AM 205: lecture 7

- Last time: introduction to numerical linear algebra
- Today's lecture: LU factorization
- Reminder: assignment 1 due at 5 PM on Friday September 26

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

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 "$\sim$" refers to asymptotic behavior, *e.g.*

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

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 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[1]

Then solve for $Ux = y$ via back substitution

[1] $y = L^{-1}b$ 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

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

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

By the same argument $(\mathrm{I} + \ell_j e_j^T)L_j = \mathrm{I}$, and hence $L_j^{-1} = (\mathrm{I} + \ell_j e_j^T)$

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

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

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

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: $U = A$, $L = \mathrm{I}$
2: **for** $j = 1 : n - 1$ **do**
3:     **for** $i = j + 1 : n$ **do**
4:         $\ell_{ij} = u_{ij}/u_{jj}$
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 $a^{\text{inv}}_{(:,k)}$ denote the kth column of A^{-1}, then $a^{\text{inv}}_{(:,k)}$ must satisfy

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

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 subequent 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)

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 $-10^{20} \in \mathbb{F}$ (a floating point number) and that $\mathrm{round}(1 - 10^{20}) = -10^{20}$

Then 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

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

Then, for example, let $b = [3,3]^T$

- 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 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

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

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"[2]

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

[2]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

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

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[3] $P \equiv P_{n-1}\cdots P_2P_1$

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

[3]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):

```
U = A, L = I, P = I
for j = 1 : n-1 do
    Select i(≥ j) that maximizes |u_ij|
    Interchange rows of U: u_(j,j:n) ↔ u_(i,j:n)
    Interchange rows of L: ℓ_(j,1:j-1) ↔ ℓ_(i,1:j-1)
    Interchange rows of P: p_(j,:) ↔ p_(i,:)
    for i = j+1 : n do
        ℓ_ij = u_ij / u_jj
        for k = j : n do
            u_ik = u_ik - ℓ_ij u_jk
        end for
    end for
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 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]])
```