AM 205: lecture 7

- Last time: LU factorization
- Today's lecture: Cholesky factorization, timing, QR factorization
- Reminder: assignment 1 due at 5 PM on Friday September 25
LU Factorization

Therefore, basic LU factorization algorithm is

1: $U = A$, $L = I$
2: for $j = 1 : n - 1$ do
3: for $i = j + 1 : n$ do
4: \[ \ell_{ij} = \frac{u_{ij}}{u_{jj}} \]
5: for $k = j : n$ do
6: \[ u_{ik} = u_{ik} - \ell_{ij} u_{jk} \]
7: end for
8: 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

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

The vector on the right is the updated $k^{th}$ column of $U$, which is computed in line 6
LU Factorization

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 \( a_{(:,k)}^{\text{inv}} \) denote the \( k \)th column of \( A^{-1} \), then \( a_{(:,k)}^{\text{inv}} \) must satisfy

\[
Aa_{(:,k)}^{\text{inv}} = e_k
\]

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**
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 subsitution 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).
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
\[
\tilde{L} = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \quad \tilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{bmatrix}
\]
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 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, i.e. perform pivoting.
## Pivoting

Recall the Gaussian elimination process

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

But we could just as easily do

\[
\begin{bmatrix}
\times & \times & \times & \times & \times \\
\times & \times & \times \\
x_{ij} & \times & \times \\
\times & \times & \times
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\times & \times & \times & \times & \times \\
0 & \times & \times \\
x_{ij} & \times & \times \\
0 & \times & \times
\end{bmatrix}
\]
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”\(^1\)

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

---

\(^1\)Full pivoting refers to searching through columns $j : n$ 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

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

Pivot selection   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\(^2\) \( P \equiv P_{n-1} \cdots P_2P_1 \)

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

\(^2\text{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: \quad \text{Select} \( i(\geq j) \) that maximizes \( |u_{ij}| \)
4: \quad \text{Interchange rows of} \ U: \ u(j,j:n) \leftrightarrow u(i,j:n)
5: \quad \text{Interchange rows of} \ L: \ \ell(j,1:j-1) \leftrightarrow \ell(i,1:j-1)
6: \quad \text{Interchange rows of} \ P: \ p(j,:) \leftrightarrow p(i,:)
7: \textbf{for} \ i = j + 1 : n \ \textbf{do}
8: \quad \ell_{ij} = u_{ij}/u_{jj}
9: \quad \textbf{for} \ k = j : n \ \textbf{do}
10: \quad \quad u_{ik} = u_{ik} - \ell_{ij}u_{jk}
11: \quad \textbf{end for}
12: \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 `scipy.linalg.lu` function can do LU factorization with pivoting.

```python
Python 2.7.5 (default, Mar 9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> import numpy as np
>>> import scipy.linalg
>>> a=np.random.random((4,4))
>>> a
array([[ 0.30178809, 0.09895414, 0.75341645, 0.55745407],
       [ 0.08879282, 0.97137694, 0.04768167, 0.28140464],
       [ 0.87253281, 0.66021495, 0.4941091 , 0.52966743],
       [ 0.7990001 , 0.45251929, 0.55493106, 0.15781707]])
>>> (p,l,u)=scipy.linalg.lu(a)
>>> p
array([[ 0., 0., 1., 0.],
       [ 0., 1., 0., 0.],
       [ 1., 0., 0., 0.],
       [ 0., 0., 0., 1.]])
>>> l
array([[ 1. , 0. , 0. , 0. ]],
       [ 0.10176445, 1. , 0. , 0. ],
       [ 0.34587592, -0.14310957, 1. , 0. ],
       [ 0.91572499, -0.16816814, 0.17525841, 1. ]])
>>> u
array([[ 0.87253281, 0.66021495, 0.4941091 , 0.52966743],
       [ 0. , 0.90419053, -0.00260107, 0.22750332],
       [ 0. , 0. , 0.58214377, 0.40681276],
       [ 0. , 0. , 0. , -0.36025118]])
```
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 $v \neq 0$, $v^T Av > 0$

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: $\ell_{jj} = \sqrt{\ell_{jj}}$
4: for $i = j + 1 : n$ do
5: $\ell_{ij} = \ell_{ij}/\ell_{jj}$
6: end for
7: for $k = j + 1 : n$ do
8: for $i = k : n$ do
9: $\ell_{ik} = \ell_{ik} - \ell_{ij}\ell_{kj}$
10: end for
11: end for
12: end for
Cholesky factorization

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

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

\[ A_{\text{sparse}} = \begin{pmatrix}
(1,1) & 2 \\
(1,2) & 2 \\
(2,2) & 1 \\
(1,3) & 2 \\
(3,3) & 1 \\
(1,4) & 2 \\
(4,4) & 1 \\
(1,5) & 2 \\
(5,5) & 1
\end{pmatrix} \]
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 $A$. 

QR Factorization
A square matrix $Q \in \mathbb{R}^{n \times n}$ is called orthogonal if its columns and rows are orthonormal vectors.

Equivalently, $Q^T Q = QQ^T = I$.

Orthogonal matrices preserve the Euclidean norm of a vector, i.e.

$$\|Qv\|_2^2 = v^T Q^T Qv = v^T v = \|v\|_2^2$$

Hence, geometrically, we picture orthogonal matrices as reflection or rotation operators.

Orthogonal matrices are very important in scientific computing, norm-preservation implies no amplification of numerical error!
QR Factorization

A matrix $A \in \mathbb{R}^{m \times n}$, $m \geq n$, can be factorized into

$$A = QR$$

where

- $Q \in \mathbb{R}^{m \times m}$ is orthogonal
- $R \equiv \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} \in \mathbb{R}^{m \times n}$
- $\hat{R} \in \mathbb{R}^{n \times n}$ is upper-triangular

QR is very good for solving overdetermined linear least-squares problems, $Ax \simeq b$ $^3$

$^3$QR can also be used to solve a square system $Ax = b$, but requires $\sim 2 \times$ as many operations as Gaussian elimination hence not the standard choice
To see why, consider the 2-norm of the least squares residual:

\[ \| r(x) \|_2^2 = \| b - Ax \|_2^2 = \| b - Q \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} x \|_2^2 \]

\[ = \| Q^T \left( b - Q \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} x \right) \|_2^2 \]

\[ = \| Q^T b - \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} x \|_2^2 \]

(We used the fact that \( \| Q^T z \|_2 = \| z \|_2 \) in the second line)
Then, let $Q^T b = [c_1, c_2]^T$ where $c_1 \in \mathbb{R}^n$, $c_2 \in \mathbb{R}^{m-n}$, so that

$$
\|r(x)\|_2^2 = \|c_1 - \hat{R}x\|_2^2 + \|c_2\|_2^2
$$

**Question:** Based on this expression, how do we minimize $\|r(x)\|_2$?
**Answer:** We can’t influence the second term, \( \|c_2\|_2^2 \), since it doesn’t contain an \( x \).

Hence we minimize \( \|r(x)\|_2^2 \) by making the first term zero.

That is, we solve the \( n \times n \) triangular system \( \hat{R}x = c_1 \) — this what Python does in its `linalg` function for solving least squares.

Also, this tells us that \( \min_{x \in \mathbb{R}^n} \|r(x)\|_2 = \|c_2\|_2 \).