Draft

```
LaVector<double> poly_fit(LaVector<double> x, LaVector<double> y, int d)
{
    int N = min(x.size(), y.size());

    LaGenMat<double> P(N,d);
    LaVector<double> 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 = LaQRSolveInPlace(P, y);   // solve Pa = y using linear least squares

    return a;
}
```

Figure 9: Code Example: polynomial data fitting.

Draft

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.

Draft

7 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.

7.1 Polynomial data fitting

This code example solves the liner least squares problem of fitting N data points (x_i, y_i) to a dth 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 9.

Draft

6.2 Linear Least Squares

Note: Needs to be filled in.

```
LaGenMat<double> A, C;
LaOrthoMat<double> Q;
LaUpperTriangMat<double> R;
LaQRFact<double> QR;

QR = LaQRFactor(A);
QR = LaQRFactorNoPivot(A);
LaLQFactor(A);
LaQLFactor(A);
LaRQFactor(A);

Q = QR.Q();
R = QR.R();

C = QR.Q();     // convert to general matrix

(LaGenMat<double>) QR.Q();      // auto conversion
```

6.3 Eigenvalue Problems

Draft

It is much more efficient to compute the condition number from the factorization of, rather than the direct matrix. The `LaCond(A)` routine first factors `A`.

- **error bounds:** after solving $AX = B$, compute compute bounds on the error of X and refine the solution to reduce the backward error. (Requires the matrix factorization of A.)

```
LaGenMat<double> A, X, B, X2;
LaSPDMat<double> S;
LaGenFact<double> LU;
LaSPDFact<double> LL
double ferr, berr;

LaImproveSolution(X2, LU, A, X, B, &ferr, &berr);
LaImproveSolution(X2, LL, A, X, B, &ferr, &berr);
```

Returns the improved solution `X2` together with forward (`ferr`) and backward (`berr`) errors.

- **inverse:** given a triangular matrix or factorization, compute A^{-1} (not provided for band matrices, because the inverse does not in general preserve bandedness.)

```
LaGenMat<double> A, Ainv;
LaSPDMat<double> S, Sinv;

Ainv = LaInvert(A);
Sinv = LaInvertInPlace(S);      // S and Sinv point to same space.
```

- **equilibrate:** equilibrate A using row and column scalings to reduce its condition number (provided for General and SPD matrices only.) The auxilary `LaEquilibrateFactors(A)` function merely computes the scaling factors, but does not perform the scaling.

```
LaGenMat<double> A(M,N), Ae(N,Nrhs);
LaSPDMat<double> S;
LaEquilibration<double> E;
LaVector<double> r(M), c(N);
double rowcnd, colcnd;
double amax;

Ae = LaEquilibrate(A);
Ae = LaEquilibrateInPlace(A);

E  = LaEquilibrateFactors(A);
```

The factors returned by `LaEquilibrateFactors()`, represent the row and column scale factors `r` and `c`, respectively; `rowcnd` and `colcond` denote the largest ratio of smallest `r(i)` (`c(i)`)to the largest `r(i)` (`c(i)`). The parameter `amax` denotes the absolute value of the largest matrix element.

Draft

6 Computational Routines

6.1 Linear Equations

LAPACK++ contains computational routines to perform the following functions often used in solving linear systems:

- **factorize:** reduce matrix into its triangular factors using LU factorization (with pivoting) or Cholesky LL^T factorization for SPD matrices.

```
LaGenMat<double> A;
LaSPDMat<double> B;

LaGenFact<double> LU;
LaSPDFact<double> LL;

LaLUFactor(LU, A);
LaLUFactor(LU, B)
LaLUFactorInPlace(LU, A);

LaCholFactor(LL, S);
LaCholFactorInPlace(LL, S);
```

  The `FactorInPlace` versions reduce the amount of required memory by rewriting the factorization in the same memory space reserved by the matrix. (Note that in this case, all other aliases to that matrix are "nullified.")

- **triangular backsolve:** solve a triangular system $TX = B$ by either forward or backward substitution. The object T can be either any triangular matrix factorization or a a triangular matrix itself.

```
LaGenMat<double>  B, X;
LaUpperTriangMat<double> T;
LaGenFact<double> LU;
LaSPDFact<double> LL;

LaSolve(LU, X, B);
LaSolve(T, X, B);
LaSolve(LL, X, B);
```

- **condition number estimate:** estimate the condition number $\kappa(A) = ||A||\,||A^{-1}||$, using Higham's modification [?] or Hager's method [?]. For symmetric positive definite triangular matrices, however, its is computed directly with comparable efficiency.

```
LaBandedMat<double> A;
LaGenFact<double> F;
double c, normA;

c = LaCond(A);
c = LaCond(F);
c = LaCond(F, 1);
c = LaCond(F, normA);
```

Draft

LU Factorization Object

```
LaUnitLowerTriangularMat L
LaUpperTriangularMat U
LaVector<int> ipiv
LaInfo info
boolean transpose
```

Choleksy Factorization Object

```
LaSymmMat L
LaInfo info
boolean transpose
```

QR Factorization Object

```
LaOrthoMat Q
LaUpperTriangularMat R
LaGenMat<t> Tau
LaVector<int> ipiv
LaInfo info
boolean transpose
```

Draft

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. A successful LU factorization, for example, will return 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 meaningless without this information. Rather than store and manage these components seperately, we create a factorization class, as in

```
LaGenMat<double> A, B, X;
LaGenFact<double> F;

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

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

```
LaUnitLowerTriangMat<double> L;
LaUpperTriangMat<double> U;
LaGenMat<double> 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.

In such cases, one can utilize the `inPlace` versions of computational routines (see section 4.2):

```
LaGenMat<double> A;
LaGenFact<double> F;

LaFactorInPlace(F, A);
```

5.2 The Factorization Data Structures

Draft

Table 4: LAPACK++ Drivers for Linear Equations: Packed Matrices.

Symmetric	`LaSymmPackedMat<t> A(N,N);`
	`LaGenMat<t> B(N,nrhs), X(N,nrhs);`
	`LaSymmPackedFact<t> F;`
	
	`LaSolve(A, X, B);`
	`LaSolveInPlace(A, X, B);`

Symmetric Positive Definite	`LaUpperPackedSPDMat<t> A(N,N);`
	`LaLowerPackedSPDMat<t> A(N,N);`
	`LaGenMat<t> B(N, nrhs), X(N, nrhs);`
	`LaSPDFact<t> F;`
	
	`LaSolve(A, X, B);`
	`LaSolveInPlace(A, X, B);`

Table 5: LAPACK++ Drivers for Linear Least Squares Problems

QR or LQ factorization	`LaGenMat<t> A(N,N);`
	`LaGenMat<t> B(N,nrhs), X(N,nrhs);`
	`LaQRFact<t> F;`
	
	`LaQRSsolve(A, X, B);`
	`LaQRSsolve(F, X, B);`

complete orthogonal factorization	`LaGenMat<t> A(N,N);`
	`LaGenMat<t> B(N, nrhs), X(N, nrhs);`
	`LaOrthoFact<t> F;`
	
	`LaOrhtoSsolve(A, X, B);`
	`LaOrhtoSsolve(F, X, B);`

singular value decomposition	`LaGenMat<t> A(N,N);`
	`LaGenMat<t> B(N, nrhs), X(N, nrhs);`
	`LaSVDFact<t> F;`
	
	`LaSVDSolve(A, X, B);`
	`LaSVDSolve(F, X, B);`

Draft

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

Draft

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

Draft

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 figuire 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

```
LaSolve(op(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 $N \times nrhs$. The syntax *op*`(A)` can denote either `A` or the transpose of `A`, expressed as `transp(A)`.

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 `LaSolve(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 `LaSolve()` operations can be retrieved via the `LaSolveInfo()` function, which returns information about the last called `LaSolve()`. A zero value denotes a successful completion. A negative value of $-i$ denotes that the ith 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 `LaSolve()` 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 $N \times 1$ (see section refsec:comp-linear and **??** for details). Such memory-efficient factorizations are accomplished with the `LaSolveInPlace()` routine:

```
LaSolveInPlace(op(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 `LaSolve()`. In the line above, both `X` and `B` refer to the same memory locations.

4.3 Linear Least Squares

Draft

```
template <class Type>
class LaTridiagMat<Type> : La_Matrix<Type>
{
    private:
        int     N;              // an NxN  (only square matrices are allowed)
        Vector<Type> d;         // N diagonal elements
        Vector<Type> du;        // N-1 superdiagonal elements
        Vector<Type> dl;        // N-1 subdiagonal elements
    public:
        LaTridiagMat();
        LaTridiagMat(int n, int m);    // error if n != m
        LaTridiagMat(const LaTridiagMat& LaTridiagMat);
        LaTridiagMat(int n,Type *d, Type *d1, Type *d2);

        Type& operator(int i, int j);
};
```

Figure 8: LAPACK++ class declaration for tridiagonal matrices.

3.7.1 Operations with Tridiagonal Matrices

3.7.2 Implementation

The tridiagonal matrix is represented internally as an Nx3 matrix. The subdiagonal is stored in the first column from [0:N-2], the diagonal is stored in the second column from [0:N-1], and the superdiagonal is in the third [0:N-2]. An example class declaration of tridiagonal matrices in shown in figure 8.

Draft

```
LaUpperTriangBandedMat<double>  U(N, kl);
LaLowerTriangBandedMat<double>  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

```
LaBandedMat<double> B(N, kl, ku) = 0.0;
LaUpperTriangBandedMat<double> U;
LaLowerTriangBandedMat<double> 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

```
LaSymmBandedMat<double> S(N, kd);
LaHermBandedMat<complex> 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

```
LaTridiagMat<double> B(N,N);
LaTridiagMat<double> A;
```

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

```
LaTridiagMat<complex> 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.

Draft

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.

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.

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

```
LaBandedMat<double> B(N, kl, ku);
LaBandedMat<double> 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 ith diagonal of `B` is accessed as `B(i)`, and `B(-i)` is the ith subdiagonal ($i > 0$). Thus `B(0)` is the main diagonal, The result is a vector with (possible non-unit stride),

```
LaVector<double> d = B(-2);      // 2nd subdiagonal of B
```

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:

Draft

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.

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.

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

`LaSymmMat<type>`	Rectangular symmetric
`LaSPDMat<type>`	Rectangular SPD
`LaHermitianMat<type>`	Rectangular Hermitian

Internally, symmetric and Hermitian matrices may be stored in an `Upper` or `Lower` format as shown in figure 6. Thus, only half of the elements need to be set. This allows one to coerce triangular matrices into symmetric ones without data movement of copying. For example,

```
LaGenMat<double> A(N,N) = 0.0;
LaUpperTriangMat<double> T(A);
LaSymmMat<double> S;

T = 1.0             // Upper triangular half of A is now 1.0

S.ref((LaSymmMat) T);  // S is 1.0 *everywhere*, A is still 0.0 in lower
                       //      triangular half.
```

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

Draft

In a reasonable compromise to these syntax limitations, submatrices in LAPACK++ are denoted by specifying a subscript range through the `LaIndex()` [1] 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 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,

```
LaGenMat<double> A(10,10), B, C;                    // 1
LaIndex I(1,2,9),                                   // 2
LaIndex J(1,2,3);                                   // 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 needs to be filled in.

The triangular matrix types consist of

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

[1] Similar constructs exists in some general-purpose array packages, e.g. `Index()` in M++ and `slice()` in Math.h++, although they are typically limited to one-dimensional vectors only.

Draft

3.3 General Matrices

One of the fundamental matrix types in LAPACK++ is a nonsymmetric rectangular matrix. The possible data element types of this matrix include

$$\{General\} \left\{ \begin{array}{l} single\ precision \\ double\ precision \\ single\ precision\ complex \\ double\ precision\ complex \end{array} \right\}$$

The corresponding LAPACK++ names are given as

```
LaGenMat<type>
```

for Lapack General Matrix. The `type` parameter 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 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};

LaGenMat<double>    A(200,100) = 0.0;       // 1
LaGenMat<complex>   B;                      // 2
LaGenMat<double>    C.ref(A);               // 3
LaGenMat<double>    D(A);                   // 4
LaGenMat<float>     E(d, 2, 2)              // 5
```

Line (1) declares `A` to be a rectangular 200x100 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 run time error. Line (3) declares `C` to share the same elements of `A`. Line (4) illustrates an equivalent way of specifying this at the time 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 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.

Note: The following paragraph is a short discourse explaining our design decisions, i.e. why **didn't** *we implement that in such and such way. We may omit this from the final paper if it seems unnecessary.*

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 impossible without redefining the language specifications. Another possibility is to index each matrix with four, rather than two, integers in which each pair specifies the endpoints of a subscript range. This notation, unfortunately, also causes confusion: specifying a subvector using an index pair (e.g. `v(i,j)`) looks very much like the element reference of a matrix, and the submatrix expression `A(i,j,k,l)` can be easily mistaken for an element of a four-dimensional array.

Draft

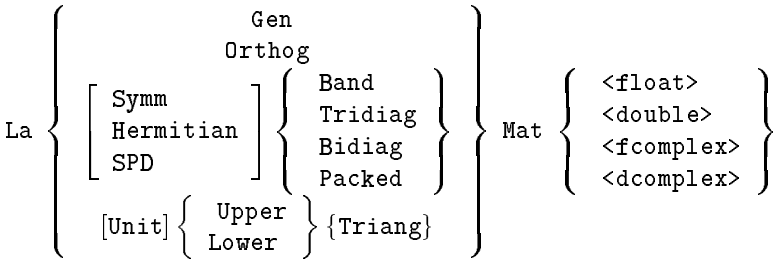

Figure 4: LAPACK++ matrix nomenclature. Items in square brackets are optional.

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

  ```
  Matrix A(M,N)
  ```

- **Indexing.** Individual elements of the matrix can be accessed by an integer pair (i, j) where $0 \leq i < M$ and $0 \leq j < N$. 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.

Draft

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.

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.

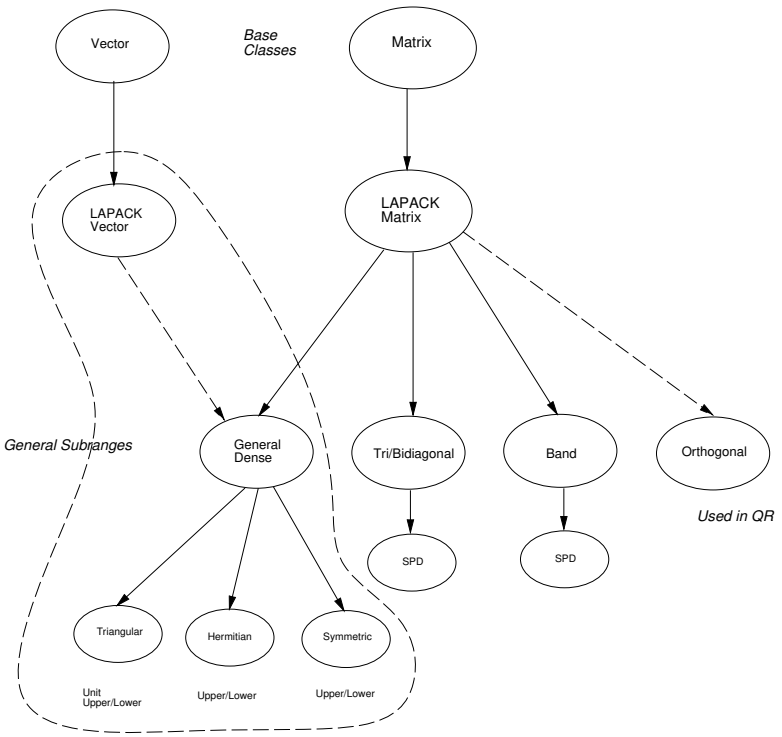

Figure 3: The Matrix Hierarchy of LAPACK++. Rectangular and Vector classes allow indexing of any rectangular submatrix by reference.

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.

Draft

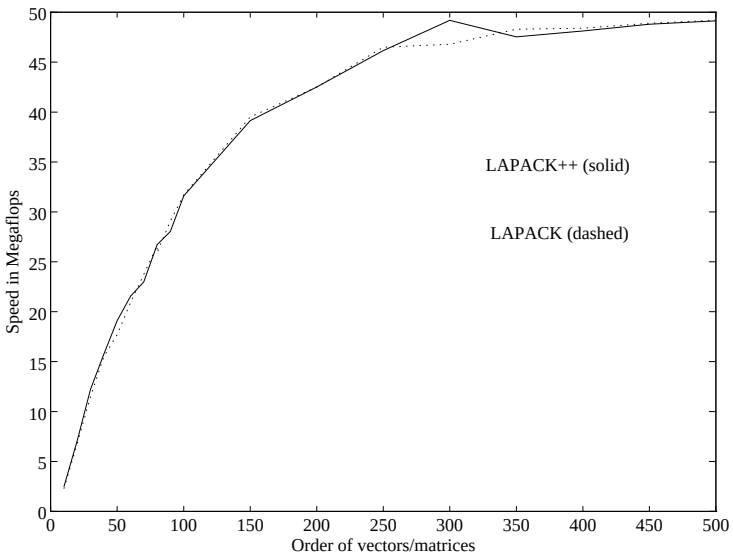

Figure 2: Performance of LAPACK++ LU factorization on the IBM RS/6000 Model 550 workstation, using GNU g++ v. 2.3.1 and BLAS routines from the IBM ESSL library. The results are nearly identical to the Fortran LAPACK performance.

Most LAPACK++ functions have a related helper function (usually the same name with `Info` suffixed) which returns information about the status of the last call made to that function. The encoded value (typically an integer) has the following semantics. A value of zero denotes a successful completion, a negative value of $-i$ denotes that the ith argument was somehow invalid or inappropriate, and a positive value typically denotes some error exception. See sections 4 and 6 for specific examples.

Draft

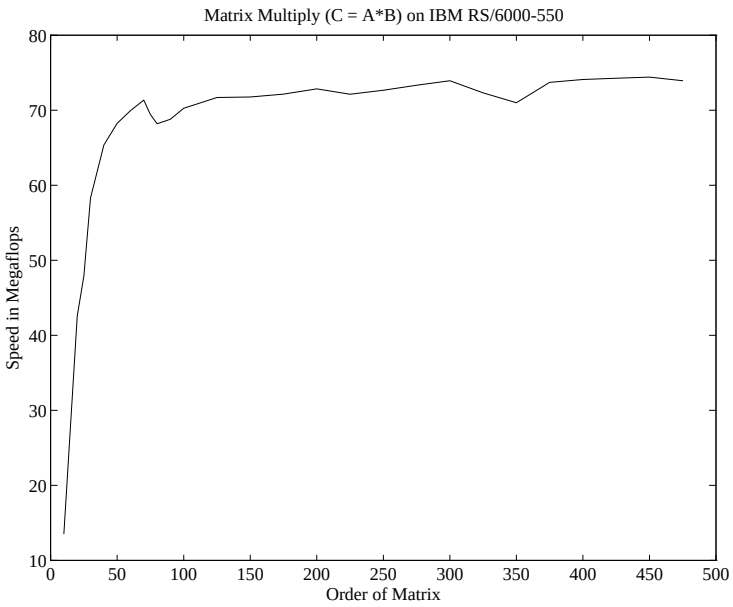

Figure 1: Performance of matrix multiply in LAPACK++ on the IBM RS/6000 Model 550 workstation. GNU g++ v. 2.3.1 was used together with the ESSL BLAS-3 routine `dgemm`.

routine, which overwrites `A` with its LU factors (see section 5 for details). This routine essentially inlines to the underlying LAPACK routine `DGETRF()` and incurs no runtime overhead.

2.4 The Cost of Polymorphism

When algorithms are expressed with specific data type and properly inlined, the C++ compiler can be essentially used an an intelligent preprocessor and avoid the usual function call overhead. When methods or functions are coded in terms of base classes, however, this is no longer possible, since the function binding is delayed until runtime [?]. In such cases, there is another overhead in virtual function table lookup to execute the corresponding code. Some experiments illustrate that on typical workstation architectures, a virtual function call is roughly three to five times as expensive as a direct call in typical situations.

2.5 Error Handling

Note: This section is still under development.

An important aspect of any numerical library is its handling of error conditions. This determines the flexibitliy of the package, its ease of use, and its integrability with other software packages. *Note: Some concerns are*

- *when are errors really* errors*?*
- *hard errors: out of memory*
- *soft errors: near-singular matrix*
- *exception handling in C++:* `catch()` *and* `throw()`

Draft

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 `LaSolve()` 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>

LaTridiagMat<double> A(N,N);
LaVector<double> x(N), b(N);
...

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

The only code modfication is in the declaration of A. In this case `LaSolve()` calls the driver routine SGTSV() for banded linear systems. The `LaSolve()` function been **overloaded** to perform different tasks depending on the type of the input matrix A. There is no runtime overhead associated with this; it is resolved by C++ at compile time.

2.3 Performance

The elegance of the LAPACK++ matrix classes may seem to imply that they incur a signifcant runtime performance overhead compared to similar computations using optimized Fortran. This section illustrates that this is not necessarily true. The design of LAPACK++ has been highly optimized for performance and can utilize the BLAS kernels as efficiently as Fortran. Figure 1, for example, illustrates the Megaflop rating of the simple code

```
LaMatMult(C, A, B); // 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;
    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. We 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.

Figure 2 illustrates performance characteristics of the LU factorization of various matrices on the same architecture using the LAPACK++

```
LaLUFactorInPlace(A,F)
```

Draft

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, `LaSolve()` and `LaEigen()`, replace several hundred subroutines in the original Fortran version.

2.1 Contents of LAPACK++

Note: This section needs to be filled in.

- 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

LaGenMat<double> A(N,N);                       // 2
LaVector<double> x(N), b(N);                   // 3
...

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

Draft

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 2.3), 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 **??**), 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.

Contents

1 Introduction **2**

2 Overview **3**
- 2.1 Contents of LAPACK++ . . . 3
- 2.2 A simple code example . . . 3
- 2.3 Performance . . . 4
- 2.4 The Cost of Polymorphism . . . 5
- 2.5 Error Handling . . . 5

3 LAPACK++ Matrix Objects **7**
- 3.1 Naming Conventions . . . 7
- 3.2 Fundamental Matrix Operations . . . 8
- 3.3 General Matrices . . . 9
  - 3.3.1 Declarations . . . 9
  - 3.3.2 Submatrices . . . 9
- 3.4 Triangular Matrices . . . 10
- 3.5 Symmetric, Hermitian and SPD Matrices . . . 11
- 3.6 Banded Matrices . . . 11
  - 3.6.1 Triangular Banded Matrices . . . 12
  - 3.6.2 Symmetric and Hermitian Banded Matrices . . . 13
- 3.7 Tridiagonal Matrices . . . 13
  - 3.7.1 Operations with Tridiagonal Matrices . . . 14
  - 3.7.2 Implementation . . . 14

4 Driver Routines **15**
- 4.1 Linear Equations . . . 15
- 4.2 Memory Optimizations: Factorizing in place . . . 15
- 4.3 Linear Least Squares . . . 15

5 Factorization Classes **19**
- 5.1 Optimizations: Factoring in Place . . . 19
- 5.2 The Factorization Data Structures . . . 19

6 Computational Routines **21**
- 6.1 Linear Equations . . . 21
- 6.2 Linear Least Squares . . . 23
- 6.3 Eigenvalue Problems . . . 23

7 Programming Examples **24**
- 7.1 Polynomial data fitting . . . 24

Draft

LAPACK++

A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra

March 16, 1993

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 via anonymous ftp from `cs.utk.edu:pub/pozo/papers/lapack++.ps.Z`. *Comments in italics such as this one are working notes and will not appear in the final manuscript.*

Draft