Numerical linear algebra software

Michael C. Grant

October 21, 2003

1 Introduction

Many of the iterative algorithms employed to solve optimization problems require the solution of a structured linear system at each iteration. For example, a single step of Newton's method, applied to a twice-differentiable convex function $f : \mathbb{R}^n \to \mathbb{R}$, requires the solution of the $n \times n$ symmetric positive definite linear system

$$H\Delta x = -g, \quad H \triangleq \nabla^2 f(x), \quad g \triangleq \nabla f(x)$$

at the current iterate $x \in \mathbb{R}^n$. In practice, we find that the bulk of computational resources (memory, time) consumed by these algorithms is spent constructing and solving these linear systems, and *not* in the higher-level algorithmic details. That is not to say that the higher-level details do not impact performance; for example, algorithmic improvements could result in a reduction in the *number* of iterations, yielding a proportional reduction in computation time. But the fact remains that the overall performance of many optimization algorithms depends heavily on the performance of the underlying matrix computations.

How do we know this? While theoretical complexity analysis can be used to provide some indication, such determinations are generally made through *profiling.* Profiling is the practice of running a program in a slightly modified manner so that its execution can be carefully monitored; for example, how often each subroutine is called, and how much time is spent in each one. In general, profiling reveals that the performance of a program is dominated by a handful of key subroutines. In the case of numerical optimization software, these critical subroutines are almost always those devoted to numerical linear algebra.

Therefore, anyone who wishes to become proficient in the construction of efficient optimization software must develop competence in the area of numerical linear algebra software as well. The most important step towards this competence is accepting this one principle:

Do not write your own.

In other words, you should *not* write C, C++, or Java code to perform seemingly simple tasks such as taking the inner product of two vectors, multipling two matrices together, or performing a Cholesky factorization—no matter how straightforward it might seem to do so. Instead, you should use one of the many existing, mature, publically available software libraries to perform these tasks. The reasons for this are severalfold:

- You can focus on other important issues, such as the algorithm and the user interface;
- The amount of code you will need to write will be greatly reduced...

- ...thereby greatly reducing the number of bugs that you will introduce;
- You will be insulated from subtle and difficult numerical accuracy issues; and
- Your code will run faster—sometimes dramatically so.

In this lecture, we will introduce you to to several software libraries for numerical linear algebra, and to some of the jargon and conventions that pervade the field. Nearly all of thse libraries can be found at one Web site, called Netlib [DGB$^+$03].[1] Netlib is a clearinghouse for some of the most (deservedly) popular numerical software available, particularly for linear algebra, and is maintained by people at the University of Tennessee and the Oak Ridge National Laboratory and colleagues around the world. Of particular interest is a web page created by one of Netlib's maintainers [Don03], which provides quite a long list of freely available linear algebra software.

Admittedly, for many people in this class it will be appropriate and reasonable to use nothing but MATLAB in their projects—in which case this lecture may not prove directly relevant. Nevertheless, what we present here will help you understand exactly what MATLAB is doing under the hood. Indeed, MATLAB uses several of the very libraries we are going to discuss here. And in your future work in business or academia, you may find some very practical reasons to move away from any dependence on MATLAB (cost, licensing restrictions, *etc.*), in which case you will be glad to have become familiar with these resources now.

Let us briefly address what some of you might otherwise consider a conspicuous omission from this document: the book *Numerical Recipes in C* [PFTV93], or related books for Fortran, C++, and Pascal. These books contain implementations of a number of useful algorithms for numerical linear algebra, and the code for these algorithms can be downloaded rather readily from the Web. Without going into detail, it is my opinion that these books have more *pedagogical* value than *practical* value. In other words, if you wish to learn how a particular algorithm works, these books can be quite helpful; but if you wish to *use* an algorithm, I would recommend that you refer instead to the resources described here.

It is important to be aware of any usage restrictions placed on the software you use, *particularly* when incoporating someone else's code into your own programs. Fortunately, there are plenty of numerical linear algebra software packages that may be freely used in any application, both educational and commercial. All of the packages listed in this document fit this criteria. Some are fully in the public domain, while others have modest attribution or documentation requirements—not unlike citing references when publishing a paper. If you use any of them, be sure to consult their documentation for specific licensing requirements.

The notes that follow assumes that the reader is familiar with the material in Appendix C of the *Convex Optimization* textbook [BV].

2 The Basic Linear Algebra Subroutines (BLAS)

The *Basic Linear Algebra Subroutines*, or BLAS, are a suite of "kernel" routines that implement a variety of fundamental "forward" linear algebra operations such as dot products, matrix-vector multiplcations, and matrix-matrix multiplications. The authors of the BLAS recognized how common these operations are in computational software, and the significant benefits that would result by creating a standardized library to implement them.

For full documentation on the BLAS library, consult the BLAS web site [BLA03].

[1] The URL's of the Web sites discussed here are found in the bibliography citations at the end of this document.

2.1 The levels of BLAS

The BLAS library as we know it today was specified and developed in three stages, or *levels*: Level 1 in 1973-1977 [LHKK79], Level 2 in 1984-1986 [DCHH88], and Level 3 in 1987-1990 [DCDH90]. Each of these levels roughly corresponds to a particular level of theoretical complexity:

- *Level 1*: $O(n)$ vector operations, such as addition, scaling, dot products, norms:

$$y \leftarrow \alpha x + y, \quad r \leftarrow x^T y, \quad r \leftarrow \|x\|_2, \quad r \leftarrow \|x\|_1$$

- *Level 2*: $O(n^2)$ matrix-vector operations, such as matrix-vector multiplications,

$$y \leftarrow \alpha A x + \beta y, \quad y \leftarrow \alpha A^T x + \beta y, \quad y \leftarrow \alpha A x + \beta B x$$

  triangular matrix-vector multiplications and matrix solves,

$$x \leftarrow \alpha T x, \quad x \leftarrow \alpha T^T x, \quad x \leftarrow \alpha T^{-1} x, \quad x \leftarrow \alpha T^{-T} x$$

  and rank-1 and symmetric rank-2 updates:

$$A \leftarrow \alpha x y^T + \beta A, \quad A \leftarrow \alpha x x^T + \beta A, \quad A \leftarrow \alpha x y^T + \alpha y x^T + \beta A$$

- *Level 3*: $O(n^3)$ matrix-matrix operations, such as matrix-matrix multiplication,

$$C \leftarrow \alpha AB + \beta C, \quad C \leftarrow \alpha AB^T + \beta C, \quad C \leftarrow \alpha A^T B + \beta C, \quad C \leftarrow \alpha A^T B^T + \beta C$$

  triangular matrix-matrix multiplications and matrix solves,

$$\begin{gathered} B \leftarrow \alpha T B, \quad B \leftarrow \alpha B T, \quad B \leftarrow \alpha T^T B, \quad B \leftarrow \alpha B T^T \\ B \leftarrow \alpha T^{-1} B, \quad B \leftarrow \alpha B T^{-1}, \quad B \leftarrow \alpha T^{-T} B, \quad B \leftarrow \alpha B T^{-T} \end{gathered}$$

  and symmetric rank-k and rank-$2k$ updates:

$$\begin{gathered} C \leftarrow \alpha A A^T + \beta C, \quad C \leftarrow \alpha A^T A + \beta C, \quad C \leftarrow \alpha A J A^T + \beta C, \quad C \leftarrow \alpha A^T J A + \beta C \\ C \leftarrow \alpha A B^T + \alpha B A^T + \beta C, \quad C \leftarrow \alpha A^T B + \alpha B^T A + \beta C \\ C \leftarrow \alpha A J B^T + \alpha B J A^T + \beta C, \quad C \leftarrow \alpha A^T J B + \alpha B^T J A + \beta C \end{gathered}$$

2.2 The BLAS naming convention

The BLAS was originally written in Fortran 66 and Fortran 77; and while a complete C "wrapper" interface has been constructed for it, vestiges of this Fortran history do remain.[2] The most prominent example is the consistent, compact naming convention that BLAS routines follow.

For the Level 1 BLAS, the naming convention is as follows:

`cblas_`	`X`	`XXXX`
prefix (C only)	data type	operation

[2]Unfortunately, not every Fortran library worth using has a C wrapper. So it is worthwile to learn how to call Fortran subroutines *directly* from C/C++ code; see §6 for more details. You may even decide to skip the C interfaces altogether once you become proficient at using Fortran libraries.

The `cblas_` prefix applies only to the C interface; a prefix is not used in the original Fortran version. Following this prefix is a single-letter code indicating the *data type* of the operation:

`s`	single precision real	`c`	single precision complex
`d`	double precision real	`z`	double precision complex

For our purposes, only `d` and occasionally `z` will be relevant. For vector functions, the data type is followed by a 3-5 letter operation code; *e.g.*,

`axpy`	scale and accumulate: $y \leftarrow \alpha x + y$
`dot`	dot (inner) product: $r \leftarrow x^T y$
`nrm2`	2-norm: $r \leftarrow \|x\|_2 = (\sum_{i=1}^n x_i^2)^{1/2}$

So, for example, the function `cblas_ddot` returns the inner product of two double-precision vectors.

For matrix operations in the Level 2/3 BLAS, the naming convention is a bit more complex:

`cblas_`	`X`	`XX`	`XXX`
prefix (C only)	data type	structure	operation

The prefix and data type are identical to the Level 1 case. Following the data type is a two-letter code indicating the *structure*, if any, of the matrix involved:

`tr`	triangular	`tp`	packed triangular	`tb`	banded triangular
`sy`	symmetric	`sp`	packed symmetric	`sb`	banded symmetric
`hy`	Hermitian	`hp`	packed Hermitian	`hn`	banded Hermitian
`ge`	general			`gb`	banded general

The "packed" and "banded" matrix types utilize a special, more efficient data format; for example, the symmetric packed `sp` format reduces the storage requirements from n^2 to $n(n+1)/2$ by eliminating the duplicated elements. Following this indicator is 1-3 letter operation code; *e.g.*,

`mv`	matrix-vector multiplication: $y \leftarrow \alpha Ax + y$
`mm`	matrix-matrix multiplication: $C \leftarrow \alpha AB + \beta C$
`r`	rank-one update: $A \leftarrow \alpha xy^T + A$

So, for example, the function `cblas_dsymv` multiplies a vector by a symmetric matrix, each stored in double precision format.

Once you gain some familiarity with these naming conventions, it quickly becomes very easy to remember. In fact, many of us who have spent perhaps too much time working with BLAS often use words like `ddot` or `daxpy` in conversation.

2.3 Using BLAS effectively

In order to use the BLAS most effectively, it is important to arrange your calculations in a way that maximizes the use of the higher-level BLAS operations, particularly the Level 3 operations. This is because the Level 3 operations exert finer control over the memory accesses than an equivalent set of Level 2 operations; and it is these memory access that generally dictate the performance of the algorithm. So, for example, consider the task of performing k rank-1 updates on a matrix:

$$A \leftarrow A + \sum_{i=1}^{k} x_i y_i^T, \qquad A = \begin{bmatrix} a_1 & \dots & a_n \end{bmatrix} \in \mathbb{R}^{m \times n},\ x_i \in \mathbb{R}^m,\ y_i \in \mathbb{R}^n$$

This operation can be performed using operations in any of the 3 BLAS levels:

- kn calls to the Level 1 routine `cblas_daxpy`: k for each column of A:

$$a_j \leftarrow a_j + y_{ji} x_i, \quad i = 1, \ldots, k, \; j = 1, \ldots, n$$

- k calls to `cblas_dgerk`, a Level 2 routine that performs rank-1 updates:

$$A \leftarrow A + x_i y_i^T, \quad i = 1, \ldots, k$$

- a single call to `cblas_dgemm`, a Level 3 routine that performs various matrix multiplications:

$$A \leftarrow A + XY^T, \quad X \triangleq \begin{bmatrix} x_1 & \ldots & x_k \end{bmatrix}, \quad Y \triangleq \begin{bmatrix} y_1 & \ldots & y_k \end{bmatrix}$$

This last choice will certainly give the best performance, because the potential for speed optimization is greater with Level 3 routines than with Level 2 routines, and in turn greater with Level 2 routines than Level 1.

On the other hand, it also has the strictest data storage requirements: the vectors x_i and y_i must be stored as the columns of respective matrices. Indeed, Level 3 routines do in general tend to impose stricter requirements on the organization of the input data. This does *not* mean that you should insert a lot of new data shuffling code into a routine simply to make it possible to call higher-level BLAS routines. Instead, you should *plan* your code design in such a manner that the data is *already* arranged in this format.

This provides only an overview of the functionality in the BLAS; for more information, consult the many resources at the BLAS home page [BLA03], including the BLAS Technical Forum document [Bla01].

2.4 Optimized BLAS and ATLAS

Many of the operations implemented in the BLAS seem rather simple, begging the question: why is it worth downloading, learning, and using the BLAS when implementing the same routines by hand is so simple? For example, consider the accumulated matrix multiplication utilized in the above example:

$$A \leftarrow A + XY^T, \quad X \in \mathbb{R}^{m \times p}, \; Y \in \mathbb{R}^{n \times p}$$

The individual elements of A are updated as follows,

$$A_{ij} \leftarrow A_{ij} + \sum_{k=1}^{p} X_{ik} Y_{jk}, \quad 1 \leq i \leq m, \; 1 \leq j \leq n$$

suggesting the following simple implementation:

```
void matmultadd( unsigned m, unsigned n, unsigned p,
                 const double* X, const double* Y, double *A ) {
    unsigned i, j, k;
    for ( i = 0 ; i < m ; ++i )
        for ( j = 0 ; j < n ; ++j )
            for ( k = 0 ; k < p ; ++k )
                A[ i + j * n ] += X[ i + k * p ] * Y[ j + k * p ];
}
```

What advantage does a call to the BLAS routine `cblas_dgemm` offer over this simple implementation—particularly when one considers the costs of using BLAS, such as the time needed to find, download, compile, and learn to use it? The answer to this question is *performance*: in this case, for example, the BLAS routine will likely be *several times faster* than this simple implementation!

Of course, there are a few simple things that one can do to the above function to improve its performance; for example, the array-indexing arithmetic can be greatly simplified. But incremental changes such as these will not be sufficient to provide the best performance. Thanks to memory bandwidth, cache architecture, and pipelining issues, it is *not trivial* to write linear algebra code that achieves high performance on modern computers. Indeed, it is rarely possible to write numerical linear algebra code that is simultaneously is easy to read and achieves high performance.

Fortunately, the acceptance of BLAS as a standard interface for numerical linear algebra has made practical a number of efforts to produce highly-optimized BLAS libraries. Many high-end workstation and processor vendors actually supply or sell BLAS for their systems (*e.g.*, [Int03]). A highly recommended, free alternative is ATLAS [WPD03, WPD00], which uses automated code generation and testing methods to *generate* an optimized BLAS for a given computer. The system is quite effective, actually: it achieves performance comparable to (and sometimes, better than) vendor-supplied BLAS libraries; it is very portable; and it has a permissive BSD-like license.

The primary technique that these optimized BLAS libraries employ to improve performance, particularly for Level 3 routines, is *blocking*. Blocking uses block matrix arithmetic to decompose a larger matrix calculation into a sequence of calculations on smaller submatrices. For example, when performing the calculation $A \leftarrow A + XY^T$ for large A, X, and Y, the subroutine `cblas_dgemm` may first break the matrices into blocks; *e.g.*,

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \leftarrow \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} + \begin{bmatrix} X_{11} \\ X_{21} \end{bmatrix} \begin{bmatrix} Y_{11}^T & Y_{21}^T \end{bmatrix} = \begin{bmatrix} A_{11} + X_{11}Y_{11}^T & A_{12} + X_{11}Y_{21}^T \\ A_{21} + X_{21}Y_{11}^T & A_{22} + X_{21}Y_{21}^T \end{bmatrix}$$

and then apply a standard, "unblocked" algorithm on these smaller blocks; *e.g.*,

$$A_{11} \leftarrow A_{11} + X_{11}Y_{11}^T, \quad A_{12} \leftarrow A_{12} + X_{11}Y_{21}^T, \quad A_{21} \leftarrow A_{21} + X_{21}Y_{11}^T, \quad A_{22} \leftarrow A_{22} + X_{21}Y_{21}^T.$$

Proper blocking makes efficient the use of the cache, so that the processor can spend most of its time performing calculations and relatively little waiting for data to arrive from the slower main memory. Block sizes are chosen to be just small enough so that each individual step of the calcuation fits within the cache; and the ordering of the calculations is carefully chosen to minimize redundant reading and writing of data to and from main memory. The best choices depend on the specific operation being performed, the sizes of the inputs, and specific architectural properties of the computer. In fact, the ATLAS library basically performs a series of tests over a wide range of block sizes and orderings to find the optimal combination.

What is the difference in performance? Admittedly, for Level 1 BLAS, speedups through careful hand-optimization are not particularly significant: say, 15 percent or so. But the improvement increases dramatically as you move to Level 2, and then to Level 3. For Level 3 BLAS, the difference can be *amazing*, with speedups speedups of *10 times* or more on many platforms, compared to a basic, "clean" implementation of the BLAS [WPD00]. Furthermore, the optimized routines come much closer to the known peak calculation rates supported by the underlying processor—in other words, they come close to known upper bounds on performance.

Therefore, due to the wide availability of high-performance BLAS libraries, both free and non-free—and due to the significant gains in performance that can be achieved by effectively using them—there is simply *no reason* not to use BLAS when writing code in C, C++, Fortran, *etc.*

2.5 The Matrix Template Library

Having just sung the praises of the BLAS, allow me to nonetheless suggest an alternative: The Matrix Template Library, or MTL [LSL01]. MTL is a numerical algebra library written specifically for C++, taking advantage of the most advanced features of the C++ language to achieve both flexibility and performance. It supports the same types of matrix structure as BLAS, but adds support for sparse matrices. We have no experience with this software; but if you are already quite familiar with the C++ language, you may want to consider this library as an alternative to BLAS.

3 LAPACK

The *Linear Algebra PACKage*, or LAPACK [LAP00, ABB+99], implements a variety of more advanced linear algebra computations designed to solve a variety of types of linear systems and perform a number of common matrix decompositions and factorizations. LAPACK was first released in February 1992, and the latest version was released in May 2000. It replaces predecessors EISPACK and LINPACK, providing more functionality, better accuracy, and better performance. LAPACK is built using the BLAS routines, which means that its performance depends heavily on the performance of the BLAS library being used. It uses a similar naming convention for its routines, and supports the same data types. Many of these routines support the many of the same types of structured matrices in the BLAS—including symmetric, triangular, and banded matrices.

For more complete documentation on the usage of LAPACK, consult the LAPACK web site [LAP00]. The full user guide [ABB+99] is available on-line at that site, although for a frequent user it is well worth purchasing a paper copy of the text. The Matrix Template Library [LSL01] provides an interface to LAPACK as well.

3.1 LAPACK routine categories

The routines in the LAPACK library are divided into three categories: *auxiliary*, *computational*, and *driver* routines. Auxiliary routines perform a number of miscellaneous low-level tasks, and are primarily intended to support the other routines. Computational routines are designed to perform single, specific computational tasks:

- factorizations: LU, LL^T/LL^H, LDL^T/LDL^H, QR, LQ, QRZ, generalized QR and RQ;
- eigenvalue decompositions for symmetric and nonsymmetric matrices;
- singular value decompositions; and
- generalized eigenvalue and singular value decompositions.

Finally, driver routines combine computational routines in sequence in order to solve a variety of standard linear algebra problems from start to finish, including:

- Linear equations: $AX = B$;
- Linear least squares problems:

$$\text{minimize}_x \quad \|b - Ax\|_2 \qquad\qquad \begin{array}{rl} \text{minimize} & \|y\| \\ \text{subject to} & d = By \end{array}$$

- Generalized linear least squares problems:

$$\begin{array}{ll} \text{minimize}_x & \|c - Ax\|_2 \\ \text{subject to} & Bx = d \end{array} \qquad \begin{array}{ll} \text{minimize} & \|y\| \\ \text{subject to} & d = Ax + By \end{array}$$

- Standard and generalized eignenvalue and singular value problems:

$$AZ = \Lambda Z, \quad A = U \Sigma V^T, \quad Z = \Lambda B Z$$

Driver routines are certainly the best choice when they fit an application. However, accessing the computational routines individually provides additional flexibility, particularly when solving non-standard problems or when access to intermediate results is required.

3.2 A usage example

To see how LAPACK can be applied in a variety of ways, let us the task of solving

$$\begin{bmatrix} P & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} d_x \\ d_v \end{bmatrix} = \begin{bmatrix} r_a \\ r_b \end{bmatrix}$$

where the positive definite matrix $P = P^T \in \mathbb{R}^{n \times n}$, the nonsingular matrix $A \in \mathbb{R}^{n \times m}$ $(m < n)$, and the vectors $r_a \in \mathbb{R}^n$, $r_b \in \mathbb{R}^m$ are all supplied separately. Linear systems with this type of structure occur quite often in KKT-based optimization algorithms.

First, we note that the $m + n \times m + n$ coefficient matrix of this linear system is symmetric indefinite and nonsingular. Therefore, it admits a permuted LDL^T factorization,

$$\begin{bmatrix} P & A^T \\ A & 0 \end{bmatrix} \rightarrow QLDL^T Q^T,$$

where Q is a permutation matrix, L is lower triangular, and D is block-diagonal with 1×1 and 2×2 blocks. Given this factorization, the result can be computed as

$$\begin{bmatrix} d_x \\ d_v \end{bmatrix} = QL^{-T} D^{-1} L^{-T} Q^T \begin{bmatrix} r_a \\ r_b \end{bmatrix}.$$

The LAPACK driver routine `dsysv` solves symmetric indefinite linear systems, by calling the computational routine `dsytrf` for computing the above factorization. So after constructing the coefficient matrix from P and A and the right-hand vector from r_a and r_b, the system can be solved with a single call to `dsysv`.

Let us consider an alternate, two-stage approach to this problem, suggested by the fact that

$$d_v = (AP^{-1}A^T)^{-1}(AP^{-1}r_a - r_b), \quad d_x = P^{-1}r_a - P^{-1}A^T d_v.$$

First, let us solve the system

$$P \begin{bmatrix} \tilde{A} & \tilde{r}_a \end{bmatrix} = \begin{bmatrix} A & r_a \end{bmatrix}$$

for the quantities $\tilde{A}$ and $\tilde{r}_a$. Because P is symmetric positive definite, we can use the routine `dspsv` to solve this problem, which will in turn call the computational routine `dsptrf` to compute a Cholesky factorization of P. Like most of LAPACK's driver routines, `dspsv` can efficiently handle multiple right-hand sides simultaneously, eliminating the need to factorize P more than once. Once these quantities are found, we then solve a second linear system

$$(\tilde{A}A^T)d_v = A\tilde{r}_a - r_b$$

for d_v. The matrix $\tilde{A}A^T$ can be computed using a relevant call to BLAS, either `dgemm` or `dsyr2k`; and once it has been computed, a second call to `dspsv` can be used to solve this second linear system. Finally, the quantity d_x is obtained by computing $d_x = \tilde{r}_a - \tilde{A}d_v$ with a single `daxpy` call.

There is a third way to solve this linear system using standard LAPACK calls, by first transforming the problem into a standard size $n \times n$ least squares problem. We leave the transformation itself as an exercise to the reader, except to say that it will require a single call to the computational routine `dsptrf`. The resulting least squares problem can be solved using a call to the driver routine `dgels`, which in turn calls the computational routine `dgeqrf` to perform a QR factorization on the coefficient matrix.

It may seem curious why we would even consider something other than an LDL^T factorization, when it seems so straightforward. In fact, there are a number of reasons. The P matrix may exhibit a significant amount of structure; say, for example, it is block-diagonal. Or perhaps the Cholesky factorization or inverse of P is readily available. In these cases, the Cholesky-based method is likely to be preferred, because it will be easier to exploit this additional information about P. Furthermore, a Cholesky-bsaed approach will often be preferred when A and P are *sparse*, for certain technical reasons we will discuss later.

4 Sparse matrices

We say that a matrix $A \in \mathbb{R}^{m\times n}$ is *sparse* if it satisfies two conditions:

- the number of non-zero elements n_{nz} is small; *i.e.*, $n_{\text{nz}} \ll mn$. For the purposes of this definition, any element which is not *known* to be zero is counted as non-zero.
- the matrix has a known *sparsity pattern*: that is, the *location* of each of the aforementioned non-zero elements is explicitly known. A sparsity pattern can be represented visually using a *spy diagram*; see Figure 1 for an example.

Sparsity can be exploited to increase the performance of many matrix computations; *e.g.*:

- Storing a matrix $A \in \mathbb{R}^{m\times n}$ using double precision numbers requires $8mn$ bytes if A is dense, or $16n_{\text{nz}}$ bytes (or less, depending on the exact storage format) if A is sparse.
- The operation $y \leftarrow y + Ax$ requires mn multiplications and additions if A is dense, but only n_{nz} multiplications and additions if A is sparse.
- The operation $x \leftarrow T^{-1}x$, where $T \in \mathbb{R}^{n\times n}$ is triangular and nonsingular, requires n divides and $n(n-1)/2$ multiplications and additions if A is dense; and n divides and $n_{\text{nz}} - n$ multiplications and additions if A is sparse.

There are a variety of methods for efficiently representing a sparse matrix on a computer. A simple (although somewhat inefficient) method is to store the data in a $3 \times n_{\text{nz}}$ matrix: the first row representing the row indices, the second row representing the column indices, and the third

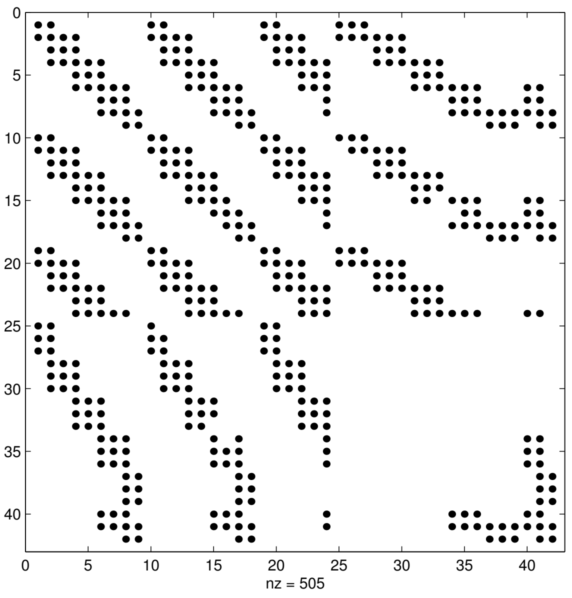

Figure 1: Spy diagram of a matrix. The dots represent non-zero entries.

row representing the values themselves. The columns are typically sorted in some fashion; and for symmetric matrices, redundant elements are eliminated. For example,

$$\begin{bmatrix} 2.5 & 1.2 & 0 & 0 \\ 1.2 & 3.0 & 0 & 0.8 \\ 0 & 0 & 4.0 & 0 \\ 0 & 0.8 & 0 & 2.5 \end{bmatrix} \implies \begin{bmatrix} 1 & 2 & 2 & 3 & 4 & 4 \\ 1 & 1 & 2 & 3 & 2 & 4 \\ 2.5 & 1.2 & 3.0 & 4.0 & 0.8 & 2.5 \end{bmatrix}$$

The best storage format is usually dictated by the particular application, or by the software library that one decides to use. See [Saa94] for a description of *thirteen* common choices.

Sparse matrix storage formats and algorithms consume a bit of extra overhead that dense matrix algorithms do not share. So in practice, sparse methods are actually be *slower* and consume *more memory* unless the matrices involved are "sufficiently" sparse. Unfortunately, the exact break-even point depends upon the sizes and sparsity patterns of the matrices involved, as well as the specific calculations to be performed. However, a common general rule is that sparse methods are employed for a matrix when the average number of elements per row or column is a small constant, independent of the size of the matrix. Expressed more technically, the means that $n_{\mathrm{nz}} = O(m+n)$ for $A \in \mathbb{R}^{m \times n}$ above.

4.1 Sparse BLAS

Is there a sparse version of the BLAS? Well, yes and no. Because of the variety of storage formats—and the fact that no one format is superior for all applications—there is not a mature, *de facto* standard library for sparse operations that mimics the dense BLAS. However, there are a number of close attempts, such as:

- *Sparse BLAS*: The BLAS Technical Forum [Bla01] has defined a standard interface for sparse matrix operations, using an abstraction approach that shields the user from the specific internal data storage format employed for the matrices. However, a reference implementation has not yet been completed, and as such mature, optimized versions are not available.

- *NIST Sparse BLAS* [RP97]: This is an alternate implementation of a sparse BLAS that was submitted to the BLAS Technical Forum. Unlike the BLAST standard, an implementation of this system is available at the cited Web site.

- *The Matrix Template Library* [LSL01]: The `C++` library mentioned above also provides support for sparse matrices.

4.2 Sparse factorizations

There are quite a few libraries for *factorizing* sparse matrices and solving sparse linear systems that are mature and readily available. These include SPOOLES [AGL$^+$99], SuperLU [DGL03], TAUCS [TCR03], and UMFPACK [Dav03]. As with LAPACK, the routines are tailored to exploit different types of matrix structure:

- $A = PLL^TP^T$ (Cholesky) factorizations for symmetric positive definite (SPD) systems;

- $A = PLDL^TP^T$ factorizations for symmetric indefinite systems; and

- $A = P_1LUP_2^T$ factorizations for general unsymmetric matrices.

In the above factorizations, L represents a lower triangular matrix; U an upper triangular matrix; D a block diagonal matrix with 1×1 or 2×2 blocks; and P, P_1, and P_2 permutation matrices. See §C.3 in [BV] for more information about these factorizations.

Permutation matrices effectively rearrange the rows and/or columns of the matrix before the true numerical factorization is performed. For this reason, permutations are often called *orderings* and are typically stored simply as vectors of indices describing the rearrangement. LAPACK optionally computes orderings to improve the accuracy of its calculations. But for sparse matrices, orderings are critically important, because they can significantly impact the sparsity of the resulting factorizations, and therefore sthe performance of any solver that uses them. For a somewhat extreme example, consider the 50×50, symmetric positive definite "arrow" matrix whose spy diagram is depicted in Figure 2(a). If no permutation is performed, than the Cholesky factorization of this matrix is fully dense (Figure 2(b)). On the other hand, by choosing a permutation matrix P that reverses the rows and columns of the matrix, the Cholesky factor is indeed sparse (Figure 2(c)).

Clearly, choosing an effective ordering is very important when working with sparse matrices. Unfortunately, the general problem of choosing a permuation which results in the sparsest possible factorization is a combinatorial graph-manipulation problem. However, a number of effective heuristics have been developed which produce particularly good results in practice. Furthermore, permutations can in theory have undesirable effects on numerical accuracy; fortunately, in practice this is usually not a problem.

Note that the spy diagrams of the Cholesky factors L in Figure 2(b)-(c) are valid no matter what the exact numerical values of the original matrix were—that is, assuming that the matrix is positive definite. In fact, they depend only on the sparsity pattern of the original matrix A, and on the rearrangement induced by the permutation matrix P. This is also true for LU factorizations: as long as the original matrix A is nonsingular, then the sparsity patterns of L and U depend only on the specific choices of P_1 and P_2 and the sparsity pattern of A.

This property is often exploited in practice by dividing the factorization into two steps: a *symbolic* step and the *numerical* step. The symbolic step computes an appropriate set of perumutations

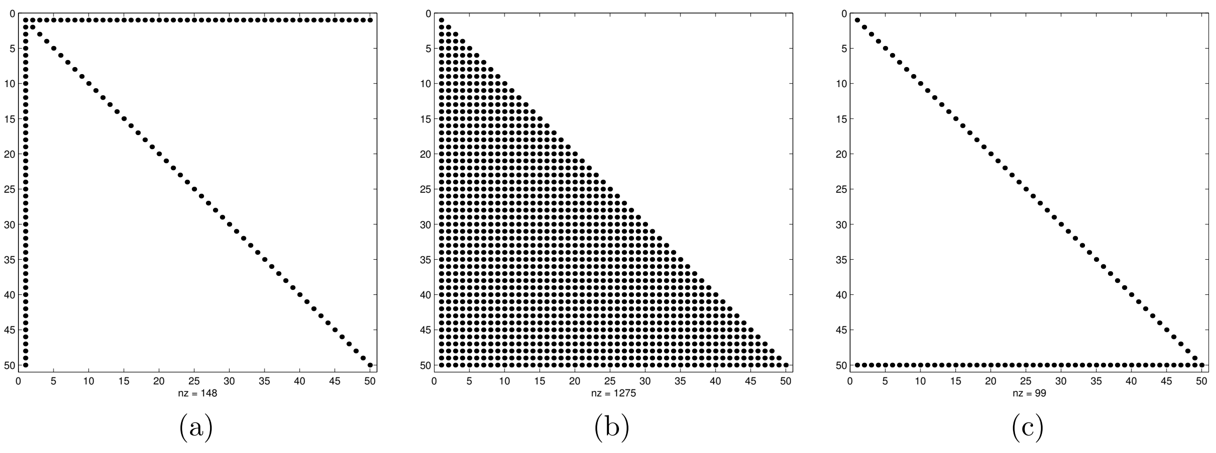

Figure 2: Sparse factorization of an "arrow" matrix: (a) original matrix; (b) Cholesky factor with $P = I$ (no permutation); (c) Cholesky factor with P chosen to reverse the rows and columns.

that (heuristically) maximize sparsity, and using these computes the sparsity pattern of the resulting factors L, U. The second step completes the process by computing the values of the non-zero elements. This two-stage approach provides significant savings when solving *multiple* linear systems with the same sparsity pattern—because the symbolic factorization can be computed only once, and shared among all of the linear systems. This precise scenario occurs quite often in iterative methods for optimization.

5 Final comments

We have only introduced a fraction of the software that is available for numerical linear algebra. There are a number of other areas that have received considerable attention in the field of numerical linear algebra that simply would not fit the time constraints of this lecture:

- *Iterative* methods for sparse and structured linear systems
- Parallel and distributed methods (MPI)
- Fast linear operators: fast Fourier transforms (FFTs), convolutions, state-space linear system simulations, *etc.*
- Low-level details of various factorization and solution methods

In all of these cases, however, there is considerable existing research, and accompanying public domain (or freely licensed) code, which likely can be applied to any problem you will encounter. While certainly your application, and even some of the key computations, may be unique, the bulk of the computational work will likely resemble calculations performed in many other fields. Hence, it makes sense to benefit from the prior efforts of others.

6 Appendix: The Fortran legacy

One of the potential pitfalls of using existing numerical algebra libraries is that many of are implemented in Fortran (specifically, Fortran 77). Using Fortran libraries with C, C++, and other more modern language poses certain interoperability issues.

In many cases, these libraries now include C or C++ "wrapper" code that shields you from many of these issues. Certainly, when such wrappers exist, you should use them. Unfortunately, not all libraries provide them. In this section, we highlight some of the issues you will encounter if you attempt to call a Fortran routine directly. The reader is encouraged to utilize Web resources for further information on calling Fortran routines from C and C++; a simple search for "Calling Fortran from C" on Google will yield a wealth of information.

6.1 Data types

When declaring variables in C that you intend to pass to Fortran functions, it is imperative that you use data types that correspond to Fortran data types. Failing to use the

Fortran	C	Fortran	C
`INTEGER`	`int`	`CHARACTER`	`char`
`LOGICAL`	`int`	`CHARACTER*(*)`	`char*`
`REAL*8`	`double`	`REAL*4`	`float`
`DOUBLE PRECISION`	`double`	`REAL`	`float`
`DOUBLE COMPLEX`	(see §6.4)	`COMPLEX`	(see §6.4)

The above table is very likely correct, but is necessarily not definitive, particularly with regards to `INTEGER` data.

6.2 Subroutine names and argument passing

Fortran compilers often modify the names of the subroutines and functions as they compile, for certain implementation-dependent reasons. (C++ compilers do this as well, although for different reasons.) This practice is completely transparent to a user writing code exclusively in Fortran; but in order to call a Fortran subroutine from C, you must call it by its modified name. In *most* cases, this involves converting the Fortran name to all lowercase, and adding an underscore.

In addition, in order to call a Fortran subroutine from C, you must actually pass *pointers* to all of the scalar (that is, non-array) arguments. For those familiar with the science of computer languages, the reason for this is that Fortran passes scalar arguments *by reference*, while C passes them *by value*. C arrays are passed by reference, so they are exempt from this rule.

So, for example, consider a Fortran function defined as follows:

```
DOUBLE PRECISION FUNCTION DOT_PRODUCT( N, X, Y )
DOUBLE PRECISION X(*), Y(*)
INTEGER N
```

An ANSI C prototype of this function would look like

```
double dot_product_( int* n, double* x, double* y );
```

Here is an example of its usage:

```
double x[10], y[10], ans;
int ten = 10;
/* fill x and y with data */
ans = dot_product_( &ten, x, y );
```

Note how the numeric constant `10` had to be stored in a temporary variable so that a pointer to it could be passed to the function.

6.3 1-D vector indexing

In Fortran, the first element in an array has an index of 1, while in C and C++ the first element has an index of 0. This difference will usually not manifest itself unless the Fortran function either returns as output or expects as input, an array index. For example, the BLAS (see §2) routine IDAMAX

```
INTEGER FUNCTION IDAMAX( N, DX, INCX )
DOUBLE PRECISION DX(*)
INTEGER N, INCX
```

returns the index of the largest element of an array (in absolute value). To properly interpret this function's output in C, you must subtract one:

```
int argmax = idamax_( &n, dx, &one ) - 1;
```

6.4 Complex numbers

Fortran includes built-in support for single- and double-precision complex numbers in its language. The C language does not—although the C99 standard includes such support, it is not yet fully implemented in compilers like `gcc`. So to emulate Fortran's complex number data type is a bit of a challenge. The most reliable way to do so is to create a double-precision array of twice the desired length, and store the real and imaginary parts in alternating order (real first). For example, the C equivalent to the Fortran array above might look like this:

```
double x[20];
#define x_real( k ) (x[2*k])
#define x_imag( k ) (x[2*k+1])
```

The macros `x_real` and `x_imag` are provided here to show how one would access the real and imaginary portions of the k-th number, respectively.

Obviously, this is rather inconvenient, so another option is to use the C `struct` feature to define a complex number data structure:

```
typedef struct { double re, im; } Fortran_Complex;
Fortran_Complex x[10];
```

Unfortunately, the C language standard cannot guarantee that this will work interchangeably with Fortran's `DOUBLE COMPLEX` data type. However, it very *likely* will work; and indeed it *will* work if the following is true:

```
sizeof(Fortran_Complex) == 2*sizeof(double)
```

Therefore, if you use this technique, make sure to place this check in your program. As you can imagine, certain C++ classes could also be used to implement Fortran-compatible complex numbers, as long as the above size requirement is satisifed for them as well.

6.5 2-D array indexing

C, C++, and Java use a *row-major* storage format for 2-dimensional arrays. This means that the elements are stored in contiguous memory, one complete row at a time:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \Longrightarrow \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

Fortran, on the other hand, uses *column-major* storage; which, as the name implies, stores the elements of a 2D array one *column* at a time.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \implies \begin{bmatrix} 1 & 4 & 7 & 2 & 5 & 8 & 3 & 6 & 9 \end{bmatrix}$$

Effectively, this means that C stores the *transpose* of Fortran arrays (or vice versa). To deal with this difference, one has several options:

- If a subroutine can work with a matrix *or* its transpose, make the appropriate adjustment in the function call.
- When possible, utilize a C wrapper which manages the row-major/column-major differences automatically. The standardized C BLAS interface (§2) does this, for example.
- *Bypass* C's built-in support for 2D arrays, and store matrices as *one-dimensional* vectors, in column-major form.

This final option may seem at first unreasonable, because it forces the user to manually implement 2D matrix indexing to access individual elements of a matrix. However, there are three reasons why this disadvantage is not significant:

- By fully exploiting external libraries for matrix manipulation and computation will minimize the need for 2D indexing.
- Matrix factorization methods assign special meaning to the columns of certain matrices—suggesting that it would be convenient if the elements of each column were contiguous. For example, in an eigenvalue decomposition $AX = X\Lambda$, the eigenvectors form the columns of X. In column-major storage, each eigenvector is stored contiguously, simplifying their use in further calculations.
- As you will see shortly, *packed* and *sparse* storage bypass *both* C and Fortran's 2D indexing ability, so it is often not possible to avoid the need to perform manual indexing calculations, even if row-major storage is used.

For these reasons, I personally recommend this final option: storing matrices in column-major format, even in C.

References

[ABB+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorenson. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, third edition, August 1999. Web site: `http://www.netlib.org/lapack/lug/index.html`.

[AGL+99] Cleve Ashcraft, Roger Grimes, Joseph Liu, Jim Patterson, Dan Pierce, Yichi Pierce, Peter Schartz, Juergen Schulze, Wei-Pai Tang, David Wah, and Jason Wu. SPOOLES 2.2: SParse Object Oriented Linear Equation Solver. Web site: `http://www.netlib.org/linalg/spooles/spooles.2.2.html`, January 1999.

[Bla01] Susan Blackford, editor. *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard*. August 2001. Web site: `http://www.netlib.org/blas/blast-forum/blas-report.pdf`.

[BLA03] BLAS (Basic Linear Algebra Subprograms). Web site: `http://www.netlib.org/blas`, 2003.

[BV] Stephen P. Boyd and Lieven Vandenberghe. Convex optimization. Available at `http://www.stanford.edu/~boyd`. To be published in 2003.

[Dav03] Timothy A. Davis. UMFPACK version 4.1. Web site: `http://www.cise.ufl.edu/research/sparse/umfpack`, April 2003.

[DCDH90] J. J. Dongarra, J. Du Croz, I S. Duff, and R. J. Hanson. Algorithm 679: An set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:18–28, 1990.

[DCHH88] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. Algorithm 656: An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 5:18–32, 1988.

[DGB$^+$03] Jack Dongarra, Eric Grosse, Petter Bjorstad, Maggie Bowman, David Gay, Tim Hopkins, and Cleve Moler. Netlib. Web site: `http://www.netlib.org`, 2003.

[DGL03] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. SuperLU users' guide. Technical report, Computer Science Division, University of California, Santa Barbara, March 2003. Web site: `http://crd.lbl.gov/~xiaoye/SuperLU`.

[Don03] Jack Dongarra. Freely available software for linear algebra on the web. Web site: `http://www.netlib.org/utk/people/JackDongarra/la-sw.html`, August 2003.

[Int03] Intel Corporation. Intel math kernel library, version 6.0. Web site: `http://developer.intel.com/software/products/mkl`, September 2003.

[LAP00] LAPACK – Linear Algebra PACKage, version 3.0. Web site: `http://www.netlib.org/lapack`, May 2000.

[LHKK79] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Algorithm 539: Basic linear algebra subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979.

[LSL01] Andrew Lumsdaine, Jeremy Siek, and Lie-Quan Lee. The matrix template library. Web site: `http://www.osl.iu.edu/research/mtl/intro.php3`, August 2001.

[PFTV93] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1993.

[RP97] Karin Remington and Roldan Pozo. The NIST sparse blas (v0.9). Web site: `http://math.nist.gov/spblas/`, June 1997.

[Saa94] Youcef Saad. SPARSKIT: a basic tool kit for sparse matrix computations. Technical report, University of Minnesota, Department of Computer Science and Engineering, June 1994. Web site: `http://www-users.cs.umn.edu/~saad/software/SPARSKIT/paper.ps`.

[TCR03] Sivan Toledo, Doron Chen, and Vladimir Rotkin. TAUCS: A library of linear sparse solvers. Web site: `http://www.tau.ac.il/~stoledo/taucs`, September 2003.

[WPD00] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the atlas project. Technical report, Department of Computer Science, University of Tennessee, Knoxville, September 2000.

[WPD03] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automatically tuned linear algebra software (ATLAS). Web site: `http://math-atlas.sourceforge.net`, August 2003.