AM 205: lecture 7

- Last time: Introduction to numerical linear algebra
- Today’s lecture: LU factorization, Cholesky factorization, timing, QR factorization
- Homework 2 is now posted.¹ It is due on Wednesday October 9 at 5 PM.²

¹There’s an Easter egg in homework 2.
²AM205 homeworks are front-loaded toward the first half of the semester. Homework 2 is the longest and more time is provided than usual. Homworks 4 & 5 are shorter.
LU Factorization
Solving $Ax = b$

Familiar idea for solving $Ax = b$ is to use Gaussian elimination to transform $Ax = b$ to a triangular system.

What is a triangular system?

- Upper triangular matrix $U \in \mathbb{R}^{n \times n}$: if $i > j$ then $u_{ij} = 0$
- Lower triangular matrix $L \in \mathbb{R}^{n \times n}$: if $i < j$ then $\ell_{ij} = 0$

**Question:** Why is triangular good?

**Answer:** Because triangular systems are easy to solve!
Solving $Ax = b$

Suppose we have $Ux = b$, then we can use “back-substitution”

\[
\begin{align*}
  x_n &= b_n / u_{nn} \\
  x_{n-1} &= (b_{n-1} - u_{n-1,n}x_n) / u_{n-1,n-1} \\
  &\vdots \\
  x_j &= \left( b_j - \sum_{k=j+1}^{n} u_{jk}x_k \right) / u_{jj} \\
  &\vdots
\end{align*}
\]
Similarly, we can use forward substitution for a lower triangular system $Lx = b$

\[
\begin{align*}
x_1 &= \frac{b_1}{\ell_{11}} \\
x_2 &= \frac{(b_2 - \ell_{21}x_1)}{\ell_{22}} \\
\vdots \\
x_j &= \left( b_j - \sum_{k=1}^{j-1} \ell_{jk}x_k \right) / \ell_{jj} \\
\vdots 
\end{align*}
\]
Solving $Ax = b$

Back and forward substitution can be implemented with doubly nested for-loops

The computational work is dominated by evaluating the sum
$$\sum_{k=1}^{j-1} \ell_{jk}x_k, \ j = 1, \ldots, n$$

We have $j - 1$ additions and multiplications in this loop for each $j = 1, \ldots, n$, i.e. $2(j - 1)$ operations for each $j$

Hence the total number of floating point operations in back or forward substitution is asymptotic to:

$$2 \sum_{j=1}^{n} j = 2n(n + 1)/2 \sim n^2$$
Solving $Ax = b$

Here “∼” refers to asymptotic behavior, e.g.

$$f(n) \sim n^2 \iff \lim_{n \to \infty} \frac{f(n)}{n^2} = 1$$

We often also use “big-O” notation, e.g. for remainder terms in Taylor expansion

$$f(x) = O(g(x))$$ if there exists $M \in \mathbb{R}_{>0}, x_0 \in \mathbb{R}$ such that

$$|f(x)| \leq M|g(x)|$$ for all $x \geq x_0$

In the present context we prefer “∼” since it indicates the correct scaling of the leading-order term

e.g. let $f(n) \equiv n^2/4 + n$, then $f(n) = O(n^2)$, whereas $f(n) \sim n^2/4$
Solving $Ax = b$

So transforming $Ax = b$ to a triangular system is a sensible goal, but how do we achieve it?

**Observation**: If we premultiply $Ax = b$ by a nonsingular matrix $M$ then the new system $MAx = Mb$ has the same solution

Hence, want to devise a sequence of matrices $M_1, M_2, \cdots, M_{n-1}$ such that $MA \equiv M_{n-1} \cdots M_1 A \equiv U$ is upper triangular

This process is **Gaussian Elimination**, and gives the transformed system $Ux = Mb$
LU Factorization

We will show shortly that it turns out that if $MA = U$, then we have that $L \equiv M^{-1}$ is lower triangular

Therefore we obtain $A = LU$: product of lower and upper triangular matrices

This is the LU factorization of $A$
LU Factorization

LU factorization is the most common way of solving linear systems!

\[ Ax = b \iff LUx = b \]

Let \( y \equiv Ux \), then \( Ly = b \): solve for \( y \) via forward substitution\(^3\)

Then solve for \( Ux = y \) via back substitution

\(^3\)\( y = L^{-1}b \) is the transformed right-hand side vector (i.e. \( Mb \) from earlier) that we are familiar with from Gaussian elimination
LU Factorization

Next question: How should we determine $M_1, M_2, \cdots, M_{n-1}$?

We need to be able to annihilate selected entries of $A$, below the diagonal in order to obtain an upper-triangular matrix.

To do this, we use “elementary elimination matrices”

Let $L_j$ denote $j^{th}$ elimination matrix (we use “$L_j$” rather than “$M_j$” from now on as elimination matrices are lower triangular).
LU Factorization

Let \( X(\equiv L_{j-1}L_{j-2} \cdots L_1A) \) denote matrix at the start of step \( j \), and let \( x(:,:,j) \in \mathbb{R}^n \) denote column \( j \) of \( X \)

Then we define \( L_j \) such that

\[
L_jx(:,:,j) \equiv \begin{bmatrix}
1 & \cdots & 0 & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & \cdots & 1 & 0 & \cdots & 0 \\
0 & \cdots & -\frac{x_{j+1,j}}{x_{jj}} & 1 & \cdots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & \cdots & -\frac{x_{nj}}{x_{jj}} & 0 & \cdots & 1
\end{bmatrix}
\begin{bmatrix}
x_{1j} \\
x_{jj} \\
x_{j+1,j} \\
x_{nj} \\
\end{bmatrix} = \begin{bmatrix}
x_{1j} \\
x_{jj} \\
x_{j+1,j} \\
x_{nj} \\
\end{bmatrix}
\]
LU Factorization

To simplify notation, we let \( \ell_{ij} \equiv \frac{x_{ij}}{x_{jj}} \) in order to obtain

\[
L_j \equiv \begin{bmatrix}
1 & \cdots & 0 & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & \cdots & 1 & 0 & \cdots & 0 \\
0 & \cdots & -\ell_{j+1,j} & 1 & \cdots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & \cdots & -\ell_{nj} & 0 & \cdots & 1
\end{bmatrix}
\]
LU Factorization

Using elementary elimination matrices we can reduce $A$ to upper triangular form, one column at a time.

Schematically, for a $4 \times 4$ matrix, we have

$\begin{bmatrix}
\times & \times & \times & \times \\
\times & \times & \times & \times \\
\times & \times & \times & \times \\
\times & \times & \times & \times \\
\end{bmatrix}
\xrightarrow{L_1} 
\begin{bmatrix}
\times & \times & \times & \times \\
0 & \times & \times & \times \\
0 & \times & \times & \times \\
0 & \times & \times & \times \\
\end{bmatrix}
\xrightarrow{L_2} 
\begin{bmatrix}
\times & \times & \times & \times \\
0 & \times & \times & \times \\
0 & 0 & \times & \times \\
0 & 0 & \times & \times \\
\end{bmatrix}
$

$A \xrightarrow{L_1} L_1A \xrightarrow{L_2} L_2L_1A$

Key point: $L_k$ does not affect columns $1, 2, \ldots, k-1$ of $L_{k-1}L_{k-2}\ldots L_1A$
After $n - 1$ steps, we obtain the upper triangular matrix
$U = L_{n-1} \cdots L_2 L_1 A$

$$U = \begin{bmatrix}
\times & \times & \times & \times \\
0 & \times & \times & \times \\
0 & 0 & \times & \times \\
0 & 0 & 0 & \times 
\end{bmatrix}$$
Finally, we wish to form the factorization $A = LU$, hence we need
$L = (L_{n-1} \cdots L_2 L_1)^{-1} = L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1}$

This turns out to be surprisingly simple due to two strokes of luck!

First stroke of luck: $L_j^{-1}$ is obtained simply by negating the
subdiagonal entries of $L_j$

$$L_j \equiv \begin{bmatrix}
1 & \cdots & 0 & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & \cdots & 1 & 0 & \cdots & 0 \\
0 & \cdots & -\ell_{j+1,j} & 1 & \cdots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & \cdots & -\ell_{nj} & 0 & \cdots & 1
\end{bmatrix}, \quad L_j^{-1} \equiv \begin{bmatrix}
1 & \cdots & 0 & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & \cdots & 1 & 0 & \cdots & 0 \\
0 & \cdots & \ell_{j+1,j} & 1 & \cdots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & \cdots & \ell_{nj} & 0 & \cdots & 1
\end{bmatrix}$$
**LU Factorization**

**Explanation:** Let $\ell_j \equiv [0, \ldots, 0, \ell_{j+1}, \ldots, \ell_{nj}]^T$ so that

$$L_j = I - \ell_j e_j^T$$

Now consider $L_j(I + \ell_j e_j^T)$:

$$L_j(I + \ell_j e_j^T) = (I - \ell_j e_j^T)(I + \ell_j e_j^T) = I - \ell_j e_j^T \ell_j e_j^T = I - \ell_j (e_j^T \ell_j) e_j^T$$

Also, $(e_j^T \ell_j) = 0$ (why?) so that $L_j(I + \ell_j e_j^T) = I$

By the same argument $(I + \ell_j e_j^T)L_j = I$, and hence

$$L_j^{-1} = (I + \ell_j e_j^T)$$
LU Factorization

Next we want to form the matrix $L \equiv L_1^{-1}L_2^{-1} \cdots L_{n-1}^{-1}$

Note that we have

$$L_j^{-1}L_{j+1}^{-1} = (I + \ell_j e_j^T)(I + \ell_{j+1} e_{j+1}^T)$$

$$= I + \ell_j e_j^T + \ell_{j+1} e_{j+1}^T + \ell_j (e_j^T \ell_{j+1}) e_{j+1}$$

$$= I + \ell_j e_j^T + \ell_{j+1} e_{j+1}^T$$

Interestingly, this convenient result doesn’t hold for $L_{j+1}^{-1}L_j^{-1}$, why?
LU Factorization

Similarly,

$$L^{-1}_j L^{-1}_{j+1} L^{-1}_{j+2} = (I + \ell_j e_j^T + \ell_{j+1} e_{j+1}^T)(I + \ell_{j+2} e_{j+2}^T)$$

$$= I + \ell_j e_j^T + \ell_{j+1} e_{j+1}^T + \ell_{j+2} e_{j+2}^T$$

That is, to compute the product $L^{-1}_1 L^{-1}_2 \cdots L^{-1}_{n-1}$ we simply collect the subdiagonals for $j = 1, 2, \ldots, n-1$. 
LU Factorization

Hence, second stroke of luck:

\[
L \equiv L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1} =
\begin{bmatrix}
1 \\
\ell_{21} & 1 \\
\ell_{31} & \ell_{32} & 1 \\
\vdots & \vdots & \ddots & \ddots \\
\ell_{n1} & \ell_{n2} & \cdots & \ell_{n,n-1} & 1
\end{bmatrix}
\]
LU Factorization

Therefore, basic LU factorization algorithm is

1: \( U = A, \ L = I \)
2: for \( j = 1 : n - 1 \) do
3: \hspace{1em} for \( i = j + 1 : n \) do
4: \hspace{2em} \ell_{ij} = u_{ij} / u_{jj} \)
5: \hspace{2em} for \( k = j : n \) do
6: \hspace{3em} u_{ik} = u_{ik} - \ell_{ij} u_{jk} \)
7: \hspace{2em} end for
8: \hspace{1em} end for
9: end for

Note that the entries of \( U \) are updated each iteration so at the start of step \( j \), \( U = L_{j-1} L_{j-2} \cdots L_1 A \)

Here line 4 comes straight from the definition \( \ell_{ij} \equiv \frac{u_{ij}}{u_{jj}} \)
LU Factorization

Line 6 accounts for the effect of $L_j$ on columns $k = j, j + 1, \ldots, n$ of $U$

For $k = j : n$ we have

$$L_j u(:,k) \equiv \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -\ell_{j+1,j} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\ell_{nj} & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{1k} \\ u_{jk} \\ u_{j+1,k} \\ u_{nk} \end{bmatrix} = \begin{bmatrix} u_{1k} \\ \vdots \\ u_{jk} \\ u_{j+1,k} - \ell_{j+1,j} u_{jk} \\ \vdots \\ u_{nk} - \ell_{nj} u_{jk} \end{bmatrix}$$

The vector on the right is the updated $k^{th}$ column of $U$, which is computed in line 6
LU Factorization involves a triply-nested for-loop, hence $O(n^3)$ calculations.

Careful operation counting shows LU factorization requires $\sim \frac{1}{3} n^3$ additions and $\sim \frac{1}{3} n^3$ multiplications, $\sim \frac{2}{3} n^3$ operations in total.
Solving a linear system using LU

Hence to solve $Ax = b$, we perform the following three steps:

**Step 1:** Factorize $A$ into $L$ and $U$: $\sim \frac{2}{3} n^3$

**Step 2:** Solve $Ly = b$ by forward substitution: $\sim n^2$

**Step 3:** Solve $Ux = y$ by back substitution: $\sim n^2$

Total work is dominated by Step 1, $\sim \frac{2}{3} n^3$
Solving a linear system using LU

An alternative approach would be to compute $A^{-1}$ explicitly and evaluate $x = A^{-1}b$, but this is a bad idea!

**Question**: How would we compute $A^{-1}$?
Solving a linear system using LU

**Answer:** Let \( a^{\text{inv}}_{(:,k)} \) denote the \( k \)th column of \( A^{-1} \), then \( a^{\text{inv}}_{(:,k)} \) must satisfy

\[ A a^{\text{inv}}_{(:,k)} = e_k \]

Therefore to compute \( A^{-1} \), we first LU factorize \( A \), then back/forward substitute for rhs vector \( e_k \), \( k = 1, 2, \ldots, n \).

The \( n \) back/forward substitutions alone require \( \sim 2n^3 \) operations, inefficient!

A rule of thumb in Numerical Linear Algebra: It is almost always a bad idea to compute \( A^{-1} \) explicitly.
Solving a linear system using LU

Another case where LU factorization is very helpful is if we want to solve \( Ax = b_i \) for several different right-hand sides \( b_i, \ i = 1, \ldots, k \)

We incur the \( \sim \frac{2}{3}n^3 \) cost only once, and then each subsequent forward/back substitution costs only \( \sim 2n^2 \)

Makes a huge difference if \( n \) is large!
Stability of Gaussian Elimination

There is a problem with the LU algorithm presented above.

Consider the matrix

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

\( A \) is nonsingular, well-conditioned (\( \kappa(A) \approx 2.62 \)) but LU factorization fails at first step (division by zero).
Stability of Gaussian Elimination

LU factorization doesn’t fail for

\[ A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix} \]

but we get

\[ L = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{bmatrix} \]
Stability of Gaussian Elimination

Let's suppose that $-10^{20} \in \mathbb{F}$ (a floating point number) and that \( \text{round}(1 - 10^{20}) = -10^{20} \)

Then in finite precision arithmetic we get

\[
\begin{align*}
\tilde{L} &= \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \\
\tilde{U} &= \begin{bmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{bmatrix}
\end{align*}
\]
Stability of Gaussian Elimination

Hence due to rounding error we obtain

\[
\tilde{L}\tilde{U} = \begin{bmatrix}
10^{-20} & 1 \\
1 & 0
\end{bmatrix}
\]

which is not close to

\[
A = \begin{bmatrix}
10^{-20} & 1 \\
1 & 1
\end{bmatrix}
\]

Then, for example, let \( b = [3, 3]^T \)

- Using \( \tilde{L}\tilde{U} \), we get \( \tilde{x} = [3, 3]^T \)
- True answer is \( x = [0, 3]^T \)

Hence large relative error (rel. err. = 1) even though the problem is well-conditioned
Stability of Gaussian Elimination

In this example, standard Gaussian elimination yields a large residual.

Or equivalently, it yields the exact solution to a problem corresponding to a large input perturbation: \( \Delta b = [0, 3]^T \)

Hence \textit{unstable algorithm!} In this case the cause of the large error in \( x \) is numerical instability, not ill-conditioning.

To stabilize Gaussian elimination, we need to permute rows, \textit{i.e.} perform \textit{pivoting}.
Recall the Gaussian elimination process

\[
\begin{bmatrix}
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
x_{jj} & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
x_{jj} & \times & \times & \times & \times \\
0 & \times & \times & \times & \times \\
0 & \times & \times & \times & \times
\end{bmatrix}
\]

But we could just as easily do

\[
\begin{bmatrix}
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
x_{ij} & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
x_{ij} & \times & \times & \times & \times \\
0 & \times & \times & \times & \times \\
0 & \times & \times & \times & \times
\end{bmatrix}
\]
Partial Pivoting

The entry $x_{ij}$ is called the pivot, and flexibility in choosing the pivot is essential otherwise we can’t deal with:

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

From a numerical stability point of view, it is crucial to choose the pivot to be the largest entry in column $j$: “partial pivoting”\(^4\)

This ensures that each $\ell_{ij}$ entry — which acts as a multiplier in the LU factorization process — satisfies $|\ell_{ij}| \leq 1$

\(\text{\footnotesize{\(^4\)Full pivoting refers to searching through columns } j : n \text{ for the largest entry; this is more expensive and only marginal benefit to stability in practice}}\)
Partial Pivoting

To maintain the triangular LU structure, we permute rows by premultiplying by permutation matrices

\[
\begin{bmatrix}
\times & \times & \times & \times \\
\times & \times & \times & \\
\times & \times & \times & \\
\timesj & \times & \times & \\
\end{bmatrix}
\xrightarrow{P_1}
\begin{bmatrix}
\times & \times & \times & \times \\
\times & \times & \times & \\
\times & \times & \times & \\
\timesj & \times & \times & \\
\end{bmatrix}
\xrightarrow{L_1}
\begin{bmatrix}
\times & \times & \times & \times \\
\times & \times & \times & \\
\times & \times & \times & \\
\timesj & \times & \times & \\
\end{bmatrix}
\]

Pivot selection \quad Row interchange

In this case

\[
P_1 = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
\end{bmatrix}
\]

and each \( P_j \) is obtained by swapping two rows of I
Partial Pivoting

Therefore, with partial pivoting we obtain

\[ L_{n-1}P_{n-1} \cdots L_2P_2L_1P_1A = U \]

It can be shown (we omit the details here, see Trefethen & Bau) that this can be rewritten as

\[ PA = LU \]

where\(^5\) \( P \equiv P_{n-1} \cdots P_2P_1 \)

**Theorem**: Gaussian elimination with partial pivoting produces nonsingular factors \( L \) and \( U \) if and only if \( A \) is nonsingular.

\(^5\)The \( L \) matrix here is lower triangular, but not the same as \( L \) in the non-pivoting case: we have to account for the row swaps
Partial Pivoting

Pseudocode for LU factorization with partial pivoting (blue text is new):

1: \( U = A, \ L = I, \ P = I \)
2: \textbf{for} \( j = 1 : n - 1 \) \textbf{do}
3: \hspace{1em} \textbf{Select} \( i(\geq j) \) that maximizes \( |u_{ij}| \)
4: \hspace{1em} \textbf{Interchange} rows of \( U \): \( u_{(j,j:n)} \leftrightarrow u_{(i,j:n)} \)
5: \hspace{1em} \textbf{Interchange} rows of \( L \): \( \ell_{(j,1:j-1)} \leftrightarrow \ell_{(i,1:j-1)} \)
6: \hspace{1em} \textbf{Interchange} rows of \( P \): \( p_{(j,:)} \leftrightarrow p_{(i,:)} \)
7: \hspace{1em} \textbf{for} \( i = j + 1 : n \) \textbf{do}
8: \hspace{2em} \ell_{ij} = u_{ij} / u_{jj}
9: \hspace{2em} \textbf{for} \( k = j : n \) \textbf{do}
10: \hspace{3em} u_{ik} = u_{ik} - \ell_{ij} u_{jk}
11: \hspace{2em} \textbf{end for}
12: \hspace{1em} \textbf{end for}
13: \textbf{end for}

Again this requires \( \sim \frac{2}{3} n^3 \) floating point operations
Partial Pivoting: Solve $Ax = b$

To solve $Ax = b$ using the factorization $PA = LU$:

- Multiply through by $P$ to obtain $PAX = LUx = Pb$
- Solve $Ly = Pb$ using forward substitution
- Then solve $Ux = y$ using back substitution
Partial Pivoting in Python

Python’s \texttt{scipy.linalg.lu} function can do LU factorization with pivoting.

Python 2.7.10 (default, Feb 22 2019, 21:55:15)
[GCC 4.2.1 Compatible Apple LLVM 10.0.1 (clang-1001.0.37.14)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```python
>>> import numpy as np
>>> import scipy.linalg
>>> a=np.random.random((4,4))
>>> a
array([[ 0.91196125, 0.93236194, 0.56445227, 0.03338295],
       [ 0.48046154, 0.24213629, 0.54943058, 0.94345735],
       [ 0.16465853, 0.93604118, 0.54355272, 0.65252368],
       [ 0.65537703, 0.39023495, 0.15012939, 0.12540183]])
>>> (p,l,u)=scipy.linalg.lu(a)
>>> p
array([[ 1., 0., 0., 0.],
       [ 0., 0., 1., 0.],
       [ 0., 1., 0., 0.],
       [ 0., 0., 0., 1.]])
>>> l
array([[ 1. , 0. , 0. , 0. ],
       [ 0.1805543 , 1. , 0. , 0. ],
       [ 0.52684425, -0.32444117, 1. , 0. ],
       [ 0.7186457 , -0.36446949, -0.23915765, 1. ]])
>>> u
array([[ 0.91196125, 0.93236194, 0.56445227, 0.03338295],
       [ 0. , 0.76769922, 0.44163844, 0.64649625],
       [ 0. , 0. , 0.39533784, 1.13561973],
       [ 0. , 0. , 0. , 0.60863161]])
```
Stability of Gaussian Elimination

Numerical stability of Gaussian Elimination has been an important research topic since the 1940s.

Major figure in this field: James H. Wilkinson (English numerical analyst, 1919–1986)

Showed that for $Ax = b$ with $A \in \mathbb{R}^{n \times n}$:

- Gaussian elimination without partial pivoting is numerically unstable (as we’ve already seen)
- Gaussian elimination with partial pivoting satisfies

$$\frac{\|r\|}{\|A\|\|x\|} \leq 2^{n-1}n^2\epsilon_{\text{mach}}$$
Stability of Gaussian Elimination

That is, pathological cases exist where the relative residual, \( \| r \| / \| A \| \| x \| \), grows exponentially with \( n \) due to rounding error.

Worst case behavior of Gaussian Elimination with partial pivoting is explosive instability but such pathological cases are extremely rare!

In over 50 years of Scientific Computation, instability has only been encountered due to deliberate construction of pathological cases.

In practice, Gaussian elimination is stable in the sense that it produces a small relative residual.
Stability of Gaussian Elimination

In practice, we typically obtain

\[
\frac{\|r\|}{\|A\| \|x\|} \lesssim n\epsilon_{\text{mach}},
\]

i.e. grows only linearly with \( n \), and is scaled by \( \epsilon_{\text{mach}} \)

Combining this result with our inequality:

\[
\frac{\|\Delta x\|}{\|x\|} \leq \kappa(A) \frac{\|r\|}{\|A\| \|x\|}
\]

implies that in practice Gaussian elimination gives small error for well-conditioned problems!
Cholesky Factorization
Cholesky factorization

Suppose that $A \in \mathbb{R}^{n \times n}$ is an “SPD” matrix, i.e.:

- **Symmetric**: $A^T = A$
- **Positive Definite**: for any $v \neq 0$, $v^T A v > 0$

Then the LU factorization of $A$ can be arranged so that $U = L^T$, i.e. $A = LL^T$ (but in this case $L$ may not have 1s on the diagonal)

Consider the $2 \times 2$ case:

\[
\begin{bmatrix}
    a_{11} & a_{21} \\
    a_{21} & a_{22}
\end{bmatrix}
= \begin{bmatrix}
    \ell_{11} & 0 \\
    \ell_{21} & \ell_{22}
\end{bmatrix}
\begin{bmatrix}
    \ell_{11} & \ell_{21} \\
    0 & \ell_{22}
\end{bmatrix}
\]

Equating entries gives

\[
\ell_{11} = \sqrt{a_{11}}, \quad \ell_{21} = a_{21}/\ell_{11}, \quad \ell_{22} = \sqrt{a_{22} - \ell_{21}^2}
\]
Cholesky factorization

This approach of equating entries can be used to derive the Cholesky factorization for the general $n \times n$ case

1: $L = A$
2: for $j = 1 : n$ do
3: \[ l_{jj} = \sqrt{l_{jj}} \]
4: for $i = j + 1 : n$ do
5: \[ l_{ij} = l_{ij}/l_{jj} \]
6: end for
7: for $k = j + 1 : n$ do
8: for $i = k : n$ do
9: \[ l_{ik} = l_{ik} - l_{ij}l_{kj} \]
10: end for
11: end for
12: end for
Notes on Cholesky factorization:

- For an SPD matrix $A$, Cholesky factorization is numerically stable and does not require any pivoting.

- Operation count: $\sim \frac{1}{3} n^3$ operations in total, i.e. about half as many as Gaussian elimination.

- Only need to store $L$, hence uses less memory than LU.
Sparse Matrices
Sparse Matrices

In applications, we often encounter **sparse matrices**

A prime example is in discretization of partial differential equations (covered in the next section)

“Sparse matrix” is not precisely defined, roughly speaking it is a matrix that is “mostly zeros”

From a computational point of view it is advantageous to store only the non-zero entries

The set of non-zero entries of a sparse matrix is referred to as its **sparsity pattern**
Sparse Matrices

\[
A = \\
\begin{pmatrix}
2 & 2 & 2 & 2 & 2 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
A_{\text{sparse}} = \\
(1,1) \ 2 \\
(1,2) \ 2 \\
(2,2) \ 1 \\
(1,3) \ 2 \\
(3,3) \ 1 \\
(1,4) \ 2 \\
(4,4) \ 1 \\
(1,5) \ 2 \\
(5,5) \ 1
\]
Sparse Matrices

From a mathematical point of view, sparse matrices are no different from dense matrices.

But from Sci. Comp. perspective, sparse matrices require different data structures and algorithms for computational efficiency.

e.g., can apply LU or Cholesky to sparse $A$, but “new” non-zeros ($i.e.$ outside sparsity pattern of $A$) are introduced in the factors.

These new non-zero entries are called “fill-in” — many methods exist for reducing fill-in by permuting rows and columns of $A$. 