Numerical Methods I: Numerical linear algebra

Georg Stadler
Courant Institute, NYU
stadler@cims.nyu.edu

September 21, 2017

Solving linear systems

We study the solution of linear systems of the form

$$A\boldsymbol{x} = \boldsymbol{b}$$

with $A \in \mathbb{R}^{n\times n}$, $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^n$. We assume that this system has a unique solution, i.e., A is invertible.

Solving linear systems is needed in many applications. Often, we have to solve

- large systems (can be up to millions of unknowns, and more)
- as fast as possible, and
- accurately and reliably.

There exist explicit formulas for solving linear systems but they are extremely expensive (e.g., Kramer's rule requires computing determinants).

Solving linear systems

Triangular systems (forward substitution):

$$\begin{bmatrix} l_{11} & 0 & \cdots & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ \vdots & & & & \vdots \\ l_{n1} & \cdots & \cdots & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{bmatrix}$$

Assume $\det(L) \neq 0$

$\det(L) = \prod_{i=1}^{n} l_{ii}$

$x_1 = b_1 / l_{11}$ 1 division

$x_2 = (b_2 - l_{21} x_1) / l_{22}$ 1 division, 1 mult, 1 addition

$\vdots$

$x_n = (b_n - l_{n1} x_1 - l_{n2} x_2 \cdots l_{n,n-1} x_{n-1}) / l_{nn}$ 1 div, $n-1$ mult, $n-1$ additions

n divisions, $\frac{n(n-1)}{2}$ multipl. $\frac{n(n-1)}{2}$ additions

overall: $\frac{n(n+1)}{2} \sim \frac{n^2}{2} = \Theta(n^2)$ additions/multiplications "flops"

Solving linear systems

Triangular systems, implementation:

$$
\begin{bmatrix}
l_{11} & 0 & \cdots & \cdots & 0 \\
l_{21} & l_{22} & 0 & \cdots & 0 \\
\vdots & & & & \vdots \\
l_{n1} & \cdots & \cdots & \cdots & l_{nn}
\end{bmatrix}
\begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ x_n \end{bmatrix}
=
\begin{bmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{bmatrix}
$$

Algorithm, row-based

x(1,1) = b(1)/L(1,1);
for i = 2:n
    x(i) = b(i) - L(i, 1:i-1) * x(1:i-1) / L(i,i)
end

Algorithm, column-based

for j = 1:n-1
    b(j) = b(j)/L(j,j);
    b(j+1:n) = b(j+1:n) - b(j) * L(j+1:n, j);
end
b(n) = b(n)/L(n,n)

→ b stores solution x.

Solving linear systems

$$\begin{bmatrix} * & & * \\ 0 & \ddots & \\ \vdots & & \ddots \\ 0 & \cdots \; 0 & x \end{bmatrix}$$

Triangular systems:

Forward and backward substitution, requires

$$\frac{n(n+1)}{2} \text{ multiplications/divisions,}$$
$$\frac{n(n-1)}{2} \text{ additions.}$$

Overall: $\sim n^2$ floating point operations (flops).

We count flops to estimate the computational time/effort. Besides floating point operations, computer memory access has a significant influence on the efficiency of numerical methods (see experiments in homework #2).

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Gaussian elimination: "new row = row i - l_{i1} row 1"

$$\begin{bmatrix} a_{11} & \cdots & \cdots & \cdots & a_{1n} \\ 0 & a'_{22} & \cdots & \cdots & a'_{2n} \\ \vdots & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ 0 & a'_{n2} & \cdots & \cdots & a'_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b'_3 \\ \vdots \\ b'_n \end{bmatrix}$$

New system matrix/rhs is: $A^{(2)} = \quad L_1 A$, $\boldsymbol{b}^{(2)} = \quad L_1 \boldsymbol{b}$.

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Gaussian elimination: "new row = row i - l_{i1} row 1"

$$\begin{bmatrix} a_{11} & \cdots & \cdots & \cdots & a_{1n} \\ 0 & a'_{22} & \cdots & \cdots & a'_{2n} \\ \vdots & 0 & a''_{33} & \cdots & a''_{3n} \\ \vdots & \vdots & & & \vdots \\ 0 & 0 & a''_{n3} & \cdots & a''_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \\ \vdots \\ b''_n \end{bmatrix}$$

New system matrix/rhs is: $A^{(3)} = L_2 L_1 A$, $\boldsymbol{b}^{(3)} = L_2 L_1 \boldsymbol{b}$.

We obtain:

$$A^{(n)} = L_{n-1} \cdots L_1 A, \quad \boldsymbol{b}^{(n)} = L_{n-1} \cdots L_1 \boldsymbol{b},$$

with the Frobenius matrices

$$L_k = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -l_{k+1,k} & 1 & & \\ & & \vdots & & \ddots & \\ & & -l_{n,k} & & & 1 \end{bmatrix}$$

Note that L_k^{-1} are also Frobenius matrices, but with different sign for the $l_{j,i}$'s.

We obtain:

lower triangular, $\begin{bmatrix} 1 & & \\ -l_{21} & \ddots & \\ -l_{n1} & \cdots & 1 \end{bmatrix}$

$$A^{(n)} = L_{n-1} \cdots L_1 A, \quad \boldsymbol{b}^{(n)} = L_{n-1} \cdots L_1 \boldsymbol{b},$$

$U = L_{n-1} \cdots L_1 \cdot A$

with the Frobenius matrices

$$L_k = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -l_{k+1,k} & 1 & & \\ & & \vdots & & \ddots & \\ & & -l_{n,k} & & & 1 \end{bmatrix}$$

$\Rightarrow A = LU$

$(L_{n-1} \cdots L_1)^{-1}$

Note that L_k^{-1} are also Frobenius matrices, but with different sign for the $l_{j,i}$'s.

Solving linear systems

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & & a_{nn} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & & 0 \\ \vdots & & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & & u_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

$$\begin{aligned} u_{11} &= a_{11} \\ u_{12} &= a_{12} \\ l_{21} u_{11} &= a_{21} \\ l_{21} u_{12} + u_{22} &= a_{22} \\ &\vdots \end{aligned}$$

Algorithm for solving linear system $A\boldsymbol{x} = \boldsymbol{b}$ (assuming diagonal elements do not vanish):

1. Compute triangular factorization $A = LU$.
2. Solve $L\boldsymbol{z} = \boldsymbol{b}$ (forward substitution).
3. Solve $U\boldsymbol{x} = \boldsymbol{z}$ (backward substitution).

$Ax = b \iff L\underbrace{Ux}_{z} = b$: Solve $Lz = b$ $\Theta(n^2)$

$Ux = z$ $\Theta(n^2)$

Algorithm for solving linear system $A\boldsymbol{x} = \boldsymbol{b}$ (assuming diagonal elements do not vanish):

1. Compute triangular factorization $A = LU$. — flops: $\sim \frac{n^3}{3}$ flops
2. Solve $L\boldsymbol{z} = \boldsymbol{b}$ (forward substitution).
3. Solve $U\boldsymbol{x} = \boldsymbol{z}$ (backward substitution).

Notes:

- Main cost is LU factorization.
- Factorization can be reused for different right hand sides $\boldsymbol{b}$.

Matlab: Solving $Ax=b$: `x=A\b;`
How about: `x=inv(A)*b;`

If diagonal "pivoting" element is zero (or very small), one has to exchange rows and/or columns–otherwise the LU factorization fails.

Basic idea:
Choose largest element (in absolute value) in the row that is eliminated as pivot.

Example: $A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \longrightarrow$ Gauss elimination fails

exchange 1^{st} & 2^{nd} row:

$\tilde{A} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = LU, \quad L = U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

Example with a 3 digit computer:

$$\begin{pmatrix} 10^{-4} & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

exact solution: $x_1 = 1.000$, $x_2 = 0.999$

Gauss elimination on machine with 3 accurate digits:

$$\left(\begin{array}{cc|c} 10^{-4} & 1 & 1 \\ 1 & 1 & 2 \end{array}\right) \longrightarrow \left(\begin{array}{cc|c} 10^{-4} & 1 & 1 \\ 0 & \underbrace{1-10^4}_{\approx -1.000 \times 10^4} & \underbrace{2-10^4}_{-1.000 \times 10^4} \end{array}\right)$$

$$\Longrightarrow \frac{x_2 = 1}{10^{-4} x_1 + 1.000 = 1.000} \Longrightarrow \underline{\underline{x_1 = 0}}$$

Solving linear systems

LU with pivoting

matrix with exactly one "1" in each row & column, zeros else

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Pivoting can be expressed by permutation matrices P_π, resulting in the LU decomposition (the permutation π also affects L and U).

Theorem: For every invertible matrix A, there exists a permutation matrix P_π such that

$$P_\pi A = LU$$

is possible. The permutation can be chosen such that all entries in L are ≤ 1.

Proof (Sketch): $\det(A) \neq 0 \rightarrow$ not all entries in first column are zero

Let's permute rows such that

$$A^{(1)} = P_{\tau_1} A, \quad |a_{11}^{(1)}| \geq |a_{j1}^{(1)}|$$

$$A^{(2)} = L_1 A^{(1)} = L_1 P_{\tau_1} A = \left[\begin{array}{c|ccc} a_{11}^{(1)} & * & \cdots & * \\ \hline 0 & & & \\ \vdots & & B^{(2)} & \\ 0 & & & \end{array}\right]$$

entries in L_1 have abs. value ≤ 1.

Solving linear systems

LU with pivoting

repeat

$$U = A^{(n)} = L_{n-1} P_{\tau_{n-1}} \cdots\cdots L_2 P_{\tau_2} L_1 P_{\tau_1} A$$

$$\hat{L}_k = P_\pi L_k P_\pi^{-1} = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & -l_{\pi(k+1),k} & \ddots & \\ & & \vdots & & \\ & & -l_{\pi(n),k} & & 1 \end{bmatrix}$$

$$U = L_{n-1} \underbrace{P_{\tau_{n-1}} L_{n-2} P_{\tau_{n-1}}^{-1}}_{\hat{L}_{n-2}} \underbrace{P_{\tau_{n-1}} P_{\tau_{n-2}} L_{n-3} P_{\tau_{n-2}}^{-1} P_{\tau_{n-1}}^{-1}}_{\hat{L}_{n-3}} P_{\tau_{n-1}} P_{\tau_{n-2}} P_{\tau_{n-3}} L$$

$$= L_{n-1} \hat{L}_{n-2} \hat{L}_{n-3} \cdots\cdots P_{\pi_0} A$$

$$\boxed{LU = P_{\pi_0} A}$$

$$L = (L_{n-1} L_{n-2} \cdots)^{-1}$$

A matrix is symmetric positive definite (spd), if $A = A^T$ and for all $\boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{x} \neq 0$, the inner product $\langle A\boldsymbol{x}, \boldsymbol{x} \rangle > 0$.
For spd matrices, we can compute the factorization:

$$A = LDL^T,$$

where L is a lower triangular matrix with 1's on the diagonal, and D is a positive diagonal matrix.
The Choleski factorization is obtained by multiplying the square root of D (which exists!) with L:

$$A = \bar{L}\bar{L}^T.$$

Choleski factorization requires $\sim \frac{n^3}{6}$ multiplications and n square roots.

Kinds of linear systems

Solvers such as MATLAB's \ take advantage of matrix properties:

- Dense matrix storage: Only entries are stored as 1D array (column or row wise)
- Sparse matrix storage: Most $a_{ij} = 0$: only store nonzero entries; stores indices and value; occur in many applications

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & & \vdots \\ \vdots & & \vdots \\ \vdots & \cdots & \cdot \end{bmatrix} \longrightarrow$$ dense just stores first column, 2nd column ... etc.

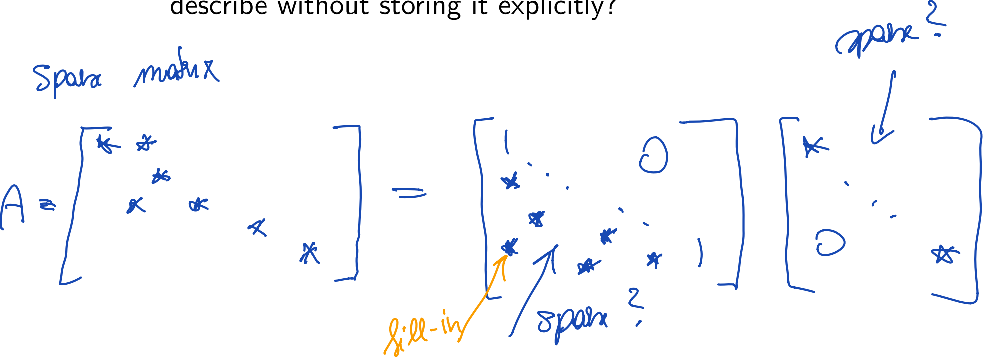
Sparse format: only store non-zeros!, need to store the value a_{ij} and i, j.

Kinds of linear systems

Solvers such as MATLAB's \ take advantage of matrix properties:

- Fast algorithms for special matrices: for computing Ax, FFT, FMM, ...
- Sparse: Most $a_{ij} = 0$: avoid fill-in in factorizations

  non-zero elements in factorization.
- Structured/unstructured: is the sparsity pattern easy to describe without storing it explicitly?



Kinds of linear systems and solvers

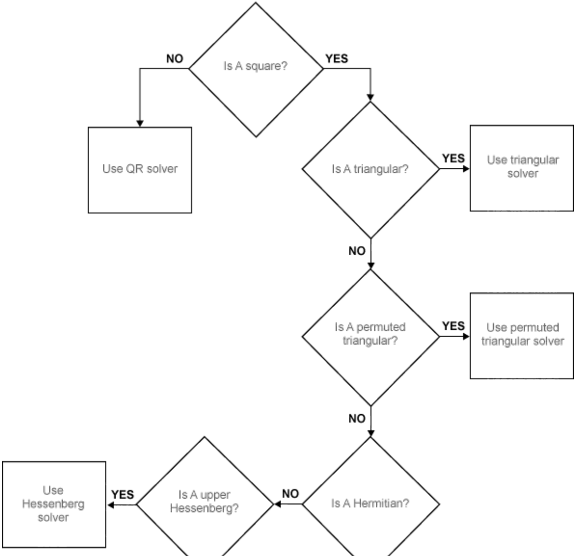
Symmetry, positivity ...

- Special factorizations for (skew) symmetric matrices
- Special factorizations for positive definite matrices (Choleski)
- Diagonally dominant matrices don't need pivoting

MATLAB's \ (i.e., UMFPACK) chooses the optimal algorithms after studying properties of the matrix (details in the "backslash" book: Tim Davis: *Direct methods for sparse linear systems*, SIAM, 2006.)

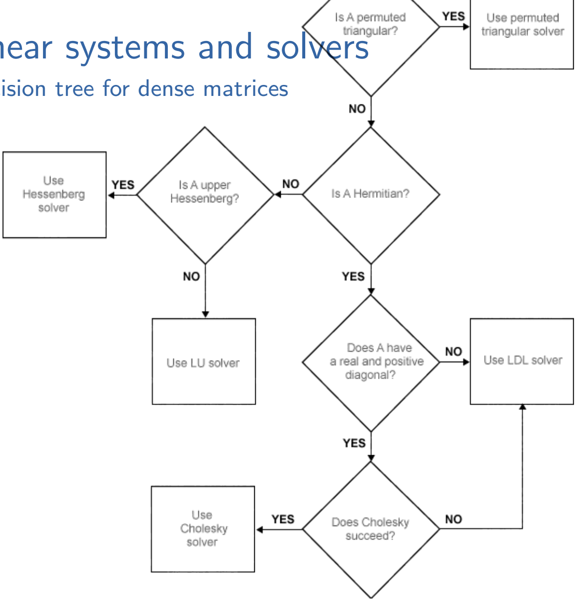
Kinds of linear systems and solvers

UMFPACK's decision tree for dense matrices



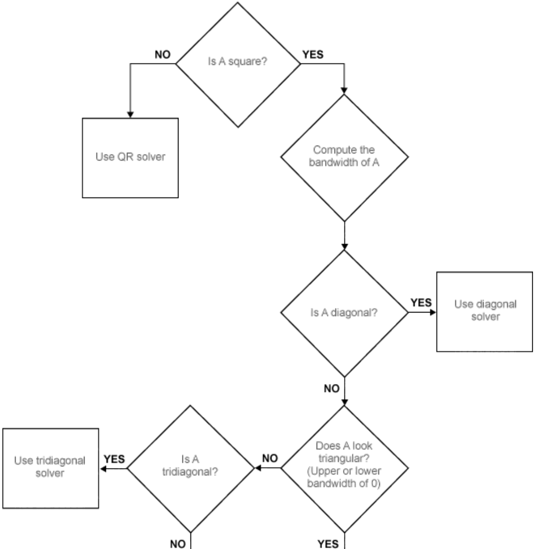
Kinds of linear systems and solvers

UMFPACK's decision tree for dense matrices



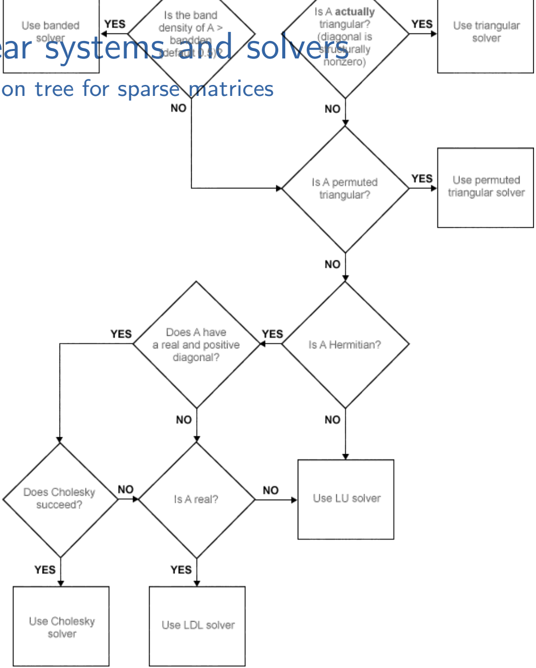
Kinds of linear systems and solvers

UMFPACK's decision tree for sparse matrices



Kinds of linear systems and solvers

UMFPACK's decision tree for sparse matrices



Kinds of linear systems and solvers

Factorization-based/direct solvers (dense/sparse LU, Choleski) require the matrix

- to fit into memory,
- to be explicitly available (sometimes only a function that applies the matrix to a vector is available) and to fit in memory,

\+ but compute exact (besides rounding error) solution

Iterative solvers

- find an ε-approximation of the solution,

\+ able to solve very large problems,

\+ often only require a function that computes Ax for given x

± might be faster or slower than a factorization-based method

MATLAB demo

- What are the different storage formats (sparse/dense)? Is it always better to use one of them?
- How long does it take to solve sparse/dense systems?
- What is fill in and how to avoid it?

Sparse/sense storage:

```
A=rand(2,2);
B=sparse(A);
whos
```

Fill-in:

```
A=bucky + 4*speye(60);
r = symrcm(A);
spy(A); spy(A(r,r)); spy(chol(A)); spy(chol(A(r,r)));
```

Which sparse solver?

```
spparms('spumoni',1);
A=gallery('poisson',8);
b=randn(64,1);
A\b;
```