Numerical Methods I
Non-Square and Sparse Linear Systems

Aleksandar Donev
Courant Institute, NYU

donev@courant.nyu.edu

MATH-GA 2011.003 / CSCI-GA 2945.003, Fall 2014

September 25th, 2014
Homework 1 is due tomorrow, Friday Sept. 19th: go through instructions in class.
Homework 2 (linear systems) is posted online.

1. Overdetermined Linear Systems
2. Sparse Matrices
3. Iterative Methods
4. Conclusions
Ax = b

The most appropriate algorithm really depends on the properties of the matrix A:

- General **dense matrices**, where the entries in $A$ are mostly non-zero and nothing special is known: Use $LU$ factorization.
- **Symmetric** ($a_{ij} = a_{ji}$) and also **positive-definite** matrices.
- General **sparse matrices**, where only a small fraction of $a_{ij} \neq 0$.
- Special **structured sparse matrices**, arising from specific physical properties of the underlying system.

It is also important to consider **how many times** a linear system with the same or related matrix or right hand side needs to be solved.
Overdetermined Linear Systems

Non-Square Matrices

- In the case of **over-determined** (more equations than unknowns) or **under-determined** (more unknowns than equations), the solution to linear systems in general becomes **non-unique**.
- One must first define what is meant by a solution, and the common definition is to use a **least-squares formulation**:

\[
x^* = \arg \min_{x \in \mathbb{R}^n} \|Ax - b\| = \arg \min_{x \in \mathbb{R}^n} \Phi(x)
\]

where the choice of the $L_2$ norm leads to:

\[
\Phi(x) = (Ax - b)^T (Ax - b).
\]

- Over-determined systems, $m > n$, can be thought of as **fitting a linear model (linear regression)**:
  The unknowns $x$ are the coefficients in the fit, the input data is in $A$ (one column per measurement), and the output data (observables) are in $b$. 

A. Donev (Courant Institute) Lecture III
It can be shown that the least-squares solution satisfies:

\[ \nabla \Phi(x) = A^T [2 (Ax - b)] = 0 \] (critical point)

This gives the square linear system of normal equations

\[ (A^T A) x^* = A^T b. \]

If \( A \) is of full rank, \( \text{rank}(A) = n \), it can be shown that \( A^T A \) is positive definite, and Cholesky factorization can be used to solve the normal equations.

Multiplying \( A^T \) \((n \times m)\) and \( A \) \((m \times n)\) takes \( n^2 \) dot-products of length \( m \), so \( O(mn^2) \) operations.
Overdetermined Linear Systems

Problems with the normal equations

\[(A^T A) x^* = A^T b.\]

- The conditioning number of the normal equations is

\[\kappa (A^T A) = [\kappa (A)]^2\]

- Furthermore, roundoff can cause \(A^T A\) to no longer appear as positive-definite and the Cholesky factorization will fail.

- If the normal equations are ill-conditioned, another approach is needed.
The QR factorization

- For nonsquare or ill-conditioned matrices of full-rank \( r = n \leq m \), the \( LU \) factorization can be replaced by the \( QR \) factorization:
  \[
  A = QR \quad [m \times n] = [m \times n][n \times n]
  \]
  where \( Q \) has orthogonal columns, \( Q^TQ = I_n \), and \( R \) is a non-singular upper triangular matrix.

- Observe that orthogonal / unitary matrices are well-conditioned \((\kappa_2 = 1)\), so the \( QR \) factorization is numerically better (but also more expensive!) than the \( LU \) factorization.

- For matrices not of full rank there are modified \( QR \) factorizations but the SVD decomposition is better (next class).

- In MATLAB, the \( QR \) factorization can be computed using \( qr \) (with column pivoting).
Solving Linear Systems via $QR$ factorization

$$(A^TA)x^* = A^Tb \text{ where } A = QR$$

- Observe that $R$ is the Cholesky factor of the matrix in the normal equations:
  
  $$A^TA = R^T(Q^TQ)R = R^TR$$

  $$(R^TR)x^* = (R^TQ^T)b \implies x^* = R^{-1}(Q^Tb)$$

  which amounts to solving a triangular system with matrix $R$.

- This calculation turns out to be much more numerically stable against roundoff than forming the normal equations (and has similar cost).
Sometimes the solution to the least-squares is still **not unique**:

- **Under-determined** systems (not enough equations to fix all unknowns)
- Singular systems, i.e., \( A \) that is **not of full rank** (use SVD):
  Any solution to \( Ax_0 = 0 \) can be added to \( x \) without changing the left hand side!

Additional condition: Choose the \( x^* \) that has **minimal Euclidean norm**, i.e., use a least-squares definition:

\[
\min_{Ax=b} \|x\|_2,
\]

although more recently of great importance are solutions that minimize the \( L_1 \) norm (**compressed sensing**).

For **under-determined full-rank** systems, \( r = m \leq n \), one does a \( QR \) factorization of \( A^T = \tilde{Q}\tilde{R} \) and the least-squares solution is

\[
x^* = \tilde{Q} \left( \tilde{R}^{-T} b \right)
\]

**Practice:** Derive the above formula.
Assume that
\[ \exists x \text{ s.t. } b = Ax, \text{ that is, } b \in \text{range}(A) \]

\[ b = Q(Rx) = Qy \quad \Rightarrow \quad x = R^{-1}y \]
showing that the columns of \( Q \) form an \textbf{orthonormal basis} for the range of \( A \) (linear subspace spanned by the columns of \( A \)).

The \( QR \) factorization is thus closely-related to the \textbf{orthogonalization} of a set of \( n \) vectors (columns) \( \{a_1, a_2, \ldots, a_n\} \) in \( \mathbb{R}^m \).

Classical approach is the \textbf{Gram-Schmidt method}: To make a vector \( b \) orthogonal to \( a \) do:

\[ \tilde{b} = b - (b \cdot a) \frac{a}{(a \cdot a)} \]

Practice: Verify that \( \tilde{b} \cdot a = 0 \)

Repeat this in sequence: Start with \( \tilde{a}_1 = a_1 \), then make \( \tilde{a}_2 \) orthogonal to \( \tilde{a}_1 = a_1 \), then make \( \tilde{a}_3 \) orthogonal to \( \text{span}(\tilde{a}_1, \tilde{a}_2) = \text{span}(a_1, a_2) \).
More efficient formula (standard Gram-Schmidt):

\[
\tilde{a}_{k+1} = a_{k+1} - \sum_{j=1}^{k} (a_{k+1} \cdot q_j) q_j, \quad q_{k+1} = \frac{\tilde{a}_{k+1}}{\|\tilde{a}_{k+1}\|},
\]

with cost \(\sim mn^2\) FLOPS.

A mathematically-equivalent but numerically much superior against roundoff error is the modified Gram-Schmidt, in which each orthogonalization is carried in sequence and repeated against each of the already-computed basis vectors:

Start with \(\tilde{a}_1 = a_1\), then make \(\tilde{a}_2\) orthogonal to \(\tilde{a}_1\), then make \(\tilde{a}_3\) orthogonal to \(\tilde{a}_1\) and then make the result orthogonal to \(\tilde{a}_2\).

The modified procedure is twice more expensive, \(\sim 2mn^2\) FLOPS, but usually worth it.

Pivoting is strictly necessary for matrices not of full rank but it can also improve stability in general.
A matrix where a substantial fraction of the entries are zero is called a **sparse matrix**. The difference with **dense matrices** is that only the nonzero entries are stored in computer memory.

Exploiting sparsity is important for **large matrices** (what is large depends on the computer).

The structure of a sparse matrix refers to the set of indices $i, j$ such that $a_{ij} > 0$, and is visualized in MATLAB using `spy`.

The structure of sparse matrices comes from the nature of the problem, e.g., in an inter-city road transportation problem it corresponds to the pairs of cities connected by a road.

In fact, just counting the number of nonzero elements is not enough: the **sparsity structure** is the most important property that determines the best method.
Banded Matrices

- **Banded matrices** are a very special but common type of sparse matrix, e.g., tridiagonal matrices

\[
\begin{bmatrix}
  a_1 & c_1 & 0 \\
  b_2 & a_2 & \ddots \\
  \ddots & \ddots & c_{n-1} \\
  0 & b_n & a_n
\end{bmatrix}
\]

- There exist special techniques for banded matrices that are much faster than the general case, e.g., only $8n$ FLOPS and no additional memory for tridiagonal matrices.

- A general matrix should be considered sparse if it has sufficiently many zeros that exploiting that fact is advantageous: usually only the case for **large matrices** (what is large?)!
Sparse Matrices

\[ A = \begin{bmatrix}
0 & 0 & 1 & 0 & 0 \\
0 & 2 & 0 & 0 & 0 \\
0 & 0 & 4 & 0 & 0 \\
0 & 0 & 0 & 5 & 0 \\
0 & 0 & 0 & 0 & 4
\end{bmatrix} \]

Sparse matrix

Directed Graph representation:

- Nodes are variables (vertices) of equations
- Arcs (edges) are the non-zero

Undirected Graph for symmetric matrices
There are general techniques for dealing with sparse matrices such as \textbf{sparse LU factorization}. How well they work depends on the structure of the matrix.

When factorizing sparse matrices, the factors, e.g., $L$ and $U$, can be much less sparse than $A$: \textbf{fill-in}.

Pivoting (\textbf{reordering} of variables and equations) has a dual, sometimes conflicting goal:

1. Reduce fill-in, i.e., \textbf{improve memory use}.
2. Reduce roundoff error, i.e., \textbf{improve stability}. Typically some \textbf{threshold pivoting} is used only when needed.

For many sparse matrices there is a large fill-in and \textbf{iterative methods} are required.
Sparse Matrices

Sparse matrices in MATLAB

\[
\begin{align*}
\text{>> } & \quad \text{A} = \text{sparse( } [1 \ 2 \ 2 \ 4 \ 4] , \ [3 \ 1 \ 4 \ 2 \ 3] , \ 1:5 \ ) \\
\text{A} &= \begin{pmatrix}
(2,1) & 2 \\
(4,2) & 4 \\
(1,3) & 1 \\
(4,3) & 5 \\
(2,4) & 3 \\
\end{pmatrix} \\
\text{>> } & \quad \text{nnz(A)} \\
\text{ans} &= 5 \\
\text{>> } & \quad \text{whos A} \\
\text{A} & 4x4 120\text{ double sparse}
\end{align*}
\]

\[
\begin{align*}
\text{>> } & \quad \text{A} = \text{sparse( } [], [], [], 4, 4, 5 \ ) \; \text{\% Pre-allocate memory} \\
\text{>> } & \quad \text{A(2,1)=2; A(4,2)=4; A(1,3)=1; A(4,3)=5; A(2,4)=3;}
\end{align*}
\]
Sparse matrices

```matlab
>> B = sprand(4,4,0.25); % Density of 25%
>> full(B)
ans =
    0     0     0   0.7655
    0   0.7952     0     0
    0   0.1869     0     0
0.4898     0     0     0
```

```matlab
>> B = sprand(100,100,0.1); spy(B)
>> X = gallery('poisson',10); spy(X)
>> [L,U,P] = lu(B); spy(L)
>> p = symrcm(B); % Symmetric Reverse Cuthill–McKee ordering
>> PBP = B(p,p); spy(PBP);
>> [L,U,P] = lu(PBP); spy(L);
```
Sparse Matrices

Random matrix $\mathbf{B}$ and structured matrix $\mathbf{X}$

The MATLAB function $spy$ shows where the nonzeros are as a plot.
Fill-in (generation of lots of nonzeros) is large for a random sparse matrix
Fill-in is much smaller for the sparse matrix but still non-negligible.
Matrix reordering cannot do much for the random matrix $B$, but it can help for structured ones!
Reducing fill-in by reordering \textbf{X}

Fill-in was reduced by about 20% (from 1000 nonzeros to 800) by the reordering for the structured \textbf{X}, but does not help much for \textbf{B}. The actual numbers are different for different classes of matrices!
Important to remember: While there are general techniques for dealing with sparse matrices that help greatly, it all depends on the structure (origin) of the matrix.

Pivoting has a dual, sometimes conflicting goal:

1. Reduce fill-in, i.e., **improve memory use**: Still active subject of research!
2. Reduce roundoff error, i.e., **improve stability**. Typically some **threshold pivoting** is used only when needed.

Pivoting for symmetric non-positive definite matrices is trickier: One can permute the diagonal entries only to **preserve symmetry**, but small diagonal entries require special treatment.

For many sparse matrices **iterative methods** (briefly covered next lecture) are required to large fill-in.
Why iterative methods?

- Direct solvers are great for dense matrices and can be made to avoid roundoff errors to a large degree. They can also be implemented very well on modern machines.
- **Fill-in** is a major problem for certain sparse matrices and leads to extreme memory requirements (e.g., three-d).
- Some matrices appearing in practice are too large to even be represented explicitly (e.g., the Google matrix).
- Often linear systems only need to be solved approximately, for example, the linear system itself may be a linear approximation to a nonlinear problem.
- Direct solvers are much harder to implement and use on (massively) parallel computers.
In iterative methods the core computation is iterative matrix-vector multiplication starting from an initial guess $x^{(0)}$.

Prototype is the linear recursion:

$$x^{(k+1)} = Bx^{(k)} + f,$$

where $B$ is an iteration matrix somehow related to $A$.

For this method to be consistent, we must have that the actual solution $x = A^{-1}b$ is a stationary point of the iteration:

$$x = Bx + f \implies A^{-1}b = BA^{-1}b + f$$

$$f = A^{-1}b - BA^{-1}b = (I - B)x$$

For this method to be stable, and thus convergent, the error $e^{(k)} = x^{(k)} - x$ must decrease:

$$e^{(k+1)} = x^{(k+1)} - x = Bx^{(k)} + f - x = B(x + e^{(k)}) + (I - B)x - x = Be^{(k)}$$
Convergence of simple iterative methods

- We saw that the error propagates from iteration to iteration as
  \[ e^{(k)} = B^k e^{(0)}. \]

- When does this converge? Taking norms,
  \[ \|e^{(k)}\| \leq \|B\|^k \|e^{(0)}\| \]
  which means that \( \|B\| < 1 \) is a **sufficient condition** for convergence.

- More precisely, \( \lim_{k \to \infty} e^{(k)} = 0 \) for any \( e^{(0)} \) iff \( B^k \to 0 \).

- Theorem: The method converges iff the **spectral radius** of the iteration matrix is less than unity:
  \[ \rho(B) < 1. \]
The spectral radius $\rho(A)$ of a matrix $A$ can be thought of as the smallest consistent matrix norm

$$\rho(A) = \max_{\lambda} |\lambda| \leq \|A\|$$

The spectral radius often determines convergence of iterative schemes for linear systems and eigenvalues and even methods for solving PDEs because it estimates the asymptotic rate of error propagation:

$$\rho(A) = \lim_{k \to \infty} \|A^k\|^{1/k}$$
The iterations of an iterative method can be terminated when:

1. The residual becomes small,
   \[ \| r^{(k)} \| \leq \varepsilon \| b \| \]
   This is good for well-conditioned systems.

2. The solution \( x^{(k)} \) stops changing, i.e., the increment becomes small,
   \[ [1 - \rho(B)] \| e^{(k)} \| \leq \| x^{(k+1)} - x^{(k)} \| \leq \varepsilon \| b \| , \]
   which can be seen to be good if convergence is rapid, \( \rho(B) \ll 1 \).

Usually a careful combination of the two strategies is employed along with some safeguards.
Iterative Methods

Fixed-Point Iteration

- A naive but often successful method for solving
  \[ x = f(x) \]
  is the **fixed-point iteration**
  \[ x_{n+1} = f(x_n). \]
- In the case of a linear system, consider rewriting \( Ax = b \) as:
  \[ x = (I - A)x + b \]
- Fixed-point iteration gives the consistent iterative method
  \[ x^{(k+1)} = (I - A)x^{(k)} + b \]
The above method is consistent but it may not converge or may converge very slowly
\[ x^{(k+1)} = (I - A) x^{(k)} + b. \]

As a way to speed it up, consider having a good approximate solver \( P^{-1} \approx A^{-1} \)

called the preconditioner \((P \text{ is the preconditioning matrix}), \), and transform
\[ P^{-1}Ax = P^{-1}b \]

Now apply fixed-point iteration to this modified system:
\[ x^{(k+1)} = (I - P^{-1}A) x^{(k)} + P^{-1}b, \]

which now has an iteration matrix \( I - P^{-1}A \approx 0 \), which means more rapid convergence.
Preconditioned Iteration

\[ x^{(k+1)} = (I - P^{-1}A) x^{(k)} + P^{-1}b \]

- In practice, we solve linear systems with the matrix \( P \) instead of inverting it:

\[ Px^{(k+1)} = (P - A) x^{(k)} + b = Px^{(k)} + r^{(k)}, \]

where \( r^{(k)} = b - Ax^{(k)} \) is the residual vector.

- Finally, we obtain the usual form of a preconditioned stationary iterative solver

\[ x^{(k+1)} = x^{(k)} + P^{-1}r^{(k)}. \]

- Note that convergence will be faster if we have a good initial guess \( x^{(0)} \).
Some Standard Examples

Splitting: \( A = L_A + U_A + D \)

- Since diagonal systems are trivial to solve, we can use the Jacobi method
  \[ P = D. \]

- Or since triangular systems are easy to solve by forward/backward substitution, we can use Gauss-Seidel method
  \[ P = L_A + D. \]

- Both of these converge for strictly diagonally-dominant matrices.
- Gauss-Seidel converges for positive-definite matrices (maybe slowly though!).
A Good Preconditioner

- Note that the matrix $A$ is only used when calculating the residual through the matrix-vector product $Ax^{(k)}$.
- We must be able to do a direct linear solver for the preconditioner $P(\Delta x) = r^{(k)}$,

so it must be in some sense simpler to deal with than $A$.
- Preconditioning is all about a balance between fewer iterations to convergence and larger cost per iteration.
- Making good preconditioners is in many ways an art and very problem-specific:
  The goal is to make $P^{-1}A$ as close to being a normal (diagonalizable) matrix with clustered eigenvalues as possible.
Some general preconditioning strategies have been designed, for example, **incomplete LU factorization** (MATLAB’s `cholinc`).

There are many **more-sophisticated iterative methods** (non-stationary, higher-order, etc) but most have the same **basic structure**:
At each iteration, solve a preconditioning linear system, do a matrix-vector calculation, and a convergence test.

For positive-(semi)definite matrices the **Preconditioned Conjugate Gradient** method is good (MATLAB’s `pcg`).

For certain types of matrices specialized methods have been designed, such as **multigrid methods** for linear systems on large grids (PDE solvers in Numerical Methods II).
MATLAB recognizes (i.e., tests for) some special matrices automatically: banded, permuted lower/upper triangular, symmetric, Hessenberg, but not sparse.

In MATLAB one may specify a matrix \( B \) instead of a single right-hand side vector \( b \).

The MATLAB function

\[
X = \text{linsolve}(A, B, \text{opts})
\]

allows one to specify certain properties that speed up the solution (triangular, upper Hessenberg, symmetric, positive definite, none), and also estimates the condition number along the way.

Use \text{linsolve} instead of backslash if you know (for sure!) something about your matrix.
The QR factorization is a numerically-stable method for solving full-rank non-square systems.

For rank-deficient matrices the singular value decomposition (SVD) is best, discussed in later lectures.

Sparse matrices deserve special treatment but the details depend on the specific field of application.

In particular, special sparse matrix reordering methods or iterative systems are often required.

When sparse direct methods fail due to memory or other requirements, iterative methods are used instead.

Convergence of iterative methods depends strongly on the matrix, and a good preconditioner is often required.

There are good libraries for iterative methods as well (but you must supply your own preconditioner!).