


    B. S. Goble Y. Ikhe, Y. C. Kim
    in Computer Science, Springer-Verlag, Berlin, 2
    ed. 1976
both sequential and distributed matrices as fundamental objects.

In both the parallel and sequential aspects of LAPACK++, decoupling the matrix algorithms from specific data decompositions provides three important attributes: (1) it results in simpler code which more closely matches the underlying mathematical formulation (2) it allows for one “universal” algorithm that supports one version for each data decomposition and (3) it allows one to postpone the data decomposition decision until runtime.

We used the inheritance mechanism of the object-oriented design to provide a common-source code for both the parallel and sequential version. Because the parallelism embedded in the sequential BLAS library, the sequential and parallel high-level matrix algorithms in SalPACK look the same. This tremendously simplifies the capacity of the parallel libraries.

We used polymorphism, dynamic binding, into these a truly portable matrix library in which the data decomposition may be deferred until runtime.

We utilized operator overloading capabilities of C++ to simplify the syntax and user interface of the LAPACK and BLAS.

We utilized the function overloading capabilities of C++ to redefine the function overloading usually associated with interfacing Fortran or assembly kernels.

In short, we have used various important aspects of object-oriented programming and C++ in the design of LAPACK++. These attributes were utilized in lieu of reality but out of necessity to incorporate a design that has scalability, portability, flexibility and ease-of-use.

References


A block scattered decomposition is a method of distributing the workload of a parallel algorithm across multiple processors. Each block of the matrix is partitioned and distributed over a 2D processor grid, where each processor is responsible for a portion of the data. This approach allows for efficient load balancing and minimizing communication overhead.

![Block Scattered Decomposition](image)

### Figure 6: Example of Block Scattered Decomposition over a 2D processor grid

The block scattered decomposition is a practical and efficient way to parallelize linear algebra computations. It allows for scalability by using a block structure, which can be represented as a matrix of blocks. This approach is particularly useful in applications where the data is naturally partitioned, such as in scientific simulations or large-scale data processing.

![Design Hierarchy of ScalAPACK++](image)

### Figure 7: Design Hierarchy of ScalAPACK++

At a higher level, the block algorithm relies on a Scalable Linear Algebra Package (ScaLAPACK++) library, which provides the foundation for parallel linear algebra computations. ScaLAPACK++ is designed to support scalable parallelism on distributed memory systems.

### 7 Conclusion

We presented a design overview for object-oriented linear algebra on high-performance architectures. We also described the benefits of using distributed memory architectures in parallel applications.
void poly_fit (LaVector<double> &x, 
   LaVector<double> &y, LaVector<double> &p) 
{
    int N = min (x.size (), y.size () );
    int d = p.size ();
    LaGenMatDouble P(N, d); 
    LaVectorDouble a(d);
    double x_to_the_j = 1;
    // construct Vandermonde matrix
    for (i=0; i<N; i++)
    {
        x_to_the_j = 1;
        for (j=0; j<d; j++)
        {
            P(i, j) = x_to_the_j;
            x_to_the_j *= x(i);
        }
    }
    // solve Pa = y using linear least squares
    LaLinSolveIP (P, p, y);
}

Figure 5 LAAK++ code example: polynomial


degree polynomial equation

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

using QR factorization. It is intended for illustrative purposes only. The implementation is not intended to solve this problem.

Given vectors \( x \) and \( y \) it returns the vector of coefficients \( a = \{ a_0, a_1, a_2, \ldots, a_d \} \). It is assumed \( N \gg d \). The solution arises from solving the overdetermined linear system \( Xa = y \):

\[
\begin{pmatrix}
1 & x_0 & x_0^2 & \ldots & x_0^d \\
1 & x_1 & x_1^2 & \ldots & x_1^d \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_N & x_N^2 & \ldots & x_N^d
\end{pmatrix}
\begin{pmatrix}
a_0 \\
a_1 \\
\vdots \\
a_d
\end{pmatrix}
= 
\begin{pmatrix}
y_0 \\
y_1 \\
\vdots \\
y_N
\end{pmatrix}
\]

Using least squares, i.e., minimizing \( || Xa - y ||^2 \).

The resulting code is shown in Figure 5.

6 ScaLAPACK++: an extension for distributed architectures

There are various ways to extend LAPACK++ libraries; we discuss one such extension ScaLAPACK++ [2], for linear algebra and distributed memory architectures. The intent is that for large scale problems ScaLAPACK++ should effectively exploit the computational issues of restructured parallel architecture, either on a few thousand processors, such as the Intel Paragon or Thinking Machines Corporations CM-5.

Splitting the data while maintaining parallelism, scalability, and performance are the key issues. The ScaLAPACK++ architecture is to provide a parallel matrix decomposition [8] to the library to operate on matrices.

Decomposing the matrix operation from details of the decomposition. In fact, splitting the matrix into a number of substructures, each of which can be allocated to a different processor. There are several strategies employed for a given matrix, such as the ability to automatically select the appropriate strategy. In many applications, the quadratic matrix decomposition is often used on linear matrices, utilizing the algorithm to intelligently allocate the matrix to a single processor. The ability to support various matrix decomposition strategies is one of the key features of ScaLAPACK++.

The recently proposed decompositions extend the ScaLAPACK++ features. These global matrix objects that are distributed across a \( P \times Q \) logical grid of processors. Matrix objects are interpreted as global objects, which is then distributed across a processor grid of \( P \times Q \) processors (Figure 6). These allow the use of a wide variety of matrix objects while ensuring scalability and maintaining good performance for various factor-
4 Driver Routines

This section describes LAMRoutines for solving linear systems of linear equations

\[ Ax = b, \]

where \( A \) is the coefficient matrix, \( b \) is the right-hand side, and \( x \) is the solution. \( A \) is assumed to be a square matrix of order \( n \), although non-square matrices allow \( A \) 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 shown in Figure 4.

To facilitate a linear equation driver in A

```latex
\text{LaLinSolve}( op(A), X, B); \]
```

where \( X \) and \( B \) are the input, and \( A \) is the output. \( A \) is an \( M \times N \) matrix of the above types. Setting \( nrhs \) defines the number of right-hand sides, i.e., \( A \times X \) and \( B \) are both rectangular matrices of size \( M \times nrhs \). The syntax \( op(A) \) can either use \( A \) or the transpose of \( A \), expressed as \( \text{trans}(A) \).

This version requires intermediate storage of approximately \( M \times N \times \text{nrhs} \) elements.

In cases where no additional information is supplied, the LAMRoutines will attempt to follow an intelligent choice of action. For example, if \( \text{LaLinSolve}(A, X, B) \) is called with a square \( M \times N \) matrix, the solution returned will be the linear least square that minimizes \[ \| Ax - b \|_2 \] using a QR factorization. If \( A \) is declared as \( \text{SD} \) then a Cholesky factorization will be used alternatively. One can directly specify the exact factorization method such as \( \text{LaUFactor}(F, A) \). In this case, if \( A \) is nonsquare, the factor returns only a partial factorization of the upper square portion of \( A \).

Error conditions in performing the \( \text{LaLinSolve}() \) operation can be retrieved via the \( \text{LaLinSolveInfo}() \) function which returns information that the last called \( \text{LaLinSolve}() \). Accurate data is useful for diagnosing problems. A value of \( -i \) denotes that the \( i \)th argument was specified to be inappropriate. A positive value of \( i \) denotes that the \( i \)th argument \( U(i, 1) \) of the factorization has been completed, but the factor is exactly singular, so the solution cannot be computed. In this case, the value returned by \( \text{LaLinSolve}() \) is a null (0) matrix.

4.1 Memory Optimizations: Factorizing in Place

When using large matrices that contain a significant portion of available memory, it may be beneficial to tolerate the requirement of storing intermediate factorization representations at the expense of destroying the contents of the input matrix \( A \). For such matrices, factorizations are required to temporary data structures roughly equal to the size of the original input matrix (for general dense matrices, one needs slightly more storage due to pivoting which is generally stored in additional bank). For example, the temporary memory requirement of a square \( N \times N \) dense matrix factorization is \( N^3 \) for \( N \) elements. Subsequent factorizations are performed with the \( \text{LaLinSolveIP}() \) routine

```latex
\text{LaLinSolveIP}(A, X, B); \]
```

where the contents of \( A \) are overwritten (with the respective factorization). These "inplace" functions are intended for advanced programs and are not recommended for general use. They assume program \( nrhs \)'s responsibility to ensure that the contents of \( A \) have been destroyed, lower and leaving the original matrix of problem be zipped on a routine with limited memory.

5 Programming Examples

This code example solves the linear least squares problem of fitting \( N \) data pairs \((x, y)\) with a 8th
for Lapack General Matrix. The type suffix can be float, double, fcomplex, or dcomplex. 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 structured algorithms.

3.1.1 Declaring

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

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

LaGenMatDouble A(200, 100) = 0.0; // 1
LaGenMatDouble B; // 2
LaGenMatDouble C = ref(A); // 3
LaGenMatDouble D(); // 4
LaGenMatFloat E(d, 2, 2); // 5
```

Line (1) declares A to be a rectangular 200 by 100 matrix, with all of its elements initialized to 0.0. Line (2) declares B to be an empty (uninitialized) matrix. Until B becomes initialized, any attempt to reference its elements will result in a runtime error. Line (3) declares C to share the same elements of A. Line (4) illustrates an equivalent way of specifying the column major form of a new object construction. Finally, line (5) demonstrates how one can initialize a 2x2 matrix with the data from a standard C++ vector. The values are initialized in column-major form so that the first column of E contains \{1.0, 2.0\} and the second column of E contains \{3.0, 4.0\}.

3.1.2 Submatrices

Blocked linear algebra algorithms utilize submatrices as their basic unit of computation. It is common for submatrix operations to be highly optimized. This is because 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. Ideally, one would like to use familiar colon notation of Fortran 90 or Matlab for expressing submatrices. However, this modification of the C++ syntax is not possible without redefining the language specifications. As a reasonable compromise, LAPACK++ denotes submatrices by specifying a subscripts through the LaIndex() function. For example, the 3x3 matrix in the upper left corner of A is denoted

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

This refers to \(i=0, 1, 2 \ j=0, 1, 2\), and is equivalent to the \((0:2, 0:2)\) colon notation used elsewhere. Submatrix expressions may be also be used in assignment for assignment, as in

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

which sets the 3x3 submatrix of A to zero. Following the Fortran 90 conventions, 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, e, i)` is equivalent to the index sequence

\[ s, s+i, s+2i, \ldots s+[\frac{e-s}{i}]i \]

The internal representation of an index is not converted to a full vector, but kept in its compact triplet form. The increment values may be negative and allow one to traverse a subscript range in the opposite direction, such as \((10, -1, -1)\) to denote the sequence \((10, 9, 8, \ldots, 1)\). Indices can be named and used in expressions as in the following submatrix assignments.

```c
LaGenMatDouble> A(10, 10), B, C; // 1
LaGenMatB(2, 9, 2), // 2
LaGenMatB(1, 3, 2); // 3
Bref(A(1, 1)); // 4
B(2, 3) = 3.1; // 5
C = B(LaIndex(2, 4, 2), 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. Line (5) uses B(2, 3) as the same memory address as A(5, 7), so that a change to B will also
are typically stored in column-order for compatibility with Fortran subroutines and libraries.

Various types of matrices are supported: banded, symmetric, Hermitian, packed, triangular, tridiagonal, bidiagonal, and non-symmetric. 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 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 Level 3 routine dgemm.

Figure 2: Performance of LAPACK++ LU factorization on the IBM RS/6000 Model 550 workstation using GNU g++ v. 2.3.1 and BLAS routines from the I BM ESSL library. The results are indistinguishable between the Fortran and C++ interfaces.

3.1 General Matrices

One of the fundamental matrix types in LAPACK++ is a general (nonsymmetric) rectangular matrix. The possible data element types of this matrix include single and double precision of real and complex numbers. The corresponding LAPACK++ names are given as

LaGenMat<ty
2.1 A simple code example

To illustrate how LAPACK++ simplifies the matrix operation interface, we present a small code fragment to solve linear systems. The examples are incomplete and meant to merely illustrate the interface style. The next few sections discuss the details of matrix and their operations.

Consider solving the linear system \( Ax = b \) using LU factorization in LAPACK++:

```cpp
#include <lapack++.h>
LaGenMatDouble A(N,N);
LaVectorDouble x(N), b(N);
// ...
LaLinSolve(A,x,b);
```

The first line includes the LAPACK++ object function declaration. The second line declares and initializes a square \( N \times N \) coefficient matrix, while the third line declares the right-hand-side and solution vectors.

Finally, the `LaLinSolve()` function in the last line calls the underlying LAPACK driver routine to perform general linear equations.

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

```cpp
#include <lapack++.h>
LaTridiagMatDouble A(N,N);
LaVectorDouble x(N), b(N);
// ...
LaLinSolve(A,x,b);
```

The only code modification is in the declaration of the coefficient matrix \( A \). In this case, `LaLinSolve()` calls the driver routine `DGTSV()` for tridiagonal linear systems.

`LaLinSolve()` function has been overloaded to support different matrix types and operations depending on the type of matrix and its operation.

Performance

The elegance of the LAPACK++ matrix class is evident, for example, in a simple matrix-matrix multiply: \( A \times B \) is referenced as `A(i,j)` by default. By default, matrix subscripts begin at zero, keeping runtime performance overhead compared to similar conventions in C++. However, they can be overridden using optimized Fortran. This is not a user-defined value. (Fortran programmers naturally prefer 1.) Internally, LAPACK++ matrices can be referenced, assigned, and used in mathematical expressions as naturally as if they were an integral part of C++. The LAPACK++ interface provides a consistent and intuitive way to manipulate matrices.
to the forTRAN library, some of the key routines such as the matrix factorizations, are actually implemented in C++ so that the general algorithmic philosophy of the LAPACK++ design is applied to derived matrix classes, such as memory matrix objects.

LAPACK++ provides speed and efficiency competitive with native Fortran codes (see Section 2.2) and allows for a more flexible and extensible interface on scalable architectures.

The motivation and design goals for LAPACK++ include:

- Maintaining competitive performance with Fortran 77.
- Providing a simple interface that hides implementation details of various matrix storage schemes and their corresponding factorization algorithms.
- Providing a universal interface and opens the Fortran 77 design for integration into user-defined data structures and third-party matrix packages.
- Replacing static work array limitations of Fortran with more flexible and type-safe dynamic data allocations schemes.
- Providing an efficient indexing scheme for matrix elements that has minimal overhead and can be optimized in most application code loops.
- Utilizing function and operator overloading in C++ to simplify and reduce the number of interface entry points to LAPACK.
- Providing the capability to access submatrices by reference, rather than by value, and perform factorizations "in place"—vital for implementing blocked algorithms efficiently.
- Providing more meaningful naming conventions for variables and functions (e.g., names no longer limited to six alphanumeric characters, and so on).

LAPACK++ also provides object-oriented interface to the Basic Linear Algebra Subprograms (BLAS) [[6 allowing programmers to utilize these optimized computational kernels in their own applications.

In this paper, we focus on matrix factorizations.
LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra

Jack J. Dongarra §†, Roldan Pozo‡, and David W. Walker§

§Oak Ridge National Laboratory
Mathematical Sciences Section
P. O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

‡University of Tennessee
Department of Computer Science
107 Ayres Hall
Knoxville, TN 37996-1301

Abstract

LAPACK is an object-oriented C++ extension of the LAPACK (Linear Algebra PACKage) library for solving the common problems of numerical linear algebra: linear systems, least squares, and eigenvalue problems on high-performance computer architectures. The advantages of an object-oriented approach include the ability to encapsulate various matrix representations, hide their implementation details, reduce the number of subroutines, simplify their calling sequences, and provide an extensible software framework that can incorporate future extensions of LAPACK such as ScalAPACK for distributed memory architectures. We present an overview of the object-oriented design of the matrix and decomposition classes in C++ and discuss its impact on efficiency, garbage, and performance.

1 Introduction

LAPACK++ is an object-oriented C++ extension to the Fortran LAPACK library for numerical linear algebra. This package includes state-of-the-art numerical algorithms for the most common linear algebra problems encountered in scientific engineering applications. It is based on the LI-NPACK[5] and EL-Spack[1] libraries for solving linear equations, linear least squares, and eigenvalue problems for dense and banded systems. The current LAPACK software consists of over 1,000 routines and 600,000 lines of Fortran 77 source code.

The numerical algorithms in LAPACK utilize block-matrix operations, such as matrix-multiply the innermost loops to achieve high performance on cached and hierarchical memory architectures. These operations, standardized as a set of subroutines of the Level 3 BLAS (Basic Linear Algebra Subprograms)[6], improve performance by increasing the granularity of the computations and keeping the most frequently accessed subregions of a matrix in the fastest level of memory. The result is that these block-matrix versions of the fundamental algorithms typically show performance improvements of a factor of three over non-blocked versions[1].

LAPACK++ provides a framework for describing general block matrix computations in C++. Without a proper design of fundamental matrix and factorization classes, the performance benefits of blocked code can be easily lost due to unnecessary data copying, inefficient access of submatrices, and excessive runtime overhead in the dynamic binding mechanisms of C++. LAPACK++, however, is not a general purpose ar

and package. There are no functions, for example, for generalized eigenvector operations on matrices, or routines that work well on high-dimensional arrays. There are several good public domain and commercial C++ packages for these problems as well. The classes in LAPACK++, however, can easily integrate with these other C++ matrix interfaces. These objects have been explicitly designed with block matrix algorithms and make extensive use of the level 3 BLAS. Furthermore, LAPACK++ is more than just a shell...