Applied Mathematics 205

Unit II: Numerical Linear Algebra

Lecturer: Dr. David Knezevic

Unit II: Numerical Linear Algebra

Chapter II.2: LU and Cholesky Factorizations

Preliminaries

Preliminaries

In this chapter we will focus on linear systems $Ax = b$ for $A \in \mathbb{R}^{n \times n}$ and $b, x \in \mathbb{R}^n$

Recall that it is often helpful to think of matrix multiplication as a linear combination of the columns of A, where x_j are the weights

That is, we have $b = Ax = \sum_{j=1}^{n} x_j a_{(:,j)}$ where $a_{(:,j)} \in \mathbb{R}^n$ is the jth column of A and x_j are scalars

Preliminaries

This can be displayed schematically as

$$
\begin{bmatrix} \\ b \\ \\ \end{bmatrix} = \left[\begin{array}{c|c|c|c} & & & \\ a_{(:,1)} & a_{(:,2)} & \cdots & a_{(:,n)} \\ & & & \end{array}\right] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}
$$

$$
= x_1 \begin{bmatrix} \\ a_{(:,1)} \\ \\ \end{bmatrix} + \cdots + x_n \begin{bmatrix} \\ a_{(:,n)} \\ \\ \end{bmatrix}
$$

Preliminaries

We therefore can interpret $Ax = b$ as: "x is the vector of coefficients of the expansion of b in the basis of columns of A"

E.g. from "linear combination of columns" view we immediately see that $Ax = b$ has a solution if

$$b \in \mathrm{span}\{a_{(:,1)}, a_{(:,2)}, \cdots, a_{(:,n)}\}$$

(this holds even if A isn't square)

Let us write $\mathrm{image}(A) \equiv \mathrm{span}\{a_{(:,1)}, a_{(:,2)}, \cdots, a_{(:,n)}\}$

Preliminaries

Existence and Uniqueness:

Solution $x \in \mathbb{R}^n$ exists if $b \in \mathrm{image}(A)$

If solution x exists and the set $\{a_{(:,1)}, a_{(:,2)}, \cdots, a_{(:,n)}\}$ is linearly independent, then x is unique[1]

If solution x exists and $\exists z \neq 0$ such that $Az = 0$, then also $A(x + \gamma z) = b$ for any $\gamma \in \mathbb{R}$, hence infinitely many solutions

If $b \notin \mathrm{image}(A)$ then $Ax = b$ has no solution

[1] Linear independence of columns of A is equivalent to $Az = 0 \implies z = 0$

Preliminaries

The inverse map $A^{-1}\colon \mathbb{R}^n \to \mathbb{R}^n$ is well-defined if and only if $Ax = b$ has unique solution for all $b \in \mathbb{R}^n$

Unique matrix $A^{-1} \in \mathbb{R}^{n\times n}$ such that $AA^{-1} = A^{-1}A = \mathrm{I}$ exists if any of the following equivalent conditions are satisfied:

- $\det(A) \neq 0$
- $\mathrm{rank}(A) = n$
- For any $z \neq 0$, $Az \neq 0$ (null space of A is $\{0\}$)

A is non-singular if A^{-1} exists, and then $x = A^{-1}b \in \mathbb{R}^n$

A is singular if A^{-1} does not exist

Norms

A norm $\|\cdot\| : V \to \mathbb{R}$ is a function on a vector space V that satisfies

- $\|x\| \geq 0$ and $\|x\| = 0 \implies x = 0$
- $\|\gamma x\| = |\gamma|\|x\|$, for $\gamma \in \mathbb{R}$
- $\|x + y\| \leq \|x\| + \|y\|$

Norms

Also, the triangle inequality implies another helpful inequality: the "reverse triangle inequality", $|\|x\| - \|y\|| \leq \|x - y\|$

Proof: Let $a = y$, $b = x - y$, then

$$\|x\| = \|a+b\| \leq \|a\|+\|b\| = \|y\|+\|x-y\| \implies \|x\|-\|y\| \leq \|x-y\|$$

Repeat with $a = x$, $b = y - x$ to show $|\|x\| - \|y\|| \leq \|x - y\|$

Vector Norms

Let's now introduce some common norms on $\mathbb{R}^n$

Most common norm is the Euclidean norm (or 2-norm):

$$\|x\|_2 \equiv \sqrt{\sum_{j=1}^{n} x_j^2}$$

2-norm is special case of the p-norm for any $p \geq 1$:

$$\|x\|_p \equiv \left(\sum_{j=1}^{n} |x_j|^p \right)^{1/p}$$

Also, limiting case as $p \to \infty$ is the ∞-norm:

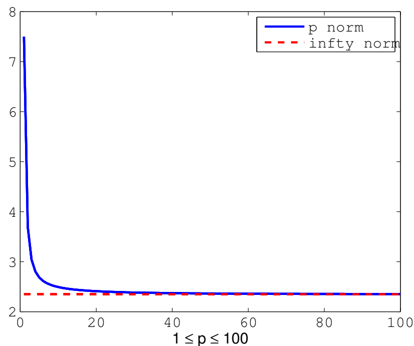
$$\|x\|_\infty \equiv \max_{1 \leq i \leq n} |x_i|$$

Vector Norms

```
% define a vector
x = [1.2 0.5 -0.1 2.3 -1.05 -2.35].';

% compute ||x||_p, 1 \leq p \leq 100
p_vector = 1:100;
for p=p_vector
    p_norm(p) = norm(x,p);
end

% compare to ||x||_\infty
inf_norm = norm(x,'inf')

% plot the norms
plot(p_vector,p_norm,'linewidth',2);
hold on
plot(p_vector,inf_norm*ones(size(p_vector)),'r--','linewidth',2)
```

Vector Norms

We see that p-norm approaches ∞-norm as p gets large



Vector Norms

We generally use whichever norm is most convenient/appropriate for a given problem, e.g. 2-norm for least-squares analysis

Different norms give different (but related) measures of size

In particular, an important mathematical fact is:

> All norms on a finite dimensional space (such as $\mathbb{R}^n$) are equivalent

Vector Norms

That is, let $\|\cdot\|_a$ and $\|\cdot\|_b$ be two norms on a finite dimensional space V, then $\exists c_1, c_2 \in \mathbb{R}_{>0}$ such that for any $x \in V$

$$c_1\|x\|_a \leq \|x\|_b \leq c_2\|x\|_a$$

(Also, from above we have $\frac{1}{c_2}\|x\|_b \leq \|x\|_a \leq \frac{1}{c_1}\|x\|_b$)

Hence if we can derive an inequality in an arbitrary norm on V, it applies (after appropriate scaling) in any other norm too

Vector Norms

In some cases we can explicitly calculate values for c_1, c_2:

e.g. $\|x\|_2 \leq \|x\|_1 \leq \sqrt{n}\|x\|_2$, since

$$\|x\|_2^2 = \left(\sum_{j=1}^{n} |x_j|^2\right) \leq \left(\sum_{j=1}^{n} |x_j|\right)^2 = \|x\|_1^2 \implies \|x\|_2 \leq \|x\|_1$$

[e.g. consider $|a|^2 + |b|^2 \leq |a|^2 + |b|^2 + 2|a||b| = (|a| + |b|)^2$]

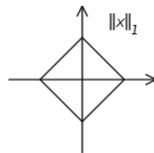
$$\|x\|_1 = \sum_{j=1}^{n} 1 \times |x_j| \leq \left(\sum_{j=1}^{n} 1^2\right)^{1/2} \left(\sum_{j=1} |x_j|^2\right)^{1/2} = \sqrt{n}\,\|x\|_2$$

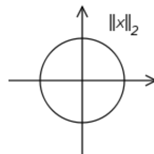
[We used Cauchy-Schwarz inequality in $\mathbb{R}^n$:
$\sum_{j=1}^{n} a_j b_j \leq (\sum_{j=1}^{n} a_j^2)^{1/2} (\sum_{j=1}^{n} b_j^2)^{1/2}$]

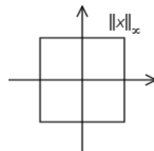
Vector Norms

Here we show "unit circle"
$\{x \in \mathbb{R}^2 : \|x\|_p = 1\}$ for $p = 1, 2, \infty$

Different norms give different measurements of size







Matrix Norms

There are many ways to define norms on matrices

For example, the Frobenius norm is defined as:

$$\|A\|_F \equiv \left(\sum_{i=1}^{n} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2}$$

(If we think of A as a vector in $\mathbb{R}^{n^2}$, then Frobenius is equivalent to the vector 2-norm of A)

Matrix Norms

Usually the matrix norms induced by vector norms are most useful, e.g.:

$$\|A\|_p \equiv \max_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p} = \max_{\|x\|_p = 1} \|Ax\|_p$$

The fact that this defines a norm follows from the properties of the vector norm, e.g. proof of triangle inequality for $A, B \in \mathbb{R}^{n \times n}$:

$$\begin{aligned} \|A + B\|_p &= \max_{\|x\|_p = 1} \|(A + B)x\|_p \\ &\leq \max_{\|x\|_p = 1} (\|Ax\|_p + \|Bx\|_p) \\ &\leq \max_{\|x\|_p = 1} \|Ax\| + \max_{\|x\|_p = 1} \|Bx\|_p \\ &= \|A\|_p + \|B\|_p. \end{aligned}$$

Matrix Norms

The definition of $\|A\|_p$ also implies the useful property $\|Ax\|_p \leq \|A\|_p \|x\|_p$, since

$$\|Ax\|_p = \frac{\|Ax\|_p}{\|x\|_p} \|x\|_p \leq \left(\max_{v \neq 0} \frac{\|Av\|_p}{\|v\|_p} \right) \|x\|_p = \|A\|_p \|x\|_p$$

Matrix Norms

See Lecture: The 1-norm and ∞-norm can be calculated straightforwardly:

$$\begin{aligned} \|A\|_1 &= \max_{1 \le j \le n} \|a_{(:,j)}\|_1 && \text{(max column sum)} \\ \|A\|_\infty &= \max_{1 \le i \le n} \|a_{(i,:)}\|_1 && \text{(max row sum)} \end{aligned}$$

We will see how to compute the matrix 2-norm next chapter

Condition Number

Recall from Unit 0 that the condition number of $A \in \mathbb{R}^{n \times n}$ is defined as

$$\kappa(A) \equiv \|A\|\|A^{-1}\|$$

The value of $\kappa(A)$ depends on which norm we use

Calculate condition number in Matlab with `cond`:

```
>> help cond
 COND(X) returns the 2-norm condition number...

 COND(X,P) returns the condition number of X in P-norm:

    NORM(X,P) * NORM(INV(X),P).

 where P = 1, 2, inf, or 'fro'.
```

If A is singular then by convention we say $\kappa(A) = \infty$

The Residual

Recall that the residual $r(x) = b - Ax$ was crucial in least-squares problems

It is also crucial in assessing the accuracy of a computed solution ($\hat{x}$) to a square linear system $Ax = b$

The residual $r(\hat{x})$ can always be computed, and if we're careful we can use it to indicate the quality of our solution

Question: What do we mean here by careful?

Answer: Don't just use the norm of the residual as an indicator of accuracy for a solution of $Ax = b$

The Residual

We have that $\|\Delta x\| = \|x - \hat{x}\| = 0$ if and only if $\|r(\hat{x})\| = 0$

However, small residual doesn't necessarily imply small $\|\Delta x\|$

Observe that

$$\|\Delta x\| = \|x - \hat{x}\| = \|A^{-1}(b - A\hat{x})\| = \|A^{-1}r(\hat{x})\| \leq \|A^{-1}\|\|r(\hat{x})\|$$

Hence

$$\frac{\|\Delta x\|}{\|\hat{x}\|} \leq \frac{\|A^{-1}\|\|r(\hat{x})\|}{\|\hat{x}\|} = \frac{\|A\|\|A^{-1}\|\|r(\hat{x})\|}{\|A\|\|\hat{x}\|} = \kappa(A)\frac{\|r(\hat{x})\|}{\|A\|\|\hat{x}\|} \quad (*)$$

The Residual

Define the relative residual as $\|r(\hat{x})\|/(\|A\|\|\hat{x}\|)$

Then our inequality states that "relative error is bounded by condition number times relative residual"

This is just like our condition number relationship from Unit 0:

$$\kappa(A) \geq \frac{\|\Delta x\|/\|x\|}{\|\Delta b\|/\|b\|}, \qquad \text{i.e.} \qquad \frac{\|\Delta x\|}{\|x\|} \leq \kappa(A)\frac{\|\Delta b\|}{\|b\|} \quad (**)$$

Inequalities $(*)$ and $(**)$ are closely related because the residual is the "input pertubation" (i.e. Δb) for $Ax = b$

The Residual

This is because we can consider $\hat{x}$ to be the exact solution for some perturbed input $A\hat{x} = b + \Delta b$

The residual associated with $\hat{x}$ is $r(\hat{x}) = b - A\hat{x} = -\Delta b$, i.e. $\|r(\hat{x})\| = \|\Delta b\|$

In general, a numerically stable algorithm gives us the exact solution to a slightly perturbed problem,[2] i.e. a small residual

This is a reasonable expectation for a stable algorithm: rounding error doesn't accumulate, so effective input perturbation is small

[2] Recall from Unit 0 that this is called a "backward stable algorithm"

The Residual

As in Unit 0, inequality $(*)$ demonstrates the importance of having a well-conditioned problem

We can use a stable numerical method to get a small rel. residual, but this only guarantees small rel. error if A is well-conditioned!

E.g. we saw this with normal equations in I.3: Small residual (stable method for $A^T Ax = A^T b$) but large error ($\kappa(A^T A) \gg 1$)

The Residual (Heath, Example 2.8)

Consider a 2×2 example to clearly demonstrate the difference between residual and error

$$Ax = \begin{bmatrix} 0.913 & 0.659 \\ 0.457 & 0.330 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.254 \\ 0.127 \end{bmatrix} = b$$

The exact solution is given by $x = [1, -1]^T$

Suppose we compute two different approximate solutions (e.g. using two different algorithms)

$$\hat{x}_{(i)} = \begin{bmatrix} -0.0827 \\ 0.5 \end{bmatrix}, \qquad \hat{x}_{(ii)} = \begin{bmatrix} 0.999 \\ -1.001 \end{bmatrix}$$

The Residual (Heath, Example 2.8)

Then,

$$\|r(\hat{x}_{(i)})\|_1 = 2.1 \times 10^{-4}, \qquad \|r(\hat{x}_{(ii)})\|_1 = 2.4 \times 10^{-2}$$

but

$$\|x - \hat{x}_{(i)}\|_1 = 2.58, \qquad \|x - \hat{x}_{(ii)}\|_1 = 0.002$$

In this case, $\hat{x}_{(ii)}$ is better solution, but has larger residual!

This is possible here because $\kappa(A) = 1.25 \times 10^4$ is quite large (i.e. rel. error $\leq 1.25 \times 10^4 \times$ rel. residual)

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"

$$
\begin{aligned}
x_n &= b_n / u_{nn} \\
x_{n-1} &= (b_{n-1} - u_{n-1,n} x_n) / u_{n-1,n-1} \\
&\vdots \\
x_j &= \left(b_j - \sum_{k=j+1}^{n} u_{jk} x_k \right) / u_{jj} \\
&\vdots
\end{aligned}
$$

Solving $Ax = b$

Similarly, we can use forward substitution for a lower triangular system $Lx = b$

$$
\begin{aligned}
x_1 &= b_1/\ell_{11} \\
x_2 &= (b_2 - \ell_{21}x_1)/\ell_{22} \\
&\vdots \\
x_j &= \left(b_j - \sum_{k=1}^{j-1} \ell_{jk}x_k\right)/\ell_{jj} \\
&\vdots
\end{aligned}
$$

Solving $Ax = b$

Back and forward substitution can be implemented with doubly nested for-loops (see Assignment 2)

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

We have $j - 1$ additions and multiplications 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 "$\sim$" 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 "$\sim$" 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 $L \equiv M^{-1}$ 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 $X (\equiv L_{j-1} L_{j-2} \cdots L_1 A)$ 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

$$L_j x_{(:,j)} \equiv \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -x_{j+1,j}/x_{jj} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -x_{nj}/x_{jj} & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_{1j} \\ \vdots \\ x_{jj} \\ x_{j+1,j} \\ \vdots \\ x_{nj} \end{bmatrix} = \begin{bmatrix} x_{1j} \\ \vdots \\ x_{jj} \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

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 4×4 matrix, we have

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

Key point: L_k does not affect columns $1, 2, \ldots, k-1$ of $L_{k-1} L_{k-2} \ldots L_1 A$

LU Factorization

After $n-1$ steps, we obtain the upper triangular matrix
$U = L_{n-1} \cdots L_2 L_1 A$

$$U = \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix}$$

LU Factorization

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 $\ell_j \equiv [0, \ldots, 0, \ell_{j+1,j}, \ldots, \ell_{nj}]^T$ so that $L_j = \mathrm{I} - \ell_j e_j^T$

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

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

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

Hence we have shown that $L_j^{-1} = (\mathrm{I} + \ell_j e_j^T)$

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

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

LU Factorization

Similarly,

$$
\begin{aligned}
L_j^{-1} L_{j+1}^{-1} L_{j+2}^{-1} &= (\mathrm{I} + \ell_j e_j^T + \ell_{j+1} e_{j+1}^T)(\mathrm{I} + \ell_{j+2} e_{j+2}^T) \\
&= \mathrm{I} + \ell_j e_j^T + \ell_{j+1} e_{j+1}^T + \ell_{j+2} e_{j+2}^T
\end{aligned}
$$

That is, to compute the product $L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1}$ 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:     for i = j + 1 : n do
4:         ℓ_ij = u_ij / u_jj
5:         for k = j : n do
6:             u_ik = u_ik − ℓ_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

Example of LU Factorization: See Lecture

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 $a^{\mathrm{inv}}_{(:,k)}$ denote the kth column of A^{-1}, then $a^{\mathrm{inv}}_{(:,k)}$ must satisfy

$$Aa^{\mathrm{inv}}_{(:,k)} = e_k$$

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 subequent forward/back subsitution costs only $\sim 2n^2$

Makes a huge difference if n is large!

Matlab example: Comparison of computation time for many rhs's

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

$$L = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{bmatrix}$$

Stability of Gaussian Elimination

Let's suppose that $10^{-20}, 10^{20} \in \mathbb{F}$ (floating point number, cf. Unit 0) and that $\mathrm{round}(1 - 10^{20}) = -10^{20}$

So that in finite precision arithmetic we get

$$\widetilde{L} = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \qquad \widetilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{bmatrix}$$

Stability of Gaussian Elimination

Hence due to rounding error we obtain

$$\widetilde{L}\widetilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 0 \end{bmatrix}$$

which is not close to

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

For example, suppose we want to solve $Ax = b$ for $b = [3, 3]^T$

- Using $\widetilde{L}\widetilde{U}$, 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 elim. yields a large residual

Or equivalently, it yields the exact solution to a problem corresponding to a “large input perturbation”

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

$$\begin{bmatrix} \times & \times & \times & \times \\ & x_{jj} & \times & \times \\ & \times & \times & \times \\ & \times & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times \\ & x_{jj} & \times & \times \\ & 0 & \times & \times \\ & 0 & \times & \times \end{bmatrix}$$

But we could just as easily do

$$\begin{bmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & x_{ij} & \times & \times \\ & \times & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times \\ & 0 & \times & \times \\ & x_{ij} & \times & \times \\ & 0 & \times & \times \end{bmatrix}$$

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"[4]

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

[4] Full pivoting refers to searching through columns $j:n$ as well for the largest entry; 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

$$\begin{bmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & \times & \times & \times \\ & x_{ij} & \times & \times \end{bmatrix} \xrightarrow{P_1} \begin{bmatrix} \times & \times & \times & \times \\ & x_{ij} & \times & \times \\ & \times & \times & \times \\ & \times & \times & \times \end{bmatrix} \xrightarrow{L_1} \begin{bmatrix} \times & \times & \times & \times \\ & x_{ij} & \times & \times \\ & 0 & \times & \times \\ & 0 & \times & \times \end{bmatrix}$$

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[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:  for j = 1 : n − 1 do
3:      Select i(≥ j) that maximizes |u_ij|
4:      Interchange rows of U: u_(j,j:n) ↔ u_(i,j:n)
5:      Interchange rows of L: ℓ_(j,1:j−1) ↔ ℓ_(i,1:j−1)
6:      Interchange rows of P: p_(j,:) ↔ p_(i,:)
7:      for i = j + 1 : n do
8:          ℓ_ij = u_ij / u_jj
9:          for k = j : n do
10:             u_ik = u_ik − ℓ_ij u_jk
11:         end for
12:     end for
13: 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 Matlab

Matlab's `lu` function does Gaussian elimination with partial pivoting, `[L,U,P] = lu(A)` gives all three matrices:

```
>> A = rand(4);
>> [L,U,P] = lu(A)

 L =
      1.0000   0        0        0
      0.6654   1.0000   0        0
      0.2042   0.1686   1.0000   0
      0.3918   0.1069   0.3327   1.0000
 U =
      0.7943   0.6020   0.7482    0.9133
      0         0.2535  -0.4140   0.2181
      0        0        0.6062    0.0057
      0        0        0        -0.2307
 P =
      0   1   0   0
      0   0   0   1
      1   0   0   0
      0   0   1   0
```

Partial Pivoting in Matlab

When we do `[L,U] = lu(A)`, Matlab gives us a matrix L in "psychologically lower triangular form"

That is, there exists P such that PL is lower triangular — this form is just as good for solving $Ax = b$

```
>> [L,U] = lu(A)

 L =
      0.2042   0.1686   1.0000   0
      1.0000   0        0        0
      0.3918   0.1069   0.3327   1.0000
      0.6654   1.0000   0        0
 U =
      0.7943   0.6020   0.7482    0.9133
      0         0.2535   -0.4140   0.2181
      0         0        0.6062    0.0057
      0         0        0         -0.2307
```

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 (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

$$\frac{\|r\|}{\|A\|\|x\|} \leq 2^{n-1} n^2 \epsilon_{\mathrm{mach}}$$

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 Sci. Comp., 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

$$\frac{\|r\|}{\|A\|\|x\|} \lesssim n\epsilon_{\mathrm{mach}},$$

i.e. residual grows only linearly with n, and is scaled by ϵ_{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 2×2 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
```

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 LU factorization
- Only need to store L, hence uses less memory than LU

Cholesky factorization

Matlab "backslash" will check if matrix is SPD, and will use Cholesky if possible

```
% generate a random matrix and rhs
n = 2000;
Z = randn(n,n); b = randn(n,1);

% solve using LU factorization
tic; x = Z\b; toc
Elapsed time is 0.653953 seconds.

% generate an SPD matrix A
A = Z'*Z;
tic; x = A\b; toc
Elapsed time is 0.274377 seconds.
```

Sparse Matrices

Sparse Matrices

In applications, we often encounter sparse matrices

Examples: Discretization of partial differential equations, or representing/analyzing networks (e.g. web search)

"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

Sparse matrices are handled in a natural way by Matlab

e.g. `sparse(A)` "squeezes out" any zero entries in A

```
n = 5;
A = eye(n);
A(1,:)  = 2*ones(1,n);
A_sparse = sparse(A);
```

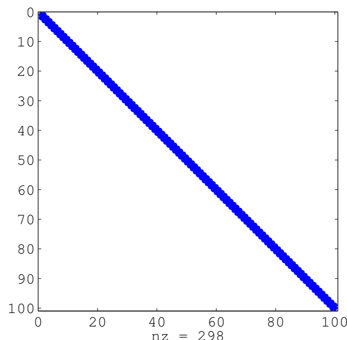
Sparse Matrices

```
 A =
      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

A_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

Can visualize the sparsity pattern in Matlab with spy, e.g. consider a matrix with -2 on the diagonal, 1 on sub and super diagonal

```
n = 100;
e = ones(n,1);
A = spdiags([e -2*e e], -1:1, n, n);
spy(A);
```



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

For example, we can use standard LU for sparse A, but "new" non-zeros (outside sparsity pattern of A) arise in the factors

These new non-zero entries are called "fill-in" — many methods exist for reducing fill-in by reordering rows and columns of A

Matlab demo (from Mathworks): Reordering algorithms for reducing fill-in