Numerical Linear Algebra Software

(based on slides written by Michael Grant)

- BLAS, ATLAS
- LAPACK
- sparse matrices

Numerical linear algebra in optimization

most *memory usage* and *computation time* in optimization methods is spent on numerical linear algebra, $e.g.$,

- constructing sets of linear equations ($e.g.$, Newton or KKT systems)
  - matrix-matrix products, matrix-vector products, . . .
- and solving them
  - factoring, forward and backward substitution, . . .

. . . so knowing about numerical linear algebra is a good thing

Why not just use Matlab?

- Matlab (Octave, . . .) is OK for prototyping an algorithm

- but you'll need to use a real language ($e.g.$, C, C++, Python) when
  - your problem is very large, or has special structure
  - speed is critical ($e.g.$, real-time)
  - your algorithm is embedded in a larger system or tool
  - you want to avoid proprietary software

- in any case, the numerical linear algebra in Matlab is done using standard free libraries

How to write numerical linear algebra software

DON'T!

whenever possible, rely on *existing, mature* software libraries

- you can focus on the higher-level algorithm
- your code will be more portable, less buggy, and will run faster—sometimes *much* faster

Netlib

the grandfather of *all* numerical linear algebra web sites

`http://www.netlib.org`

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

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

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

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

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

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

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

$$A \leftarrow A + XY^T, \quad X = [x_1 \cdots x_k], \quad Y = [y_1 \cdots y_k]$$

the Level 3 choice will perform much better

Is BLAS necessary?

why use BLAS when writing your own routines is so easy?

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

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

```
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 ];
}
```

Is BLAS necessary?

- tuned/optimized BLAS will run faster than your home-brew version — often $10\times$ or more

- BLAS is tuned by selecting block sizes that fit well with your processor, cache sizes

- ATLAS (automatically tuned linear algebra software)

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

  uses automated code generation and testing methods to *generate* an optimized BLAS library for a specific computer

Improving performance through blocking

blocking is used to improve the performance of matrix/vector and matrix/matrix multiplications, Cholesky factorizations, etc.

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

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

optimal block size, and order of computations, depends on details of processor architecture, cache, memory

Linear Algebra PACKage (LAPACK)

LAPACK contains subroutines for solving linear systems and performing common matrix decompositions and factorizations

- first release: February 1992; latest version (3.0): May 2000
- supercedes predecessors EISPACK and LINPACK
- supports same data types (single/double precision, real/complex) and matrix structure types (symmetric, banded, . . .) as BLAS
- uses BLAS for internal computations
- routines divided into three categories: *auxiliary* routines, *computational* routines, and *driver* routines

LAPACK computational routines

computational routines perform single, specific 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 a sequence of computational routines to solve standard linear algebra problems, such as

- linear equations: $AX = B$
- linear least squares: minimize$_x$ $\|b - Ax\|_2$
- linear least-norm:

$$\begin{array}{ll} \text{minimize}_y & \|y\|_2 \\ \text{subject to} & d = By \end{array}$$

- generalized linear least squares problems:

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

LAPACK example

solve KKT system

$$\begin{bmatrix} H & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}$$

$x \in \mathbf{R}^n$, $v \in \mathbf{R}^m$, $H = H^T \succ 0$, $m < n$

option 1: driver routine `dsysv` uses computational routine `dsytrf` to compute permuted LDL^T factorization

$$\begin{bmatrix} H & A \\ A & 0 \end{bmatrix} \rightarrow PLDL^TP^T$$

and performs remaining computations to compute solution

$$\begin{bmatrix} x \\ y \end{bmatrix} = P^TL^{-1}D^{-1}L^{-T}P \begin{bmatrix} a \\ b \end{bmatrix}$$

option 2: block elimination

$$y = (AH^{-1}A^T)^{-1}(AH^{-1}a - b), \qquad x = H^{-1}a - H^{-1}A^T y$$

- first we solve the system $H[Z\ w] = [A^T\ a]$ using driver routine `dspsv`
- then we construct and solve $(AZ)y = Aw - b$ using `dspsv` again
- $x = w - Zy$

using this approach we could exploit structure in H, *e.g.*, banded

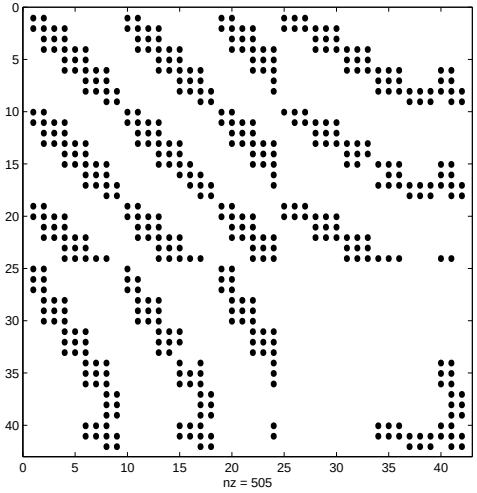
What about other languages?

BLAS and LAPACK routines can be called from C, C++, Java, Python, . . .

an alternative is to use a "native" library, such as

- C++: Boost uBlas, Matrix Template Library
- Python: NumPy/SciPy, CVXOPT
- Java: JAMA

Sparse matrices



- $A \in \mathbf{R}^{m \times n}$ is *sparse* if it has "enough zeros that it pays to take advantage of them" (J. Wilkinson)
- usually this means $n_{\rm nz}$, number of elements known to be nonzero, is small: $n_{\rm nz} \ll mn$

Sparse matrices

sparse matrices can save memory and time

- storing $A \in \mathbf{R}^{m \times n}$ using double precision numbers
  - dense: $8mn$ bytes
  - sparse: $\approx 16 n_{\mathrm{nz}}$ bytes or less, depending on storage format

- operation $y \leftarrow y + Ax$:
  - dense: mn flops
  - sparse: n_{nz} flops

- operation $x \leftarrow T^{-1}x$, $T \in \mathbf{R}^{n \times n}$ triangular, nonsingular:
  - dense: $n^2/2$ flops
  - sparse: n_{nz} flops

Representing sparse matrices

- several methods used
- simplest (but typically not used) is to store the data as list of (i, j, A_{ij}) triples
- column compressed format: an array of pairs (A_{ij}, i), and an array of pointers into this array that indicate the start of a new column
- for high end work, exotic data structures are used
- sadly, no universal standard (yet)

Sparse BLAS?

sadly there is not (yet) a standard sparse matrix BLAS library

- the "official" *sparse BLAS*

  `http://www.netlib.org/blas/blast-forum`
  `http://math.nist.gov/spblas`

- C++: Boost uBlas, Matrix Template Library, SparseLib++

- Python: SciPy, PySparse, CVXOPT

Sparse factorizations

libraries for factoring/solving systems with sparse matrices

- most comprehensive: SuiteSparse (Tim Davis)

  `http://www.cise.ufl.edu/research/sparse/SuiteSparse`

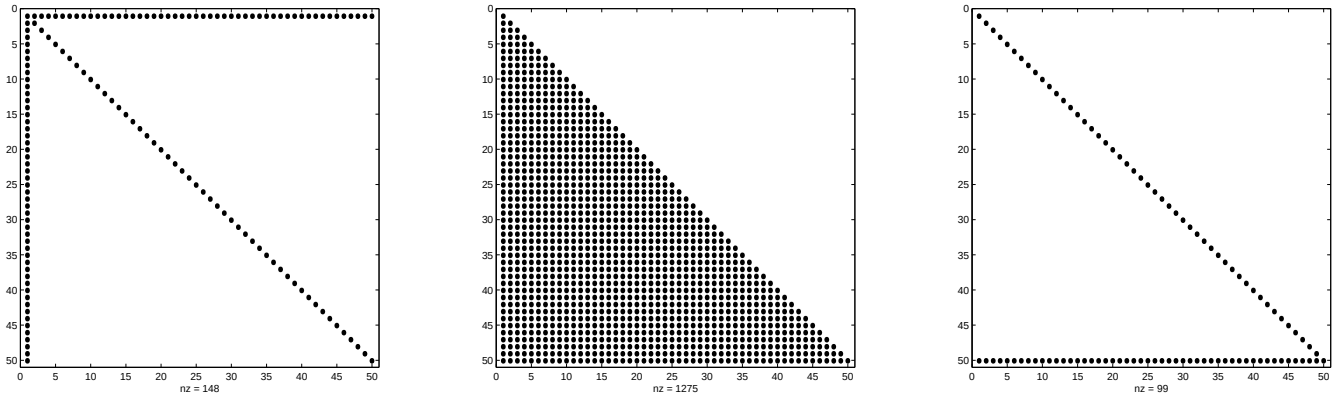
- others include SuperLU, TAUCS, SPOOLES

- typically include
  - $A = PLL^TP^T$ Cholesky
  - $A = PLDL^TP^T$ for symmetric indefinite systems
  - $A = P_1LUP_2^T$ for general (nonsymmetric) 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



- left: spy diagram of original NW arrow matrix
- center: spy diagram of Cholesky factor with no permutation ($P = I$)
- right: spy diagram of Cholesky factor with the best permutation (permute $1 \rightarrow n$)

Sparse orderings

- general problem of choosing the ordering that produces the sparsest factorization is hard
- but, several simple heuristics are very effective
- more exotic ordering methods, $e.g.$, nested disection, can work very well

Symbolic factorization

- for Cholesky factorization, the ordering can be chosen based only on the sparsity pattern of A, and *not* its numerical values

- factorization can be divided into two stages: *symbolic* factorization and *numerical* factorization
  - when solving *multiple* linear systems with identical sparsity patterns, symbolic factorization can be computed just once
  - more effort can go into selecting an ordering, since it will be amortized across multiple numerical factorizations

- ordering for LDL^T factorization usually has to be done on the fly, $i.e.$, based on the data

Other methods

we list some 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

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