Numerical Methods I
Non-Square and Sparse Linear Systems

Aleksandar Donev
Courant Institute, NYU[1]
donev@courant.nyu.edu

[1]MATH-GA 2011.003 / CSCI-GA 2945.003, Fall 2014

September 25th, 2014

Outline

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

Numerical Solution of Linear Systems

$$\mathbf{Ax} = \mathbf{b}$$

- 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.

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**:

$$\mathbf{x}^\star = \arg\min_{\mathbf{x}\in\mathbb{R}^n} \|\mathbf{A}\mathbf{x} - \mathbf{b}\| = \arg\min_{\mathbf{x}\in\mathbb{R}^n} \Phi(\mathbf{x})$$

  where the choice of the L_2 norm leads to:

$$\Phi(\mathbf{x}) = (\mathbf{A}\mathbf{x} - \mathbf{b})^T (\mathbf{A}\mathbf{x} - \mathbf{b}) .$$

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

Normal Equations

- It can be shown that the least-squares solution satisfies:

$$\nabla \Phi(\mathbf{x}) = \mathbf{A}^T \left[2\left(\mathbf{A}\mathbf{x} - \mathbf{b}\right)\right] = \mathbf{0} \text{ (critical point)}$$

- This gives the square linear system of **normal equations**

$$\left(\mathbf{A}^T\mathbf{A}\right)\mathbf{x}^\star = \mathbf{A}^T\mathbf{b}.$$

- If $\mathbf{A}$ is of full rank, $\text{rank}\left(\mathbf{A}\right) = n$, it can be shown that $\mathbf{A}^T\mathbf{A}$ is positive definite, and Cholesky factorization can be used to solve the normal equations.
- Multiplying $\mathbf{A}^T$ $(n \times m)$ and $\mathbf{A}$ $(m \times n)$ takes n^2 dot-products of length m, so $O(mn^2)$ operations

Problems with the normal equations

$$\left(\mathbf{A}^T\mathbf{A}\right)\mathbf{x}^\star = \mathbf{A}^T\mathbf{b}.$$

- The conditioning number of the normal equations is

$$\kappa\left(\mathbf{A}^T\mathbf{A}\right) = \left[\kappa(\mathbf{A})\right]^2$$

- Furthermore, roundoff can cause $\mathbf{A}^T\mathbf{A}$ 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:

$$\mathbf{A} = \mathbf{QR}$$
$$[m \times n] = [m \times n][n \times n]$$

  where $\mathbf{Q}$ has **orthogonal columns**, $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}_n$, and $\mathbf{R}$ 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

$$\left(\mathbf{A}^T\mathbf{A}\right)\mathbf{x}^\star = \mathbf{A}^T\mathbf{b} \text{ where } \mathbf{A} = \mathbf{QR}$$

- Observe that $\mathbf{R}$ is the Cholesky factor of the matrix in the normal equations:

$$\mathbf{A}^T\mathbf{A} = \mathbf{R}^T\left(\mathbf{Q}^T\mathbf{Q}\right)\mathbf{R} = \mathbf{R}^T\mathbf{R}$$

$$\left(\mathbf{R}^T\mathbf{R}\right)\mathbf{x}^\star = \left(\mathbf{R}^T\mathbf{Q}^T\right)\mathbf{b} \quad \Rightarrow \quad \mathbf{x}^\star = \mathbf{R}^{-1}\left(\mathbf{Q}^T\mathbf{b}\right)$$

  which amounts to solving a triangular system with matrix $\mathbf{R}$.
- This calculation turns out to be much **more numerically stable** against roundoff than forming the normal equations (and has similar cost).

Undetermined Linear Systems

- 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 $\mathbf{A}\mathbf{x}_0 = \mathbf{0}$ can be added to $\mathbf{x}$ without changing the left hand side!
- Additional condition: Choose the $\mathbf{x}^\star$ that has **minimal Euclidean norm**, i.e., use a least-squares definition:

$$\min_{\mathbf{A}\mathbf{x}=\mathbf{b}} \|\mathbf{x}\|_2 ,$$

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 $\mathbf{A}^T = \tilde{\mathbf{Q}}\tilde{\mathbf{R}}$ and the least-squares solution is

$$\mathbf{x}^\star = \tilde{\mathbf{Q}} \left(\tilde{\mathbf{R}}^{-T} \mathbf{b} \right)$$

Practice: Derive the above formula.

Computing the QR Factorization

- Assume that

$$\exists \mathbf{x} \text{ s.t. } \mathbf{b} = \mathbf{A}\mathbf{x}, \text{ that is,} \mathbf{b} \in \text{range}(\mathbf{A})$$

$$\mathbf{b} = \mathbf{Q}(\mathbf{R}\mathbf{x}) = \mathbf{Q}\mathbf{y} \quad \Rightarrow \quad \mathbf{x} = \mathbf{R}^{-1}\mathbf{y}$$

  showing that the columns of $\mathbf{Q}$ form an **orthonormal basis** for the range of $\mathbf{A}$ (linear subspace spanned by the columns of $\mathbf{A}$).
- The QR factorization is thus closely-related to the **orthogonalization** of a set of n vectors (columns) $\{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n\}$ in $\mathbb{R}^m$.
- Classical approach is the **Gram-Schmidt method**: To make a vector $\mathbf{b}$ orthogonal to $\mathbf{a}$ do:

$$\tilde{\mathbf{b}} = \mathbf{b} - (\mathbf{b} \cdot \mathbf{a}) \frac{\mathbf{a}}{(\mathbf{a} \cdot \mathbf{a})}$$

  Practice: Verify that $\tilde{\mathbf{b}} \cdot \mathbf{a} = \mathbf{0}$
- Repeat this in sequence: Start with $\tilde{\mathbf{a}}_1 = \mathbf{a}_1$, then make $\tilde{\mathbf{a}}_2$ orthogonal to $\tilde{\mathbf{a}}_1 = \mathbf{a}_1$, then make $\tilde{\mathbf{a}}_3$ orthogonal to $\text{span}(\tilde{\mathbf{a}}_1, \tilde{\mathbf{a}}_2) = \text{span}(\mathbf{a}_1, \mathbf{a}_2)$.

Modified Gram-Schmidt Orthogonalization

- More efficient formula (**standard Gram-Schmidt**):

$$\tilde{\mathbf{a}}_{k+1} = \mathbf{a}_{k+1} - \sum_{j=1}^{k} \left(\mathbf{a}_{k+1} \cdot \mathbf{q}_j\right) \mathbf{q}_j, \quad \mathbf{q}_{k+1} = \frac{\tilde{\mathbf{a}}_{k+1}}{\|\tilde{\mathbf{a}}_{k+1}\|},$$

  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 $\tilde{\mathbf{a}}_1 = \mathbf{a}_1$, then make $\tilde{\mathbf{a}}_2$ orthogonal to $\tilde{\mathbf{a}}_1$, then make $\tilde{\mathbf{a}}_3$ orthogonal to $\tilde{\mathbf{a}}_1$ and **then make the result orthogonal to** $\tilde{\mathbf{a}}_2$.
- 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.

Sparse Matrices

- 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**

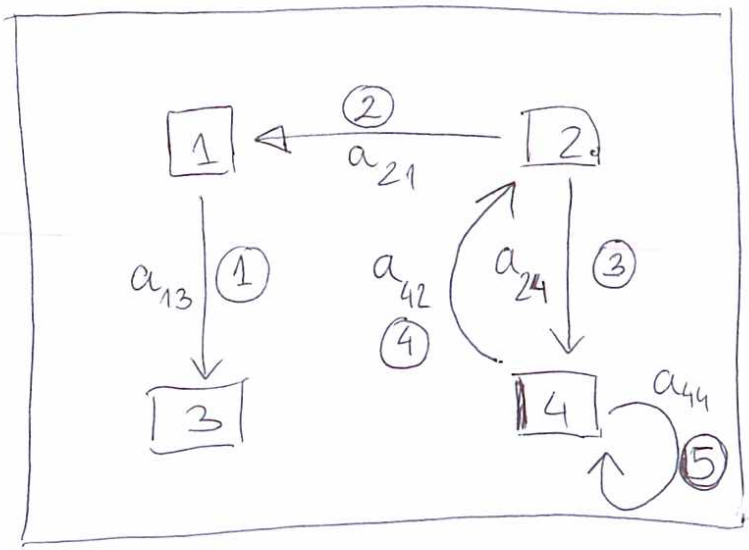
$$\begin{bmatrix} a_1 & c_1 & & \mathbf{0} \\ b_2 & a_2 & \ddots & \\ & \ddots & \ddots & c_{n-1} \\ \mathbf{0} & & b_n & a_n \end{bmatrix}$$

- 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

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 5 \end{bmatrix}$$

Sparse matrix



DIRECTED Graph representation:

→ NODES (VERTICES) are variables or equations

→ ARCS (EDGES) are the non-zeros

UNDIRECTED GRAPH FOR SYMMETRIC MATRICES

Fill-In

- There are general techniques for dealing with sparse matrices such as **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**: **fill-in**.
- Pivoting (**reordering** of variables and equations) has a dual, sometimes conflicting goal:
  1. Reduce fill-in, i.e., **improve memory use**.
  2. Reduce roundoff error, i.e., **improve stability**. Typically some **threshold pivoting** is used only when needed.
- For many sparse matrices there is a large fill-in and **iterative methods** are required.

Sparse matrices in MATLAB

```
>> A = sparse( [1 2 2 4 4], [3 1 4 2 3], 1:5 )
A =
   (2,1)        2
   (4,2)        4
   (1,3)        1
   (4,3)        5
   (2,4)        3
>> nnz(A)
ans =     5
>> whos A
  A          4x4                   120  double     sparse

>> A = sparse([],[],[],4,4,5); % Pre-allocate memory
>> A(2,1)=2; A(4,2)=4; A(1,3)=1; A(4,3)=5; A(2,4)=3;
```

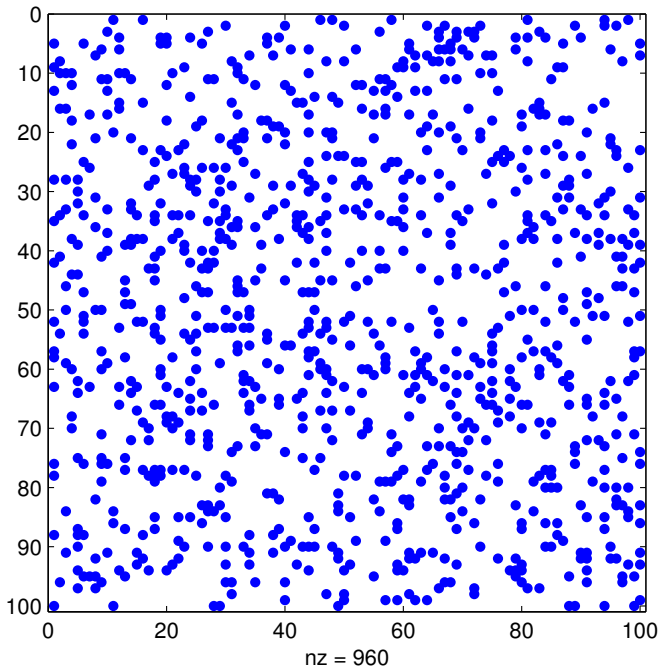
Sparse matrix factorization

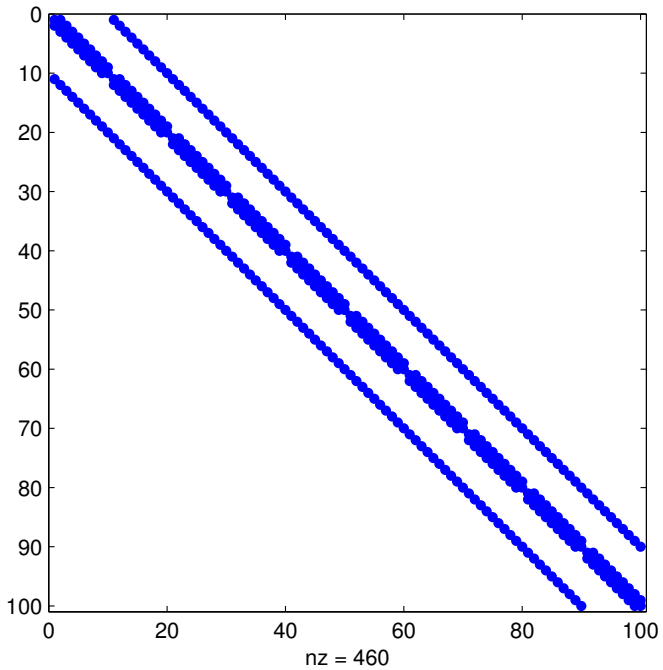
```
>> B=sprand(4,4,0.25); % Density of 25%
>> full(B)
ans =
         0         0         0    0.7655
         0    0.7952         0         0
         0    0.1869         0         0
    0.4898         0         0         0

>> 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 or
>> PBP=B(p,p); spy(PBP);
>> [L,U,P]=lu(PBP); spy(L);
```

Random matrix **B** and structured matrix **X**

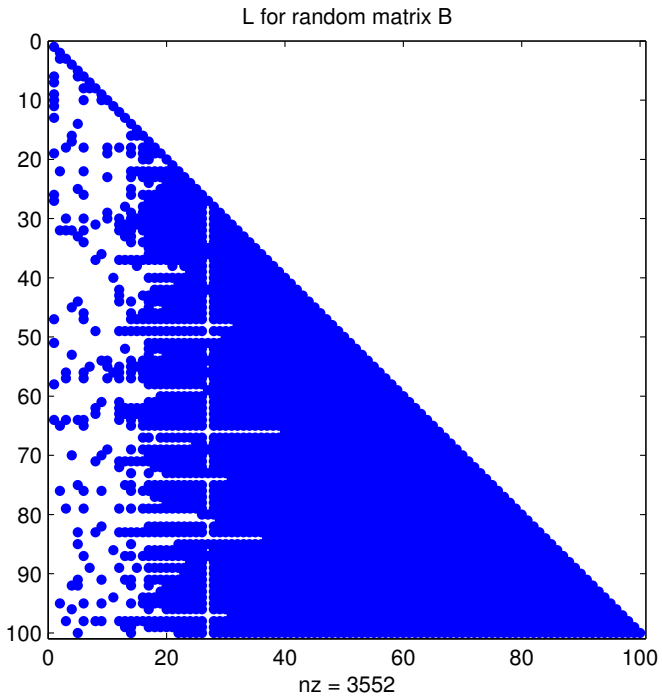
The MATLAB function *spy* shows where the nonzeros are as a plot

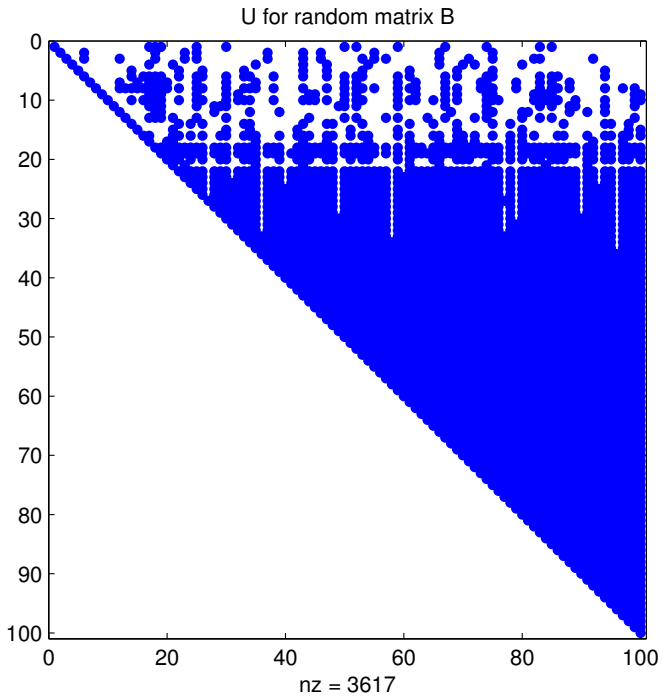




LU factors of random matrix $\mathbf{B}$

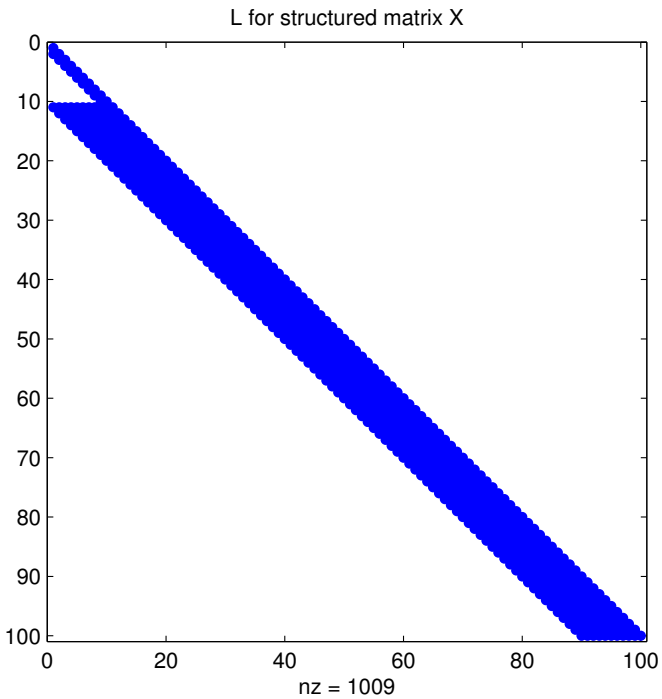
Fill-in (generation of lots of nonzeros) is large for a random sparse matrix

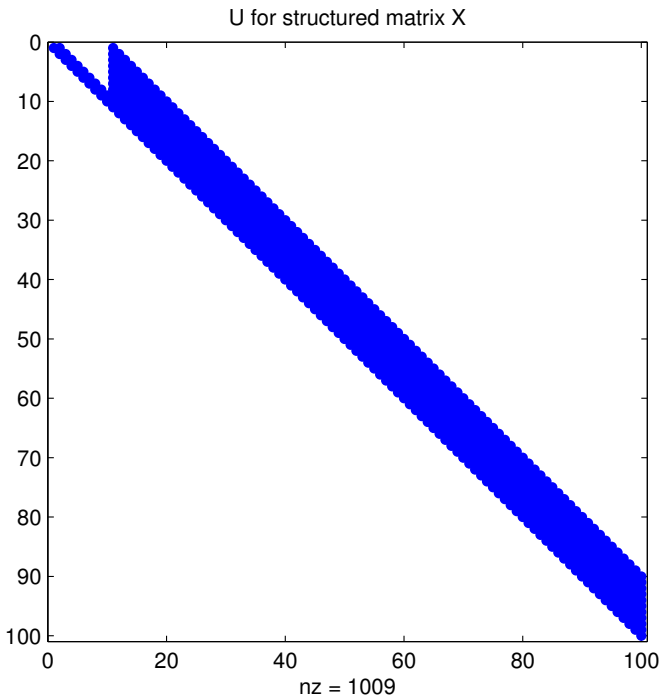




LU factors of structured matrix $\mathbf{X}$

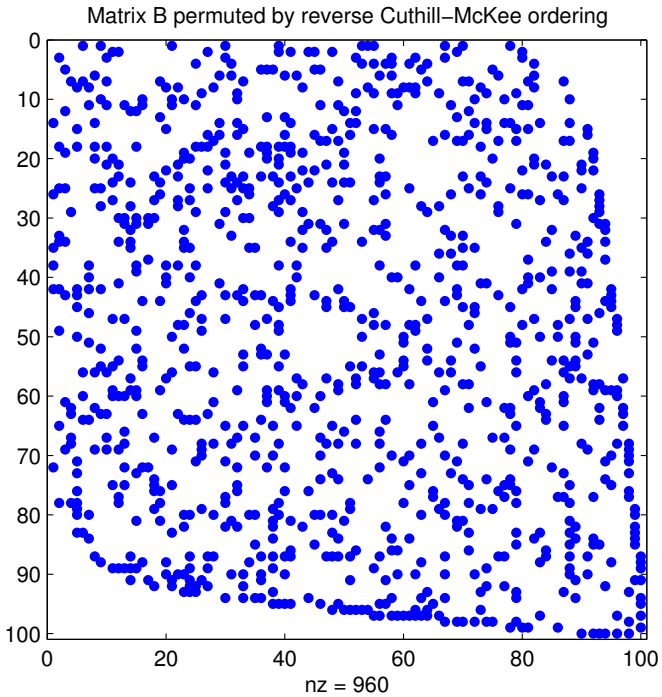
Fill-in is much smaller for the sparse matrix but still non-negligible.

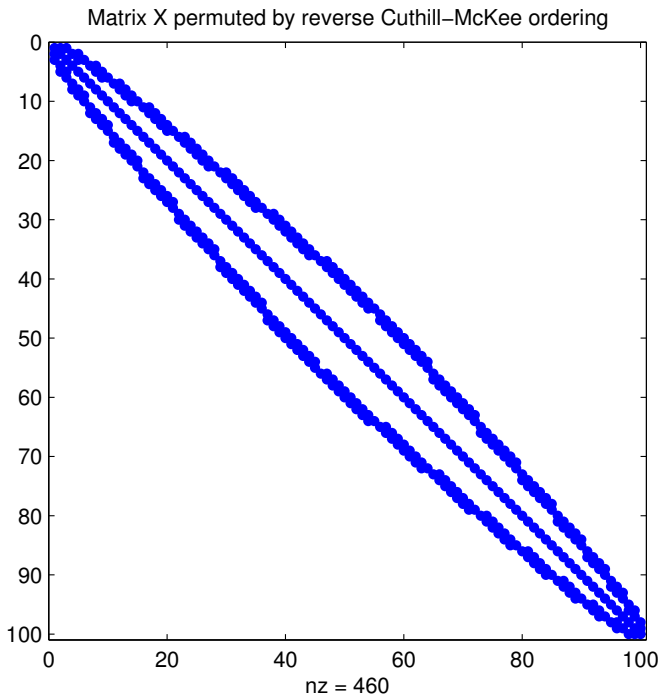




Matrix reordering

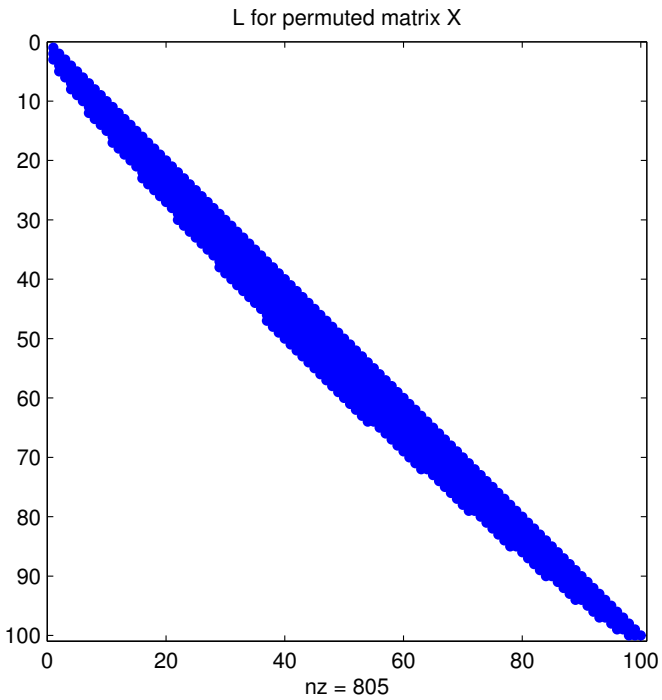
Matrix reordering cannot do much for the random matrix **B**, but it can help for structured ones!

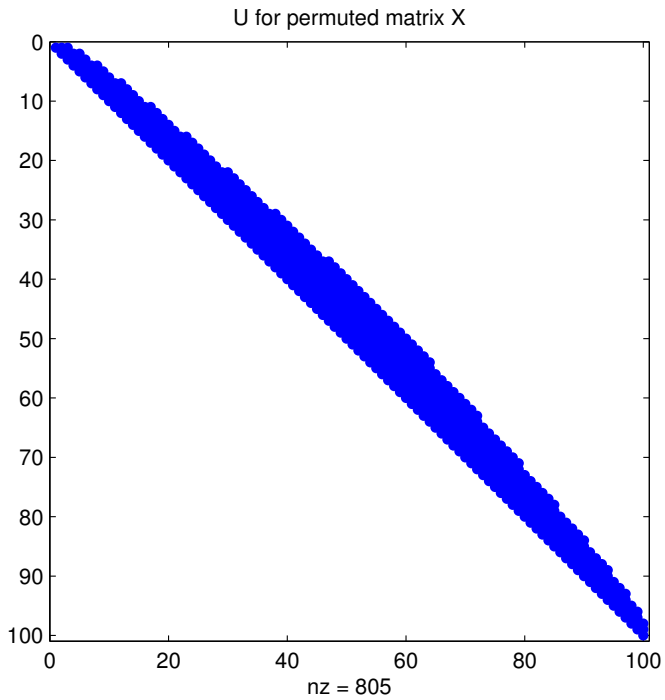




Reducing fill-in by reordering **X**

Fill-in was reduced by about 20% (from 1000 nonzeros to 800) by the reordering for the structured **X**, but does not help much for **B**.
The actual numbers are different for different classes of matrices!





Importance of Sparse Matrix Structure

- 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**.

Stationary Linear Iterative Methods of First Order

- In iterative methods the core computation is **iterative matrix-vector multiplication** starting from an **initial guess** $\mathbf{x}^{(0)}$.
- Prototype is the **linear recursion**:

$$\mathbf{x}^{(k+1)} = \mathbf{B}\mathbf{x}^{(k)} + \mathbf{f},$$

  where $\mathbf{B}$ is an **iteration matrix** somehow related to $\mathbf{A}$.
- For this method to be **consistent**, we must have that the actual solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ is a **stationary point** of the iteration:

$$\mathbf{x} = \mathbf{B}\mathbf{x} + \mathbf{f} \quad \Rightarrow \quad \mathbf{A}^{-1}\mathbf{b} = \mathbf{B}\mathbf{A}^{-1}\mathbf{b} + \mathbf{f}$$

$$\mathbf{f} = \mathbf{A}^{-1}\mathbf{b} - \mathbf{B}\mathbf{A}^{-1}\mathbf{b} = (\mathbf{I} - \mathbf{B})\,\mathbf{x}$$

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

$$\mathbf{e}^{(k+1)} = \mathbf{x}^{(k+1)} - \mathbf{x} = \mathbf{B}\mathbf{x}^{(k)} + \mathbf{f} - \mathbf{x} = \mathbf{B}\left(\mathbf{x} + \mathbf{e}^{(k)}\right) + (\mathbf{I} - \mathbf{B})\,\mathbf{x} - \mathbf{x} = \mathbf{B}\mathbf{e}^{(k)}$$

Convergence of simple iterative methods

- We saw that the error propagates from iteration to iteration as

$$\mathbf{e}^{(k)} = \mathbf{B}^k \mathbf{e}^{(0)}.$$

- When does this converge? Taking norms,

$$\left\| \mathbf{e}^{(k)} \right\| \leq \|\mathbf{B}\|^k \left\| \mathbf{e}^{(0)} \right\|$$

  which means that $\|\mathbf{B}\| < 1$ is a **sufficient condition** for convergence.
- More precisely, $\lim_{k\to\infty} \mathbf{e}^{(k)} = \mathbf{0}$ for any $\mathbf{e}^{(0)}$ iff $\mathbf{B}^k \to \mathbf{0}$.
- Theorem: The method converges iff the **spectral radius** of the iteration matrix is less than unity:

$$\rho(\mathbf{B}) < 1.$$

Spectral Radius

- The **spectral radius** $\rho(\mathbf{A})$ of a matrix $\mathbf{A}$ can be thought of as the smallest consistent matrix norm

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

- 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:

$$\rho(\mathbf{A}) = \lim_{k\to\infty} \left\|\mathbf{A}^k\right\|^{1/k}$$

Termination

- The iterations of an iterative method can be terminated when:
  1. The **residual** becomes small,

$$\left\|\mathbf{r}^{(k)}\right\| \leq \varepsilon \|\mathbf{b}\|$$

     This is good for well-conditioned systems.
  2. The solution $\mathbf{x}^{(k)}$ stops changing, i.e., the **increment** becomes small,

$$[1-\rho(\mathbf{B})]\left\|\mathbf{e}^{(k)}\right\| \leq \left\|\mathbf{x}^{(k+1)}-\mathbf{x}^{(k)}\right\| \leq \varepsilon \|\mathbf{b}\|,$$

     which can be seen to be good if convergence is rapid, $\rho(\mathbf{B}) \ll 1$.
- Usually a careful **combination** of the two strategies is employed along with some **safeguards**.

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 $\mathbf{Ax} = \mathbf{b}$ as:

$$\mathbf{x} = (\mathbf{I} - \mathbf{A})\,\mathbf{x} + \mathbf{b}$$

- Fixed-point iteration gives the consistent iterative method

$$\mathbf{x}^{(k+1)} = (\mathbf{I} - \mathbf{A})\,\mathbf{x}^{(k)} + \mathbf{b}$$

Preconditioning

- The above method is consistent but it may not converge or may converge very slowly

$$\mathbf{x}^{(k+1)} = (\mathbf{I} - \mathbf{A})\,\mathbf{x}^{(k)} + \mathbf{b}.$$

- As a way to speed it up, consider having a **good approximate solver**

$$\mathbf{P}^{-1} \approx \mathbf{A}^{-1}$$

called the **preconditioner** ($\mathbf{P}$ is the preconditioning matrix), and transform

$$\mathbf{P}^{-1}\mathbf{A}\mathbf{x} = \mathbf{P}^{-1}\mathbf{b}$$

- Now apply fixed-point iteration to this modified system:

$$\mathbf{x}^{(k+1)} = \left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right)\mathbf{x}^{(k)} + \mathbf{P}^{-1}\mathbf{b},$$

which now has an iteration matrix $\mathbf{I} - \mathbf{P}^{-1}\mathbf{A} \approx \mathbf{0}$, which means more **rapid convergence**.

Preconditioned Iteration

$$\mathbf{x}^{(k+1)} = \left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right)\mathbf{x}^{(k)} + \mathbf{P}^{-1}\mathbf{b}$$

- In practice, we solve linear systems with the matrix $\mathbf{P}$ instead of inverting it:

$$\mathbf{P}\mathbf{x}^{(k+1)} = (\mathbf{P} - \mathbf{A})\,\mathbf{x}^{(k)} + \mathbf{b} = \mathbf{P}\mathbf{x}^{(k)} + \mathbf{r}^{(k)},$$

  where $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$ is the **residual vector**.
- Finally, we obtain the usual form of a **preconditioned stationary iterative solver**

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{P}^{-1}\mathbf{r}^{(k)}.$$

- Note that convergence will be faster if we have a **good initial guess** $\mathbf{x}^{(0)}$.

Some Standard Examples

$$\text{Splitting: } \mathbf{A} = \mathbf{L}_A + \mathbf{U}_A + \mathbf{D}$$

- Since diagonal systems are trivial to solve, we can use the **Jacobi method**

$$\mathbf{P} = \mathbf{D}.$$

- Or since triangular systems are easy to solve by forward/backward substitution, we can use **Gauss-Seidel method**

$$\mathbf{P} = \mathbf{L}_A + \mathbf{D}.$$

- 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** $\mathbf{A}\mathbf{x}^{(k)}$.
- We must be able to do a **direct** linear solver for the preconditioner

$$\mathbf{P}\left(\Delta\mathbf{x}\right) = \mathbf{r}^{(k)},$$

  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 $\mathbf{P}^{-1}\mathbf{A}$ as close to being a normal (diagonalizable) matrix with **clustered eigenvalues** as possible.

In the Real World

- 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).

Special Matrices in MATLAB

- 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

$$X = linsolve(A, B, opts)$$

  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 *linsolve* instead of backslash if you know (for sure!) something about your matrix.

Conclusions/Summary

- The QR factorization is a numerically-stable method for solving **full-rank non-square systems**.
- For **rank-defficient** 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!).