Outline

- Matrix Decompositions summary
- Updates
- Iterative methods for linear systems
  - Why?
- Matrix splittings and fixed-point schemes
  - SOR, Jacobi, Gauss Seidel, etc.
- Krylov Methods
  - Conjugate Gradient, GMRES

Updating Factorizations

- Suppose we have LU decomposition of A and we want to solve $(A-uv^t)x=b$
- Solve equation without recomputing LU of new matrix
- Solve $Az=u$, and $Ay=b$, so we have $A^{-1}u$ and $A^{-1}b$
- We can formally write

$$A^{-1}(A-uv^t)x= A^{-1}b \qquad (I-zv^t)x=y$$

- Let $v^tx= \alpha$. (scalar, which depends on x).

$$x- z\alpha=y \quad \text{or} \quad v^tx – v^tz\alpha = v^ty$$

$$\text{or} \quad \alpha – v^tz\alpha = v^ty \qquad \alpha = v^ty/(1 – v^tz)$$

- So $x= y + \alpha z$
- Cost is $O(n^2)$

Sherman Morrison Woodbury Formula

- Assumes A^{-1} is already known.

$$(A - UV^T)^{-1} = A^{-1} + A^{-1}U(I - V^T A^{-1}U)^{-1}V^T A^{-1}$$

- Here U and V are $n \times r$ and $r \times n$ matrices, and this is a rank r update
- So cost of computing the inverse is $O(rn^2)$

Decomposition	Cost	Use
LU	$n^3/3$	• solving linear systems • computing determinants
QR	$mn^2 - 1/3n^3$	• solving well-conditioned linear least squares problems • representing the range or null-space of a matrix
rank-revealing QR	$mn^2 - 1/3n^3$	• solving ill-conditioned linear least squares problems • representing the range or null-space of a matrix

Decomposition	Cost	Use
SVD	$O(mn^2)$	• solving ill-conditioned linear least squares problems • solving discretizations of ill-posed problems • representing the range or null-space of a matrix
eigendecomposition	$O(n^3)$	• determining eigenvalues or eigenvectors of a matrix • determining invariant subspaces • determining stability of a control system • determining convergence

BLAS, LAPACK

- Basic Linear Algebra Subroutines
- Linear Algebra Package
- Routines for single core, cache-efficient, linear algebra
- BLAS – focuses on operations
  - Level 1 – O(n) SAXPY, DAXPY, SNORM, ZNORM, etc
  - Level 2 – $O(n^2)$ SGEMV, etc.
  - Level 3 – $O(n^3)$ SGEMM etc
- LAPACK on decompositions and solutions
  - SGETRF, SGESV, SGBSV …
- Installers which tune to your machine by solving test problems available
- Multicore versions --- PLASMA, MKL
- JAVA, C wrappers`

Basic Costs

- Memory --- $O(n^2)$
- Computation --- $O(n^3)$
- Given 24 GB of RAM on a high end workstation,
  - One double takes 8 bytes,
  - Memory available = $24 \times 1024^3 = 3 \times 2^{30}$ doubles
- So upper bound on the size of a problem is 3×2^{30}
  - largest matrix that can be fit in RAM $1.7 \times 2^{15} \sim 50000$
  - In practice about 1/2 that.
- But we routinely may need to solve larger problems
- Solutions:
  - parallel processing; out-of-core-algorithms;
  - Look for structure in the matrices

Matrices with structure

- Sparse matrix is one in which most elements are zero.
  - If storage space is more important than access speed, it may be preferable to store a sparse matrix as a list of (index, value) pairs.
  - For a given sparsity structure it may be possible to define a fast matrix-vector product/linear system algorithm
- Kronecker structure --- matrix entries created via tensor products

$$A = \begin{bmatrix} 1 & 2 \\ 0 & -1 \end{bmatrix}_{2\times 2} \qquad B_{2\times 3} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{2\times 3}$$

$A \otimes B$ is

$$A \otimes B = \begin{bmatrix} 1 & 2 & 3 & 2 & 4 & 6 \\ 4 & 5 & 6 & 8 & 10 & 12 \\ 0 & 0 & 0 & -1 & -2 & -3 \\ 0 & 0 & 0 & -4 & -5 & -6 \end{bmatrix}_{4\times 6}.$$

Fast matrix vector products via structure

- Fourier, Toeplitz, Circulant …
- Fourier entries are created via raising primitive roots of unity to power *kn*

$$= \sum_{n=0}^{N-1} \phi[n] \; e^{-2\pi i k n / N}$$

- Toeplitz, Hankel, Vandermonde, Circulant etc. are matrices created from displacing entries in a vector
- Fast multipole methods

$$s(x_j) = \sum_{i=1}^{N} \alpha_i \phi(x_j - x_i), \quad \{s_j\} = [\Phi_{ji}]\{\alpha_i\}.$$

- Methods do not need to store the matrix entries

Iterative Methods

- Take home message
  - Iterative methods often require matrix vector products
  - Can be written using special purpose algorithms
  - Some methods can be guaranteed to converge in *N* steps
  - With good guess and clever algorithms may converge much faster

References

- C.T. Kelley, "Iterative methods for Linear and Nonlinear Equations, SIAM, 1995)
- J. Shewchuck, "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain"
- (downloadable from http://www-2.cs.cmu.edu/~jrs/jrspapers.html)
- "Templates for the solution of linear systems: Building Blocks and Iterative Methods," Barrett et al, SIAM
  (downloadable at http://www.netlib.org/linalg/html_templates/Templates.html)
- Yousef Saad has two good books online
  NumericalMethods for Large Eigenvalue Problems
  Iterative methods for sparse linear systems.
  (downloadable at http://www-users.cs.umn.edu/~saad/books.html)

Iterative Methods: Notation

$$Ax = b$$

- A is a nonsingular $N \times N$ matrix, x,b in R^N
- Ax can be calculated using a fast algorithm
- x^* is the solution to be found.
- Definitions

Norm of A	$\|\|A\|\| = \max_{\|\|x\|\|=1} \|\|Ax\|\|$
Condition number of A	$\kappa(A) = \|\|A\|\|\|\|A^{-1}\|\|$
Residual	$r = b - Ax$
Error	$e = x - x^*$

Fixed point iteration

- In fixed point iteration we write $x=Mx$
- If M is a contraction ($||M|| < 1$) then following converges
  - Start with guess x_0
  - Generate successive estimates $x_k=M\,x_{k-1}$
- How to write our equation as a fixed point scheme?

$$Ax=Ix + (A-I)x = b$$

- So, $I\,x = (I-A)x + b$
- (Richardson iteration) $x_{k+1} = (I-A)x_k + b$
- For convergence we require $||I-A|| < 1$
- Iteration Matrix M *here is* $||I-A||$

Classical fixed point methods

- Write $A=A_1+A_2$
- Then iteration becomes $x_{k+1} = A_1^{-1}(b - A_2x_k)$
  - For convergence $||A_1^{-1}A_2|| < 1$
  - A_1^{-1} should be easy to compute
  - In addition the FMM should be used to compute A_2x_k
- Jacobi iteration $A_1=D$ $A_2=L+U$
  - A_1^{-1} is easy to compute (1/ entries along diagonal)
  - This is easy to compute with the FMM
  - At element level $(x_{k+1})_i = a_{ii}^{-1}\left(b_i - \sum_{j\neq i} a_{ij}(x_k)_j\right)$
- Other classical iterations (Gauss-Seidel, SOR).

Krylov methods

- Different class of methods
- Do not involve an iteration matrix
- Motivation: Say in functional space we are at a point x_0 and we want to reach the solution x^*
- Can do it by taking steps along some directions
- Method of steepest descent
- Define function *f(x)*
- So minimum of *f(x)* is attained at $\nabla f(x)=0$

Conjugate Gradient

- Definition: Krylov subspace $\mathcal{K}_k = \mathrm{span}(r_0, Ar_0, \ldots, A^{k-1}r_0)$
- Definition: Energy or *A*-norm of a vector $||x||_A=(x^t Ax)^{1/2}$
- Idea of conjugate gradient method
  - Generate Krylov subspace directions, and take a step that minimizes the *A* norm of the residual along this direction
- Let search direction be at step $k+1$ be d_{k+1}
- We require $f(x_k+\alpha_{k+1} d_{k+1})$ is minimized along direction
- Conjugacy property $d_{k+1}AK^k=0$

Convergence of CG

- If there are k distinct eigenvalues of A, then CG converges in at most k iterations to the exact solution (in exact arithmetic)
- Usually we go to $||b-Ax_k||_2 \leq \eta||b||_2$

$$\|x_* - x_m\|_A \leq 2\left[\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right]^m \|x_* - x_0\|_A.$$

- Converges quickly

$$||e_k||_A \leq \left(\frac{\kappa(A)-1}{\kappa(A)+1}\right)^k ||e_0||_A \quad \text{where} \quad \kappa(A) = \lambda_{max}/\lambda_{min}$$

Non symmetric matrices

- Can't apply CG to nonsymmetric matrices
- One simple solution – CGNR j
  - convert system to a symmetric system

$$A^tAx=A^tb$$

- However we will need two matrix vector multiplies per iteration
- Also, if A is poorly conditioned then A^tA is even more poorly conditioned (condition number is squared)
- Method of choice is GMRES
- Blackbox implementation in Matlab

GMRES

- Instead of requiring minimization along conjugate direction, minimize residual in a subspace
- Krylov subspace $\mathcal{K}_k = \mathrm{span}(r_0, Ar_0, \ldots, A^{k-1}r_0)$
- Require x to minimize $||b-Ax||_2$ $\forall x$ in $x_0 + \mathcal{K}^k$
- Construct basis

$$w^{(i)} = Av^{(i)}$$
$$\textbf{for } k = 1, \ldots, i$$
$$\quad w^{(i)} = w^{(i)} - (w^{(i)}, v^{(k)})v^{(k)}$$
$$\textbf{end}$$
$$v^{(i+1)} = w^{(i)}/\|w^{(i)}\|$$

- Then require that each $x^{(i)} = x^{(0)} + y_1 v^{(1)} + \cdots + y_i v^{(i)}$, satisfies minimum $\|b - Ax^{(i)}\|$
- Can be done by simple minimization
- Implemented as a black-box in Matlab