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

$$Ax = b$$

with $A \in \mathbb{R}^{n \times n}$, $x, b \in \mathbb{R}^n$. 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):

\[
\begin{bmatrix}
l_{11} & 0 & \cdots & \cdots & 0 \\
l_{21} & l_{22} & 0 & \cdots & 0 \\
\vdots & & \ddots & \ddots & \vdots \\
l_{n1} & \cdots & \cdots & \cdots & l_{nn}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_n
\end{bmatrix}
= 
\begin{bmatrix}
b_1 \\
b_2 \\
\vdots \\
b_n
\end{bmatrix}
\]

Assume \( \det(L) \neq 0 \)

\[ \frac{n}{i=1} \sum \frac{x_i}{l_{ii}} \]

\[
\begin{align*}
x_1 &= \frac{b_1}{l_{11}} \\
x_2 &= \frac{b_2 - l_{21}x_1}{l_{22}} \\
&\vdots \quad \text{(division, multiplication, addition)} \\
x_n &= \frac{b_n - l_{n1}x_1 - l_{n2}x_2 - \cdots - l_{n,n-1}x_{n-1}}{l_{nn}} \\
&\text{(division, multiplication, addition)}
\end{align*}
\]

\( \frac{n(n+1)}{2} \) multiplications, \( \frac{n(n-1)}{2} \) additions

\( n \) divisions, \( \frac{n(n-1)}{2} \) multiplications, \( \frac{n(n-1)}{2} \) additions
Solving linear systems

Triangular systems, implementation:

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

Algorithm, row-based

\[
x(i) = \frac{b(i)}{L(i,i)}
\]

\textbf{for} \ i = 2:n \\
\textbf{do} \ x(i) = b(i) - L(i,1:i-1) \times x(1:i-1) / L(i,i) \\
\textbf{end}

Algorithm, column-based

\textbf{for} \ j = 1:n-1 \\
b(j) = \frac{b(j)}{L(j,j)} \times b(j+1:n) = b(j+1:n) - b(j) \times L(j+1:n, j) \\
\textbf{end}

\textbf{Solution} \ x
Solving linear systems

Triangular systems:

Forward and backward substitution, requires

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

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).
Solving linear systems

Gaussian elimination—LU factorization

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

Gaussian elimination: “new row = row $i - l_{i1}$ row 1”

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

New system matrix/rhs is: $A^{(2)} = L_1 A$, $b^{(2)} = L_1 b$. 
Solving linear systems
Gaussian elimination—LU factorization

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

Gaussian elimination: “new row = row \( i \) - \( l_{i1} \) row 1”

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

New system matrix/rhs is: \( A^{(3)} = L_2L_1A \), \( b^{(3)} = L_2L_1b \).
Solving linear systems
Gaussian elimination—LU factorization

We obtain:

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

with the Frobenius matrices

\[
L_k = \begin{bmatrix}
1 \\
\vdots \\
-1_{k+1,k} & 1 \\
\vdots \\
-1_{n,k} & 1
\end{bmatrix}
\]

Note that \( L_k^{-1} \) are also Frobenius matrices, but with different sign for the \( l_{j,i} \)'s.
Solving linear systems
Gaussian elimination—LU factorization

We obtain:

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

with the Frobenius matrices

\[ L_k = \begin{bmatrix}
1 \\
\vdots \\
-1_{k+1,k} & 1 \\
\vdots \\
-1_{n,k} & \cdots & 1
\end{bmatrix} \]

Note that \( L_k^{-1} \) are also Frobenius matrices, but with different sign for the \( l_{j,i} \)'s.
Solving linear systems

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

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

\[
\begin{align*}
  u_{11} &= a_{11} \\
  u_{12} &= a_{12} \\
  l_{21} u_{11} &= a_{21} \\
  l_{21} u_{12} + u_{22} &= a_{22} \\
  \vdots 
\end{align*}
\]
Solving linear systems
Gaussian elimination—LU factorization

Algorithm for solving linear system $Ax = b$ (assuming diagonal elements do not vanish):

1. Compute triangular factorization $A = LU$.
2. Solve $Lz = b$ (forward substitution).
3. Solve $Ux = z$ (backward substitution).

$A x = b \rightarrow L U x = b : \text{Solve } L z = b \quad \text{and } \quad U x = z \quad \Theta(n^2) \quad \Theta(n^2)$
Solving linear systems
Gaussian elimination—LU factorization

Algorithm for solving linear system $Ax = b$ (assuming diagonal elements do not vanish):

1. Compute triangular factorization $A = LU$.
2. Solve $Lz = b$ (forward substitution).
3. Solve $Ux = z$ (backward substitution).

Notes:

- **Main cost** is $LU$ factorization.
- Factorization can be reused for different right hand sides $b$.

Matlab: Solving $Ax = b$: $x = A \backslash b$.

How about: $x = \text{inv}(A) \ast b$.
Solving linear systems
LU with pivoting

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:

\[ A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \Rightarrow \text{Gauss elimination fails} \]

exchange 1st & 2nd row:

\[ \hat{A} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = LU, \quad L = U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \]
Solving linear systems
LU with pivoting

Example with a 3 digit computer:

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

Exact Solution:
\[
x_1 = 1.000 \\
x_2 = 0.999
\]

Gauss elimination on machine with 3 accurate digits:

\[
\begin{pmatrix}
10^{-4} & 1 \\
1 & 1 \\
1 & 2
\end{pmatrix}
\rightarrow
\begin{pmatrix}
10^{-4} & 1 & 1 \\
0 & 1-10^{-4} & 2-10^{-4} \\
0 & -1.000 & -1.000 	imes 10^4
\end{pmatrix}
\]

\[
x_2 = \frac{1}{10^{-4}} \\
10^{-4} x_1 + 1.000 = 1.000 \rightarrow x_1 = 0
\]
Solving linear systems
LU with pivoting

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

**Theorem:** For every invertible matrix \( A \), there exists a permutation matrix \( P_\pi \) such that

\[ P_\pi A = LU \]

is possible. The permutation can be chosen such that all entries in \( L \) are \( \leq 1 \).

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

Let's permute rows such that

\[ A^{(1)} = P_{\pi 1} A, \quad |a_{11}^{(1)}| \geq |a_{11}^{(2)}| \]

\[ A^{(2)} = L_1 A^{(1)} = L_1 P_{\pi 1} A = \begin{bmatrix} a_{11}^{(2)} & \star & \ldots & \star \\ 0 & \ddots & \ddots & \ddots \\ 0 & \ddots & \ddots & \ddots \\ \vdots & \ddots & \ddots & \ddots \\ 0 & \cdots & 0 & \ddots \\ \end{bmatrix} B^{(2)} \]

Entries in \( L_1 \) have abs. value \( \leq 1 \).
Solving linear systems

LU with pivoting

\[ U = A^{(n)} = L_{n-1} P_{c_{n-1}} \ldots L_2 P_{c_2} L_1 P_{c_1} A \]

\[ L_k = P_{Ti} L_k P_{Ti}^{-1} = \begin{bmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & \\ -l_{k+1,k} & \cdots & -l_{nk,k} & 1 \end{bmatrix} \]

\[ U = L_{n-1} P_{c_{n-1}} L_{n-2} P_{c_{n-2}} L_{n-3} P_{c_{n-3}} \ldots P_{To} A \]

\[ \text{LU} = P_{To} A \]
A matrix is **symmetric positive definite (spd)**, if \( A = A^T \) and for all \( x \in \mathbb{R}^n, x \neq 0 \), the inner product \( \langle A x, x \rangle > 0 \).

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

\[ A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \ddots & \vdots \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \rightarrow \text{dense just shows first column, 2nd column... etc.} \]

**Sparse format**: only store non-zero, 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
- **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

1. Is A square?
   NO: Use QR solver
   YES: Is A triangular?
      NO: Is A permuted triangular?
         NO: Use Hessenberg solver
         YES: Use permutated triangular solver
      YES: Use triangular solver
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

- Is A square?
  - NO: Use QR solver
  - YES: Compute the bandwidth of A

- Is A diagonal?
  - YES: Use diagonal solver
  - NO: Does A look triangular? (Upper or lower bandwidth of 0)
    - NO: Use tridiagonal solver
    - YES: Is A tridiagonal?
Kinds of linear systems and solvers

UMFPACK’s decision tree for sparse matrices

1. Use banded solver
2. Is the band density of A > band dens. def (D. B.)?
   - NO
   - YES
     - Is A actually triangular? (diagonal is artificially nonzero)
       - NO
       - YES
         - Use triangular solver
       - NO
         - Is A permuted triangular?
           - NO
             - Use LU solver
           - YES
             - Use permuted triangular solver
         - YES
           - Does A have a real and positive diagonal?
             - NO
               - Use LU solver
             - YES
               - Is A Hermitian?
                 - NO
                   - Use LU solver
                 - YES
                   - Does Cholesky succeed?
                     - NO
                       - Use LU solver
                     - YES
                       - Use Cholesky solver
   - YES
     - Use LU solver
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 $\varepsilon$-approximation of the solution,

+ able to solve very large problems,

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

\pm might be faster or slower than a factorization-based method
Kinds of linear systems and solvers

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?
Kinds of linear systems and solvers
MATLAB demo

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;