i=0; i<N; i++)     // construct Vandermonde matrix
    {
        x_to_the_j = 1;
        for (j=0; j<d; j++)
        {
            P(i,j) = x_to_the_j;
            x_to_the_j *= x(i);
        }
    }
    a = LaQRLinearSolveInP(P, y);   // solve Pa = y using linear least squares

    return a;
}
```

Figure 8: Code Example: polynomial data fitting.