Numerical Linear Algebra Software

Michael C. Grant

- Introduction
- Basic computations: BLAS, ATLAS
- Linear systems and factorizations: LAPACK
- Sparse matrices

Numerical linear algebra in optimization

Many optimization algorithms ($e.g.$ barrier methods, primal/dual methods) requires the solution of a sequence of structured linear systems; for example, Newton's method:

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

The bulk of *memory usage* and *computation time* is spent constructing and solving these systems. How do we know this?

- theoretical complexity analysis
- *profiling*—a method for measuring the resources consumed by various subroutines of a program during execution

Therefore, writing efficient code to solve optimization problems requires competence in numerical linear algebra software

How to write numerical linear algebra software

DON'T!

Whenever possible, rely on *existing, mature* software libraries for performing numerical linear algebra computations. By doing so...

- You can focus on other important issues, such as the higher-level 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
- Your code will run faster—sometimes *dramatically* so

Netlib

A centralized clearinghouse for some of the best-known numerical software:

`http://www.netlib.org`

- Maintained by the University of Tennessee, the Oak Ridge National Laboratory, and colleagues worldwide
- Most of the code is public domain or freely licensed
- Much written in FORTRAN 77 (gasp!)

A useful list of freely available linear algebra libraries:

`http://www.netlib.org/utk/people/JackDongarra/la-sw.html`

The Basic Linear Algebra Subroutines (BLAS)

Written by people who had the foresight to understand the future benefits of a standard suite of "kernel" routines for linear algebra.

Created and organized in three *levels*:

- *Level 1*, 1973-1977: $O(n)$ vector operations: addition, scaling, dot products, norms
- *Level 2*, 1984-1986: $O(n^2)$ matrix-vector operations: matrix-vector products, triangular matrix-vector solves, rank-1 and symmetric rank-2 updates
- *Level 3*, 1987-1990: $O(n^3)$ matrix-matrix operations: matrix-matrix products, triangular matrix solves, low-rank updates

BLAS operations

Level 1	addition/scaling	$\alpha x, \quad \alpha x + y$
	dot products, norms	$x^T y, \quad \|x\|_2, \quad \|x\|_1$
Level 2	matrix/vector products	$\alpha Ax + \beta y, \quad \alpha A^T x + \beta y$
	rank 1 updates	$A + \alpha x y^T, \quad A + \alpha x x^T$
	rank 2 updates	$A + \alpha x y^T + \alpha y x^T$
	triangular solves	$\alpha T^{-1} x, \quad \alpha T^{-T} x$
Level 3	matrix/matrix products	$\alpha AB + \beta C, \quad \alpha AB^T + \beta C$
		$\alpha A^T B + \beta C, \quad \alpha A^T B^T + \beta C$
	rank-k updates	$\alpha AA^T + \beta C, \quad \alpha A^T A + \beta C$
	rank-$2k$ updates	$\alpha A^T B + \alpha B^T A + \beta C$
	triangular solves	$\alpha T^{-1} C, \quad \alpha T^{-T} C$

Level 1 BLAS naming convention

All BLAS routines have a Fortran-inspired naming convention:

`cblas_`	`X`	`XXXX`
prefix	data type	operation

Data types:

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

Operations:

`axpy`	$y \leftarrow \alpha x + y$	`dot`	$r \leftarrow x^T y$
`nrm2`	$r \leftarrow \|x\|_2 = \sqrt{x^T x}$	`asum`	$r \leftarrow \|x\|_1 = \sum_i \|x_i\|$

Example:

`cblas_ddot` double precision real dot product

BLAS naming convention: Level 2/3

`cblas_`	`X`	`XX`	`XXX`
prefix	data type	structure	operation

Matrix structure:

`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

Operations:

`mv`	$y \leftarrow \alpha Ax + \beta y$	`sv`	$x \leftarrow A^{-1}x$ (triangular only)
`r`	$A \leftarrow A + xx^T$	`r2`	$A \leftarrow A + xy^T + yx^T$
`mm`	$C \leftarrow \alpha AB + \beta C$	`r2k`	$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$

Examples:

`cblas_dtrmv`	double precision real triangular matrix-vector product
`cblas_dsyr2k`	double precision real symmetric rank-$2k$ update

Using BLAS efficiently

Always choose a higher-level BLAS routine over multiple calls to a lower-level BLAS routine, even when the results would be the same

$$A \leftarrow A + \sum_{i=1}^{k} x_i y_i^T, \quad A \in \mathbb{R}^{m \times n},\ x_i \in \mathbb{R}^m,\ y_i \in \mathbb{R}^n$$

Two choices: k separate calls to the Level 2 routine `cblas_dger`

$$A \leftarrow A + x_1 y_1^T, \quad A \leftarrow A + x_2 y_2^T, \quad \ldots \quad A \leftarrow A + x_k y_k^T$$

or a single call to the Level 3 routine `cblas_dgemm`

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

The Level 3 choice will perform much better in practice.

Is BLAS necessary?

Why use BLAS at all when writing your own routines is so easy?

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

$$A_{ij} \leftarrow A_{ij} + \sum_{k=1}^{p} X_{ik} Y_{jk}$$

```
void matmultadd( int m, int n, int p, double* A,
                 const double* X, const double* Y ) {
    int 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 ];
}
```

Why use BLAS when the code is this simple?

Is BLAS necessary?

The answer: *performance*—a factor of *5 to 10 times* or more.

It is *not* trivial to write numerical code on today's computers that achieves high performance, due to pipelining, memory bandwidth, cache architectures, $etc.$

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

One free example: ATLAS

`http://math-atlas.sourceforge.net`

uses automated code generation and testing methods (basically, automated trial-and-error) to *generate* an optimized BLAS library for a specific computer

Improving performance through blocking

Blocking can be used to improve the performance of matrix/vector and matrix/matrix multiplications, Cholesky factorizations, $etc.$.

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

The optimal block size is *not* easy to determine: it depends on the specific architecture of the processor, cache, and memory. But once they are chosen, each block can be computed separately, and ordered in a manner which minimizes memory traffic; $e.g.$

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

The Matrix Template Library

Having just sung the praises of the BLAS, here is an alternative for C++:

`http://www.osl.iu.edu/research/mtl/intro.php3`

- Uses the most advanced features of C++ for flexibility and performance
- Supports structured matrices (triangular, symmetric, banded, *etc.*)
- Uses blocked algorithms to improve performance
- Also supports LAPACK, sparse matrices
- Attractively licensed

It looks interesting. Has anyone tried this?

The Linear Algebra PACKage (LAPACK)

LAPACK contains a variety of subroutines for solving linear systems and performing a number of common matrix decompositions and factorizations.

- First release: February 1992; latest version (3.0): May 2000
- Supercedes predecessors EISPACK and LINPACK
- Supports the same data types (single/double precision, real/complex) and matrix structure types (symmetric, banded, $etc.$) as BLAS
- Uses BLAS for internal computations
- Routines divided into three categories: *auxiliary* routines, *computational* routines, and *driver* routines

LAPACK Computational 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
- symmetric/Hermitian and nonsymmetric eigenvalue decompositions
- singular value decompositions
- generalized eigenvalue and singular value decompositions

LAPACK Driver Routines

Driver routines call computational routines in sequence to solve a variety of standard linear algebra problems, including:

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

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

- 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:

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

A LAPACK usage example

Consider the task of solving

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

for $d_x \in \mathbb{R}^n$, $d_v \in \mathbb{R}^m$, where

$$P \in \mathbb{R}^{n \times n}, \quad P = P^T \succ 0$$

$$A \in \mathbb{R}^{m \times n}, \quad r_a \in \mathbb{R}^n, \quad r_b \in \mathbb{R}^m, \quad (m < n)$$

Linear systems with this structure occur quite often in KKT-based optimization algorithms

A LAPACK usage example: option 1

The coefficient matrix is symmetric indefinite, so a permuted LDL^T factorization exists:

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

The *computational routine* `dsytrf` performs this factorization

the *driver routine* `dsysv` uses `dsytrf` to compute this factorization and performs the remaining computations to determine the solution:

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

A LAPACK usage example: option 2

A bit of algebraic manipulation yields

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

So first, we solve the system

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

for $\tilde{A}$ and $\tilde{r}_a$; then we can construct and solve

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

using a *second* call to `dspsv`; finally, $d_x = \tilde{r}_a - \tilde{A}d_v$.
The *driver routine* `dspsv` solves both of these systems, using the *computational routine* `dsptrf` to perform Cholesky factorizations

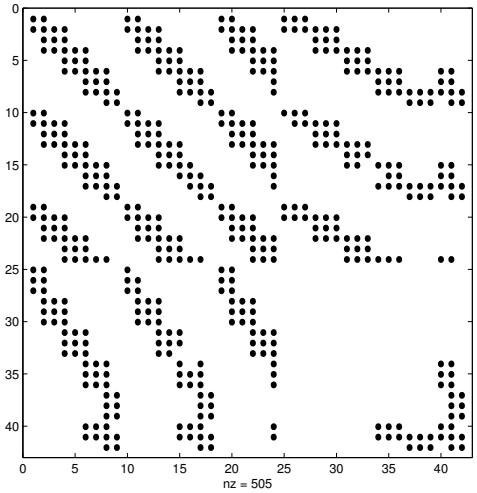
Why choose?

Why even consider the second method if the first seems so straightforward?

- In some applications, the inverse of P, or its Cholesky factorization, may be readily available or particularly easy to compute
- Structure in P can be more easily exploited
- If A and P are sparse, the Cholesky factorization is often preferred over the LDL^T factorization (see below)

It is worthwhile, therefore, to consider all equivalent options and be able to choose between them

Sparse matrices



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

- the number of non-zero elements $n_{\rm nz}$ is small; *i.e.*, $n_{\rm nz} \ll mn$. 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

Why sparse matrices?

Sparse matrices can save memory and time

- Storing a matrix $A \in \mathbb{R}^{m \times n}$ using double precision numbers:
  - Dense: $8mn$ bytes
  - Sparse: $\approx 16n_{\mathrm{nz}}$ bytes or less, depending on the storage format
- The operation $y \leftarrow y + Ax$:
  - Dense: mn multiplies and additions
  - Sparse: n_{nz} multiplies and additions
- The operation $x \leftarrow T^{-1}x$, $T \in \mathbb{R}^{n \times n}$ triangular and nonsingular:
  - Dense: n divides, $n(n-1)/2$ multiplies and additions
  - Sparse: n divides, $n_{\mathrm{nz}} - n$ multiplies and additions

Representing sparse matrices

There are a variety of methods for efficiently representing a sparse matrix on a computer.

One simple method is to store the data in a $3 \times n_{\mathrm{nz}}$ matrix: row indices in the first row, column indices in the second, values in the third

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

The best storage format is usually dictated by the particular application or computational library

Sparse methods do incur additional overhead, so they are generally used when matrices are "very" sparse; *e.g.*, $n_{\mathrm{nz}} = O(m+n)$

Sparse BLAS?

Unfortunately there is *not* a mature, *de facto* standard sparse matrix BLAS library. Three potential options:

- "Official" *Sparse BLAS*: a reference implementation is not yet available

  `http://www.netlib.org/blas/blast-forum`

- *NIST Sparse BLAS*: An alternative BLAS system; a reference implementation is available

  `http://math.nist.gov/spblas`

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

  `http://www.osl.iu.edu/research/mtl/intro.php3`

Sparse factorizations

There are quite a few libraries for *factorizing* sparse matrices and solving sparse linear systems, including:

- SPOOLES: `http://www.netlib.org/linalg/spooles`
- SuperLU: `http://crd.lbl.gov/~xiaoye/SuperLU`
- TAUCS: `http://www.tau.ac.il/~stoledo/taucs`
- UMFPACK: `http://www.cise.ufl.edu/research/sparse/umfpack`

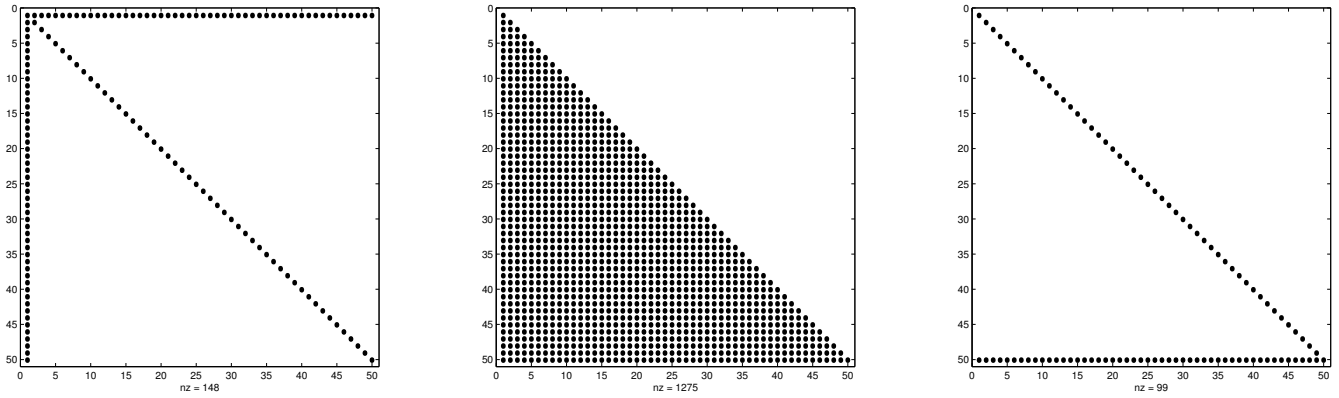
Several factorization methods are available:

- $A = PLL^TP^T$ (Cholesky) for symmetric positive definite systems
- $A = PLDL^TP^T$ for symmetric indefinite systems
- $A = P_1LUP_2^T$ for general unsymmetric matrices

P, P_1, P_2 are permutations or *orderings*

Sparse orderings

Sparse orderings can have a *dramatic* effect on the sparsity of a factorization



These *spy diagrams* depict the sparsity of a sparse "arrow" matrix and two Cholesky factorizations:

- Left: original matrix
- Center: Cholesky factorization with no permutation ($P = I$)
- Right: Cholesky factorization with the best permutation

Sparse orderings (continued)

The sparsity of the factorization has a direct bearing the performance, so choosing an effective ordering is *very* important

- The general problem of choosing the ordering that produces the sparsest factorization is a combinatorial (NP-hard) problem
- A number of effective heuristics have been developed which produce particularly good results in practice
- In theory, permutations can also have undesirable effects on numerical accuracy—in practice, this is usually not a problem

Symbolic factorization

- For Cholesky and LU factorizations, the sparsity pattern of the factorization depends only on the ordering chosen and the sparsity pattern of the matrix—*not* on the numerical values of the non-zero elements
- This property enables these factorizations to be divided into two stages: *symbolic* factorization and *numerical* factorization
- When solving *multiple* linear systems with identical sparsity patterns—for example, in an iterative optimization algorithm—the symbolic factorization may be computed just once
- LDL^T factorizations for symmetric indefinite matrices do not support symbolic factorization

Other areas

There are a number of other areas in numerical linear algebra that have received significant attention:

- *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, there is considerable existing research, and accompanying public domain (or freely licensed) code

Conclusion

Certainly, your applications, and even some of the key computations, may be unique...

But the bulk of the computational work will likely resemble calculations performed in many other fields

So it makes sense to benefit from the prior efforts of others

The Fortran Legacy

Many of the best numerical libraries are written in FORTRAN 77, and not all have been blessed with "C wrappers"

How do you call FORTRAN from C?

- Mapping data types
- Subroutine calling
- Vector indexing
- Matrix indexing
- Complex numbers

The solutions depend in part on the *operating system*

FORTRAN to C: data types

Mapping FORTRAN data types to C:

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`	(later)	`COMPLEX`	(later)

Unfortunately this mapping is *not* definitive, particularly for `INTEGER` and `LOGICAL`, so make sure to check your system's and compiler's documentation. (This is accurate for Linux and Windows)

FORTRAN Subroutines

Key points:

- FORTRAN names are case insensitive; C is case sensitive.

  FORTRAN subroutine names are *generally* converted to lowercase, and an underscore _ is often appended to them

- FORTRAN differentiates between `SUBROUTINE` a `FUNCTION`; C basically does not

  Treat `SUBROUTINE`s as `void` functions in C

- FORTRAN passes all arguments *by reference*; C passes scalar arguments *by value*, and vectors/matrices *by reference*

  When calling FORTRAN functions from C, pass *pointers* to scalar arguments, not the values themselves.

FORTRAN Subroutines (continued)

FORTRAN prototype:

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

ANSI C prototype:

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

Example usage:

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

Vector indexing

FORTRAN indexes vectors from 1; C indexes them from 0. This is only a problem if a routine *returns* an index

For example, the BLAS routine IDAMAX:

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

To properly interpret this function's output in C, you must subtract one:

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

Matrix indexing

C, C++, and Java store matrices in *row-major* format:

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

But FORTRAN stores matrices in *column-major* format:

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

My recommendation: do not use C's built-in support for 2D matrices, and store matrices in FORTRAN (column-major) format

Complex numbers

FORTRAN has a built-in complex number data type; C does not. Two solutions: Store complex data in real vectors:

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

Or define a `struct`

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

The `complex<double>` class in C++ should work as well