RC 20011 (03/31/95)
Computer Science

IBM Research Report

Scientific and Engineering C++: Wrapping LAPACK in Objects

John J. Barton and Lee R. Nackman

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Research Division
Almaden · T.J. Watson · Tokyo · Zurich

C++ offers object-oriented tools for organizing software: we think in terms of objects, classes, and polymorphism. But some of the best scientific and engineering software is organized procedurally, in vast subroutine libraries. Fortunately, we don't need to give up on objects and we don't have to rewrite existing libraries in C++: we can organize existing libraries into object-oriented systems with class wrappers.
In one of our earlier columns [1], we implemented matrix multiplication for a Fortran-compatible matrix class using the Basic Linear Algebra Subroutines (BLAS). This was straightforward because there wasn't much structure to capture: we used overloading and templates to express variation in matrix element type, but that's it. We defined the objects and used the subroutine library to implement some of the member functions.
This column tackles a more difficult problem. We want to provide a C++ wrapper for a subroutine library that, if you look carefully, implicitly defines a rich set of objects with sophisticated interactions. The library is LAPACK (Linear Algebra PACKage) [2], a well-designed package in wide use and applicable to a variety of scientific and engineering problems. LAPACK consists of some 600,000 lines of Fortran-77 code, including routines for linear equations, linear least squares problems, eigenvalue and singular value problems, and generalized eigenvalue problems. There are over 250 routines for solving the linear equation

$$A\mathbf{x} = \mathbf{b},$$

where A is the coefficient matrix, $\mathbf{b}$ is the right-hand side, and $\mathbf{x}$ is the sought-for solution. We'll focus on this part of LAPACK. Chapters 15 and 18 of our book [3] describe our approach in more detail; [4] describes a different approach. A C++ class library based on LAPACK is also available commercially [5].

A Bit of Math. We'll start with an instant course on linear equations. Linear equations arise when two or more quantities are interdependent. Sometimes the dependency is naturally linear, but more often a complex interaction is "linearized", usually by chopping a complex problem into tiny pieces that are close enough that their interactions are linear. For example, linear equations arise when space or time in differential equations is approximated by a discrete mesh. Much creative effort in technical computing goes into formulating problems in terms of linear equations.
Once we have the equations, we can rely on vast experience in their efficient solution. Linear equations are solved by factoring the coefficient matrix into a product of "nicer" matrices and then solving the equations using the factored form. For example, LAPACK factors a general matrix A into the product of a permutation matrix P, a unit lower triangular matrix L (ones on the diagonal and zeros above the diagonal) and an upper triangular matrix U (zeros below the diagonal). The solution $\mathbf{x}$ is obtained by solving two successive triangular systems of equations:

$$LY = P^{-1}B \text{ and } U\mathbf{x} = Y.$$

Triangular systems are easy to solve: one equation is solved at the tip of the triangle and we substitute the solution into the next equation to solve it, and so on.

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \qquad \begin{bmatrix} \times & a & b & c \\ a & \times & d & e \\ b & d & \times & f \\ c & e & f & \times \end{bmatrix}$$

General Symmetric

$$\begin{bmatrix} \times & a & b & c \\ \overline{a} & \times & d & e \\ \overline{b} & \overline{d} & \times & f \\ \overline{c} & \overline{e} & \overline{f} & \times \end{bmatrix} \qquad \begin{bmatrix} \times & \times & 0 & 0 \\ \times & \times & \times & 0 \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix}$$

Hermitian Banded

$$\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \qquad \begin{bmatrix} \times & 0 & 0 & 0 \\ \times & \times & 0 & 0 \\ \times & \times & \times & 0 \\ \times & \times & \times & \times \end{bmatrix}$$

Upper Triangular Lower Triangular

$$\begin{bmatrix} \times & \times & 0 & 0 \\ \times & \times & \times & 0 \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}$$

Tridiagonal

Figure 1: Some LAPACK Matrix Structures. Elements that are not necessarily zero are indicated by $\times$ or by a letter. Elements with the same letter have the same value, and an overbar indicates complex conjugation.

Matrix Representations. In practical computations, we encounter huge coefficient matrices: compact representations are a must. Matrices come in different flavors, depending on their structure (patterns of zero and non-zero elements) and various special properties. Figure 1 illustrates some of the matrix structures utilized by LAPACK. As you can gather from Figure 1, a symmetric matrix can be stored more compactly than a general matrix of the same number of rows and columns. Likewise for matrices with other structures. LAPACK uses four basic representations [2, Section 5.3]. The conventional scheme stores a matrix in a two-dimensional Fortran array. The packed scheme stores symmetric, Hermitian, and triangular matrices packed by columns into a one-dimensional Fortran array. An $m \times n$ banded matrix is stored in a two-dimensional FORTRAN array smaller than $m \times n$. Unsymmetric bidiagonal and tridiagonal matrices are stored in two or three one-dimensional Fortran arrays.

In addition to more compact storage, a linear equation having a coefficient matrix with symmetry or an interesting pattern of zeros can be solved more efficiently than an equation with an arbitrary coefficient matrix. Moreover, a linear equation with a symmetric coefficient matrix can be solved even more efficiently if the matrix, $A = A^T$, is *positive definite*, that is, if $\mathbf{x}^T A\mathbf{x} > 0$ for all nonzero $\mathbf{x}$; a matrix not known to be positive definite is called *indefinite*. Positive definite symmetric matrices arise often in practice. For example, if the matrix A represents a kinetic energy operator then

X	Element Type
S	REAL
D	DOUBLE PRECISION
C	COMPLEX
Z	COMPLEX*16

ZZZ	Computation
TRF	Factor
TRS	Solve from factorization
CON	Estimate condition
RFS	Refine solution
TRI	Inverse from factorization
EQU	Equilibrate (scale)

YY	Matrix Structure
GB	General band
GE	General
GT	General tridiagonal
PO	Symmetric (Hermitian) positive definite
PP	Symmetric (Hermitian) positive definite packed
PB	Symmetric (Hermitian) positive definite banded
PT	Symmetric (Hermitian) positive definite tridiagonal
SY	Symmetric (complex) indefinite
SP	Symmetric (complex) indefinite packed
HE	Hermitian indefinite
HP	Hermitian indefinite packed
TR	Triangular
TP	Triangular packed
TB	Triangular banded

Table 1: LAPACK Subroutine Naming Convention. All subroutine names consist of six letters of the form XYYZZZ, where X indicates the type of the matrix elements, YY indicates the structure of the matrix, and ZZZ indicates the computation to be done by the subroutine.

$\mathbf{x}^T A\mathbf{x}$ is a kinetic energy that must always be positive.

LAPACK Naming Convention. Each LAPACK subroutine can be thought of as a point in a carefully organized five-dimensional design space, with the subroutine's coordinates encoded in its name. The dimensions of the design space are element type, operation, matrix structure, matrix properties, and matrix representation. We have already discussed matrix structure, properties, and representation. LAPACK deals with matrix element types real, double precision, complex, and double precision complex. Besides factoring and solving operations, LAPACK can also estimate the condition number of a matrix (roughly: a measure of how much errors in A or $\mathbf{b}$ affect the solution of the equation), refine a solution to reduce error, compute A^{-1}, and equilibrate (roughly: scale) a coefficient matrix. These elements all combine in the six character subroutine name. Each subroutine name consists of three parts: element type, operation, and matrix kind, with the latter a melding of matrix structure, properties, and representation. Table 1 shows the naming convention.

LAPACK and C++. So if LAPACK is designed so well, how can C++ be used to improve it? Three ways are to add support for objects, classes, and polymorphism.
Objects group the information about linear equations that are scattered across several variables in LAPACK. For example, LAPACK represents a matrix as a chunk of storage, one or more sizes, and sometimes auxiliary information like pivots; with C++ we can represent a matrix as a single object.
Classes introduce type checking based on the kind of matrix. For example, LAPACK cannot prevent you from attempting to factor a symmetric packed matrix with a general matrix factoring function. The C++ wrapper we will develop catches such errors at compile time.

Polymorphism let's us write code abstract from a concrete representation, code that gets reused because it does not depend on non-essential details. For example, LAPACK can factor and solve many kinds of matrices, but a new factoring and solving function is needed for each kind of matrix even if the operations on the matrix are logically identical.
We'd like our C++ code to look something like this:

```
template<class LapackMatrix, class EltT>
LapackRect<EltT>
xForAxEqb(LapackMatrix& matrixA, LapackRect<EltT>& b) {
    return matrixA.factor().solve(b);
}
```

so that for simple problems we can just call `xForAxEqb(A,b)` to solve linear equations, independent of the element type or matrix structure. Here you can see that all the information about the matrix must be captured in the object `matrixA`, that only the correct `solve()` function for `matrixA` will get called, and that many different types of linear equations can be solved with this function if the underlying classes provide the `factor()` and `solve()` member functions.

Finding Some Objects. Clearly we are on the trail to the objects in LAPACK: one class for each matrix type makes sense, with the six matrix operations being the member functions. This is a good start, but let's look more carefully. To keep things simple, we'll just look at two of the operations: factoring and solving, the `xyyTRF` and `xyyTRS` family of subroutines.
Consider the calling sequences shown in Table 2. Unpacked matrices are passed to the factoring subroutines as a triple or quadruple: The matrix in a Fortran array `A`, `A`'s leading dimension `LDA`, the number of columns `N`, and, for general matrices (which can be rectangular), the number of rows `M`. Upon return, `A` contains the factored result and `IPIV` contains the pivots (where used—this is the permutation matrix P mentioned in our little math course). The matrix and pivots go back into the solving subroutines along with a matrix of the right-hand sides, passed as the triple `B`, `LDB`, and `N`. The solution to the linear system is returned in `B`. Similar patterns, different in detail, appear in the calling sequences for other LAPACK subroutines.
The form of the factored matrix depends on the kind of matrix. For example, `SGETRF` uses single precision to factor a general matrix into the product of a permutation (represented by pivots), a unit lower triangular matrix, and an upper triangular matrix. `SPSTRF` factors positive definite symmetric matrices into the product $U^T U$, where U is upper triangular. There are similar variations for the other kinds of matrices.
Each factoring subroutine overwrites its input matrix with a factored form, but these factored forms are not interchangeable. Each must be passed to its corresponding solving subroutine: If `SGETRF` factorization is used, then `SGETRS` must be called for the solution, not `SSPTRS`.
So now we see that for every kind of matrix there are really two objects: an unfactored form and a factored form. The unfactored form behaves like a matrix and can be factored; the factored form is not a matrix in the mathematical sense (indeed, for the general case, the factored form requires pivots in its representation), but it is easy to solve.

Options	Problem Size	Matrix	Pivots	Work Space	Right Side	Status
SGETRF(	M,N,	A, LDA,	IPIV,			INFO)
SGETRS(TRANS,	N,NRHS,	A, LDA,	IPIV,		B,LDB,	INFO)
SPOTRF(UPLO,	N,	A, LDA,				INFO)
SPOTRS(UPLO,	N,NRHS,	A, LDA,			B,LDB,	INFO)
SPPTRF(UPLO,	N,	AP,				INFO)
SPPTRS(UPLO,	N,NRHS,	AP,			B,LDB,	INFO)
SSYTRF(UPLO,	N,	A, LDA,	IPIV,	WORK,LWORK,		INFO)
SSYTRS(UPLO,	N,NRHS,	A, LDA,	IPIV,	WORK,LWORK,	B,LDB,	INFO)
CHETRF(UPLO,	N,	A, LDA,	IPIV,	WORK,LWORK,		INFO)
CHETRS(UPLO,	N,NRHS,	A, LDA,	IPIV,	WORK,LWORK,	B,LDB,	INFO)
SSPTRF(UPLO,	N,	AP,	IPIV,			INFO)
SSPTRS(UPLO,	N,NRHS,	AP,	IPIV,		B,LDB,	INFO)
CHPTRF(UPLO,	N,	AP,	IPIV,			INFO)
CHPTRS(UPLO,	N,NRHS,	AP,	IPIV,		B,LDB,	INFO)

Table 2: Calling Sequences for Pairs of LAPACK Subroutines. One routine of each pair factors the matrix A (TRF-ending subroutines) and the other routine solves the resulting factored linear systems (TRS-ending subroutines). Subroutines for banded and tridiagonal matrices are omitted.

Classes. For each kind of matrix we need a factored form and an unfactored form, each available for multiple element types. We capture this with pairs of class templates, one class template for the factored and one for the unfactored form, each parameterized by element type. For element storage we'll use array class templates `ConcreteFortranArray2d<T>` and `ConcreteFortranSymmetricPackedArray2d<T>`. These class templates only provide two-dimensional array behavior, no math. The first stores element type T in the conventional Fortran storage layout and the second uses the packed LAPACK layout for symmetric matrices. We won't describe these classes here, but they act like 2D arrays [3].

Given these storage templates, the class templates for the unfactored form of general matrices and symmetric positive definite matrices might look like these:

```
template<class T>
class LapackRect :
    public ConcreteFortranArray2d<T> { // LAPACK A, M, & N
public:
    LapackRect(int nrows, int ncols);
    FactoredLapackRect<T> factor();
    // ...
};

template<class T>
class LapackSymPosDefPacked :
    public ConcreteFortranSymmetricPackedArray2d<T> {
public:
```

```
    LapackSymPosDefPacked(int size);
    FactoredLapackSymPosDefPacked<T> factor();
};
```

Both are derived from the appropriate array class template to give them array-like behavior and representation; the derived classes add the ability to factor.
Corresponding class templates can be defined for the factored forms:

```
template<class T>
class FactoredLapackRect {
public:
    LapackRect<T>& solve(LapackRect<T>&);
private:
    friend FactoredLapackRect<T> LapackRect<T>::factor();
    FactoredLapackRect(int nrows, int ncols);

    ConcreteFortranArray2d<T>   fmat;     // LAPACK "A"
    ConcreteFortranArray1d<int> pivots;   // LAPACK "IPIV"
};
```

Since the factored forms don't behave like arrays, they are not derived from the array class templates, but instead use them as members. The only sensible way to create an instance of this class is to factor a `LapackRect<T>`. Therefore we make the constructor private and give `LapackRect<T>`'s `factor()` access to it via the friend declaration.
The symmetric case is similar:

```
template<class T>
class FactoredLapackSymPosDefPacked {
public:
    LapackRect<T>& solve(LapackRect<T>&);
private:
    friend FactoredLapackSymPosDefPacked<T>
                LapackSymPosDefPacked<T>::factor();
    FactoredLapackSymPosDefPacked(int size);

    ConcreteFortranArray2d<T>   fmat;     // LAPACK "A"
};
```

In both cases, the `factor()` and `solve()` member functions can be written as template functions using the function overloading technique we illustrated with BLAS subroutines in an earlier column[1]. Depending on the type of `T`, the overloaded function called by, for example,`LapackRect<T>`'s `factor()` will resolve to either `STRFGE`, `DTRFGE`, `CTRFGE`, or `ZTRFGE`. These member functions may also need to provide temporary workspace and to handle error codes.
These classes let us write natural code for solving equations:

```
LapackRect<float> a1(10,10);
// ... fill a1 ...
FactoredLapackRect<float> factored_a1 = a1.factor();
LapackRect<float> b1(10, 1);
// ... fill b1 with right hand side  ...
factored_a1.solve(b1); // Solution is in b1

LapackSymPosDefPacked<float> a2(10);
```

```
// ... fill a2 ...
FactoredLapackSymPosDefPacked<float>
                           factored_a2 =  a2.factor();
LapackRect<float> b2(10, 1);
// ... fill b2 with right hand side  ...
factored_a2.solve(b2); // Solution is in b2
```

Alternatively, we can just call `xForAxEqb(a1, b1)` or `xForAxEqb(a2, b2)` since our LAPACK classes supply the function names needed for the expansion of this function template.

Summary. By following the pattern we have set up here, all of LAPACK can be organized into classes that support object-oriented programming. With one exception, the performance penalty of using this wrapper approach rather than using pure Fortran or C++ code will be insignificant. Mostly we have bundled several LAPACK variables together and abstracted the underlying LAPACK function to its conceptual name, using, for example, `factor()` in place of `SGETTF`, `SPOTRF`, `SSYTRF`, `CHETRF`, and so on.
The one exception is our failure to support in-place factoring of matrices. LAPACK supports factoring a matrix right back onto the storage it occupies. For many applications, even doubling the storage required for element representation is not a big cost these days. But true state-of-the-art computations always push the limits of our machines: they are going to need in-place factorization. We'll pick up on that problem in our next column.

References

[1] John J. Barton and Lee R. Nackman. Scientific and engineering C++: Grafting onto our roots. *C++ Report*, pages 33–37, September 1993.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1992.

[3] John J. Barton and Lee R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley, Reading, MA, 1994.

[4] Jack J. Dongarra, Roldan Pozo, and David W. Walker. Lapack++: A design overview of object-oriented extensions for high performance linear algebra. Available on the World-Wide Web at URL <http://www.netlib.org/c%2b%2b/lapack%2b%2b/>.

[5] Rogue Wave Software. Lapack.h++: A toolbox of c++ classes for numerical linear algebra. Available from Rogue Wave Software, P. O. Box 2328, Corvallis OR.