Lecture 4: LU factorization, Row pivoting

Jamie Haddock

Table of contents

1 LU Factorization

1.1 Triangular Products

```
using LinearAlgebra
L = tril( rand(1:9,3,3) )
```

```
3×3 Matrix{Int64}:
 6  0  0
 7  3  0
 7  2  5
```

```
U = triu( rand(1:9, 3,3) )
```

```
3×3 Matrix{Int64}:
 4  6  9
 0  6  6
 0  0  9
```

```
L*U
```

```
3×3 Matrix{Int64}:
 24  36   54
 28  60   81
 28  54  120
```

One view of matrix multiplication is as a sum of rank-one matrices formed as *outer products* of corresponding columns of $\mathbf{A}$ and rows of $\mathbf{B}$,

$$\mathbf{C} = \sum_{k=1}^{n} \mathbf{A}_{:k}\mathbf{B}_{k:}.$$

```
L[:,1]*U[1,:]'
```

```
3×3 Matrix{Int64}:
 24  36  54
 28  42  63
 28  42  63
```

Only the first outer product contributes to the first row and column of the product **LU**.

```
L[:,2]*U[2,:]'
```

```
3×3 Matrix{Int64}:
 0   0   0
 0  18  18
 0  12  12
```

```
L[:,3]*U[3,:]'
```

```
3×3 Matrix{Int64}:
 0  0   0
 0  0   0
 0  0  45
```

The triangular zero structures of these matrices create rows and columns of zeros in the inner product.

1.2 Triangular factorization

When factorizing $n \times n$ matrix **A** into a triangular product **LU**, note that **L** and **U** have $n^2 + n > n^2$ entries, so we may choose the diagonal entries of **L** to be one (a **unit lower triangular** matrix).

```
A = [
    2 0 4 3
    -4 5 -7 -10
    1 15 2 -4.5
    -2 0 2 -13
];
L = diagm(ones(4))
U = zeros(4,4);
```

Since $L_{11} = 1$, the first row of **U** is the first row of **A**.

```
U[1,:] = A[1,:]
U
```

```
4×4 Matrix{Float64}:
 2.0  0.0  4.0  3.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
```

The rest of the first column of **L** can be computed from the first column of **A** since only the first outer product contributes to this portion of **A**.

```
L[:,1] = A[:,1]/U[1,1]
L
```

```
4×4 Matrix{Float64}:
  1.0  0.0  0.0  0.0
 -2.0  1.0  0.0  0.0
  0.5  0.0  1.0  0.0
 -1.0  0.0  0.0  1.0
```

```
A  = A  - L[:,1]*U[1,:]'
```

```
4×4 Matrix{Float64}:
 0.0   0.0  0.0    0.0
 0.0   5.0  1.0   -4.0
 0.0  15.0  0.0   -6.0
 0.0   0.0  6.0  -10.0
```

Using the same logic as before, we may set the second rows and columns of $\mathbf{U}$ and $\mathbf{L}$ using $\mathbf{A}_2$.

```
U[2,:] = A [2,:]
L[:,2] = A [:,2]/U[2,2];
```

```
A  = A  - L[:,2]*U[2,:]'
```

```
4×4 Matrix{Float64}:
 0.0  0.0   0.0    0.0
 0.0  0.0   0.0    0.0
 0.0  0.0  -3.0    6.0
 0.0  0.0   6.0  -10.0
```

```
U[3,:] = A [3,:]
L[:,3] = A [:,3]/U[3,3]
A  = A  - L[:,3]*U[3,:]'
```

```
4×4 Matrix{Float64}:
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  2.0
```

```
U[4,4] = A [4,4]

A  - L*U
```

```
4×4 Matrix{Float64}:
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
```

Definition: LU factorization

Given $n \times n$ matrix $\mathbf{A}$, its **LU factorization** is

$$\mathbf{A} = \mathbf{LU},$$

where $\mathbf{L}$ is a unit lower triangular matrix and $\mathbf{U}$ is an upper triangular matrix.

```
"""
    lufact(A)

Compute the LU factorization of square matrix `A`, returning the factors.
"""
function lufact(A)
    n = size(A,1)
    L = diagm(ones(n))  #ones on diagonal, zeros elsewhere
    U = zeros(n,n)
    Ak = float(copy(A))

    #Reduction by outer products
    for k in 1:n-1
        U[k,:] = Ak[k,:]
        L[:,k] = Ak[:,k]/U[k,k]
        Ak -= L[:,k]*U[k,:]'
    end
    U[n,n] = Ak[n,n]
    return LowerTriangular(L),UpperTriangular(U)
end
```

```
lufact
```

1.3 Solving linear systems with LU

We can solve $\mathbf{Ax} = \mathbf{b}$ with three steps:

1. Factor $\mathbf{A} = \mathbf{LU}$.
2. Solve $\mathbf{Lz} = \mathbf{b}$ for $\mathbf{z}$ using forward substitution.
3. Solve $\mathbf{Ux} = \mathbf{z}$ for $\mathbf{x}$ using backward substitution.

> Lemma:
>
> Solving a triangular $n \times n$ system by forward or backward substitution takes $\mathcal{O}(n^2)$ flops asymptotically.

Let $f(n)$ and $g(n)$ be positive-valued functions. We say $f(n) = \mathcal{O}(g(n))$ as $n \to \infty$ if $f(n)/g(n)$ is bounded above as $n \to \infty$.

> Theorem: Efficiency of LU factorization
>
> The LU factorization of an $n \times n$ matrix takes $\mathcal{O}(n^3)$ flops as $n \to \infty$. This dominates the cost of solving an $n \times n$ system.

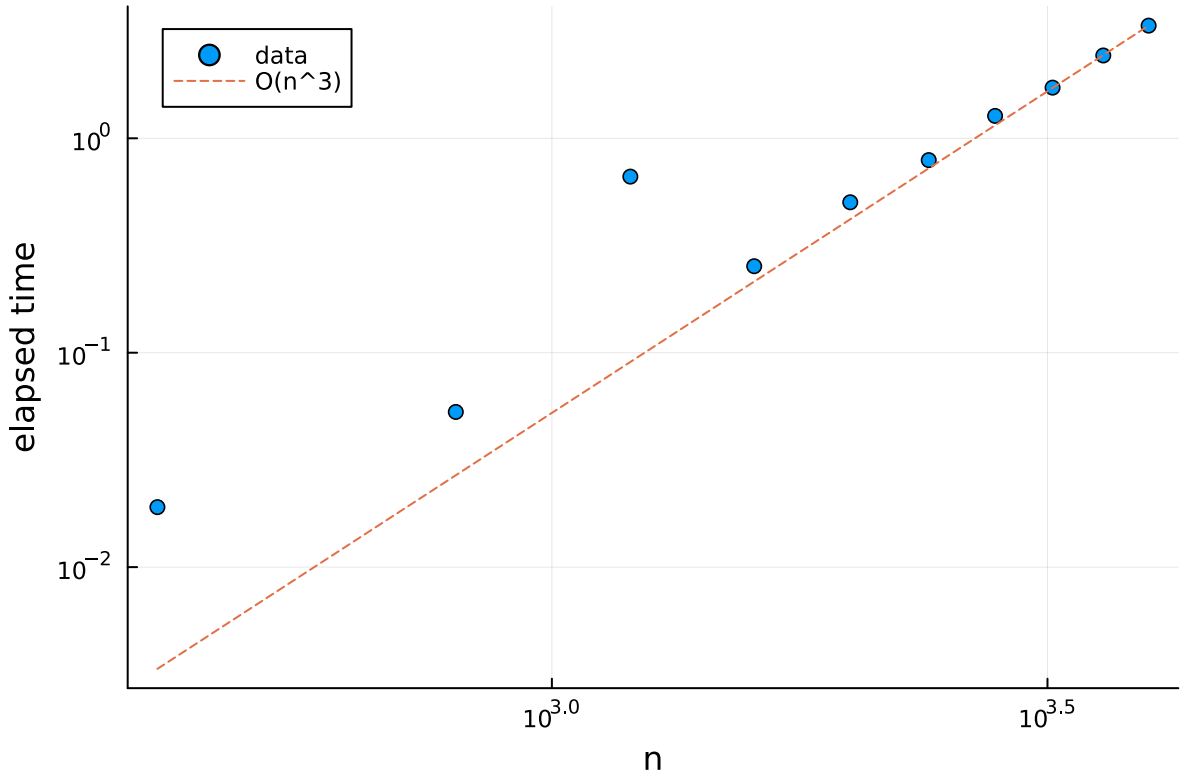
```
lu(randn(3,3)); #throwaway to force compilation of the lu function code_llvm

n = 400:400:4000
t = []
for n in n
    A = randn(n,n)
    time = @elapsed for j in 1:12; lu(A); end
    push!(t,time)
end
```

```
using Plots

scatter(n,t,label="data",legend=:topleft,
    xaxis=(:log10,"n"), yaxis=(:log10,"elapsed time"))
plot!(n,t[end]*(n/n[end]).^3,l=:dash,label="O(n^3)")
```

```
WARNING: using Plots.rotate! in module Main conflicts with an existing identifier.
```



2 Row pivoting

```
A = [2 0 4 3; -4 5 -7 -10; 1 15 2 -4.5; -2 0 2 -13];
L,U = lufact(A);
L
```

```
4×4 LowerTriangular{Float64, Matrix{Float64}}:
  1.0
 -2.0  1.0
  0.5  3.0   1.0
 -1.0  0.0  -2.0  1.0
```

```
A[[2,4],:] = A[[4,2],:]
L,U = lufact(A);
L
```

```
4×4 LowerTriangular{Float64, Matrix{Float64}}:
  1.0
 -1.0  NaN
  0.5   Inf   NaN
```

```
 -2.0   Inf   NaN    1.0
```

After swapping the 2nd and 4th rows of **A**, the matrix is still nonsingular, but now the LU factorization fails due to dividing by a diagonal element of **U** that is zero. The diagonal element of **U** by which we divide is called the **pivot element** of its column.

2.1 Choosing a pivot

In order to avoid a zero pivot in the factorization process, we use a technique known as **row pivoting**: when performing elimination in the jth column, choose as the pivot the element in column j that is largest in absolute value.

```
A  = [2 0 4 3; -2 0 2 -13; 1 15 2 -4.5; -4 5 -7 -10]
```

```
4×4 Matrix{Float64}:
  2.0   0.0   4.0    3.0
 -2.0   0.0   2.0  -13.0
  1.0  15.0   2.0   -4.5
 -4.0   5.0  -7.0  -10.0
```

```
i = argmax( abs.(A [:,1]) )
```

```
4
```

```
L,U = zeros(4,4),zeros(4,4)
U[1,:] = A [i,:]
L[:,1]= A [:,1]/U[1,1]
A  = A  - L[:,1]*U[1,:]'
```

```
4×4 Matrix{Float64}:
 0.0   2.5   0.5   -2.0
 0.0  -2.5   5.5   -8.0
 0.0  16.25  0.25  -7.0
 0.0   0.0   0.0    0.0
```

```
@show i = argmax( abs.(A [:,2]) )
U[2,:] = A [i,:]
L[:,2] = A [:,2]/U[2,2]
A  = A  - L[:,2]*U[2,:]'
```

```
i = argmax(abs.(A [:, 2])) = 3
```

```
4×4 Matrix{Float64}:
 0.0  0.0  0.461538  -0.923077
 0.0  0.0  5.53846   -9.07692
 0.0  0.0  0.0        0.0
 0.0  0.0  0.0        0.0
```

```
@show i = argmax( abs.(A [:,3]) )
U[3,:] = A [i,:]
L[:,3] = A [:,3]/U[3,3]
A  = A  - L[:,3]*U[3,:]'
```

```
i = argmax(abs.(A [:, 3])) = 2
```

```
4×4 Matrix{Float64}:
```

```
 0.0  0.0  0.0  -0.166667
 0.0  0.0  0.0   0.0
 0.0  0.0  0.0   0.0
 0.0  0.0  0.0   0.0
```

```
@show i = argmax( abs.(A [:,4]) )
U[4,:] = A [i,:]
L[:,4] = A [:,4]/U[4,4];
```

```
i = argmax(abs.(A [:, 4])) = 1
```

```
A  - L*U
```

```
4×4 Matrix{Float64}:
 0.0  -1.38778e-16  0.0  0.0
 0.0   1.38778e-16  0.0  0.0
 0.0   0.0          0.0  0.0
 0.0   0.0          0.0  0.0
```

```
U
```

```
4×4 Matrix{Float64}:
 -4.0   5.0   -7.0      -10.0
  0.0  16.25   0.25      -7.0
  0.0   0.0    5.53846   -9.07692
  0.0   0.0    0.0       -0.166667
```

```
L
```

```
4×4 Matrix{Float64}:
 -0.5    0.153846  0.0833333   1.0
  0.5   -0.153846  1.0        -0.0
 -0.25   1.0       0.0        -0.0
  1.0    0.0       0.0        -0.0
```

L doesn't have the required lower triangular structure!

> Theorem: Row pivoting
>
> The row-pivoted LU factorization runs to completion if and only if the original matrix is invertible.

Linear systems with uninvertible matrices have either no solution or infinitely many. We need techniques other than LU factorization to deal with such systems.

2.2 Permutations

The **L** matrix calculated in the last example is not lower-triangular, but is if we simply reverse the rows. In fact, it will be true in general that the **L** calculated will be lower-triangular after a permutation of the rows.

We can think of the algorithm as behaving exactly the original LU factorization technique (without row pivoting) if we reorder the rows of the original matrix in order of the row pivot indices identified in each of the steps. (For our example, this would be putting row 4 at the top, then row 3, then row 2, and then row1.)

```
B = A[ [4,3,2,1], : ]
L, U = lufact(B);
```

```
U
```

```
4×4 UpperTriangular{Float64, Matrix{Float64}}:
 -4.0   5.0   -7.0       -10.0
       16.25   0.25       -7.0
               5.53846    -9.07692
                         -0.166667
```

```
L
```

```
4×4 LowerTriangular{Float64, Matrix{Float64}}:
  1.0
 -0.25   1.0
  0.5   -0.153846  1.0
 -0.5    0.153846  0.0833333  1.0
```

L is the same matrix as before, but with the rows permuted to reverse order!

Theorem: PLU factorization

Given $n \times n$ matrix $\mathbf{A}$, the **PLU factorization** is a unit lower triangular matrix $\mathbf{L}$, an upper triangular matrix $\mathbf{U}$, and a permutation $i_1, \cdots, i_n$ of the integers $1, \cdots, n$ such that

$$\tilde{\mathbf{A}} = \mathbf{LU},$$

where rows $1, \cdots, n$ of $\tilde{\mathbf{A}}$ are rows $i_1, \cdots, i_n$ of $\mathbf{A}$.

```
"""
    plufact(A)

Compute the PLU factorization of square matrix `A`, returning factors and row permutation.
"""
function plufact(A)
    n = size(A,1)
    L,U,p,Ak = zeros(n,n),zeros(n,n),fill(0,n),float(copy(A))

    #Reduction by outer products
    for k in 1:n-1
        p[k] = argmax(abs.(Ak[:,k]))
        U[k,:] = Ak[p[k],:]
        L[:,k] = Ak[:,k]/U[k,k]
        Ak -= L[:,k]*U[k,:]'
    end
    p[n] = argmax(abs.(Ak[:,n]))
    U[n,n] = Ak[p[n],n]
    L[:,n] = Ak[:,n]/U[n,n]
    return LowerTriangular(L[p,:]),U,p
end
```

```
plufact
```

2.3 Linear systems

```
A = rand(1:20,4,4)
L,U,p = plufact(A)
A[p,:] - L*U
```

```
4×4 Matrix{Float64}:
 0.0  0.0   0.0          0.0
 0.0  0.0   0.0          0.0
 0.0  0.0   0.0          0.0
 0.0  0.0  -8.88178e-16  3.55271e-15
```

```
using FundamentalsNumericalComputation;

b = rand(4)
z = FNC.forwardsub(L,b[p])
x = FNC.backsub(U,z)
```

```
4-element Vector{Float64}:
  0.02074712679808451
  0.029105191666686472
 -0.0019461283650047195
 -0.019593097289861985
```

```
b - A*x
```

```
4-element Vector{Float64}:
 1.1102230246251565e-16
 1.1102230246251565e-16
 0.0
 5.551115123125783e-17
```

2.4 Stability

The reason for choosing the largest magnitude pivot during row-pivoting is numerical stability!

Consider the example

$$\mathbf{A} = \begin{bmatrix} -\epsilon & 1 \\ 1 & -1 \end{bmatrix}.$$

```
  = 1e-12
A = [-  1; 1 -1]
b = A*[1,1];
```

```
L,U = lufact(A)
x = FNC.backsub( U, FNC.forwardsub(L,b) )
```

```
2-element Vector{Float64}:
 0.9999778782798785
 1.0
```

We have only five digits of accuracy. This gets even worse if ϵ is smaller!

```
  = 1e-20
A = [-  1; 1 -1]
b = A*[1,1]
L,U = lufact(A)
x = FNC.backsub( U, FNC.forwardsub(L,b) )
```

```
2-element Vector{Float64}:
 -0.0
  1.0
```

This is not due to ill-conditioning of the problem – a solution with PLU factorization works perfectly! (PLU factorization is used under the hood in the \ operator.)

```
A\b
```

```
2-element Vector{Float64}:
 1.0
 1.0
```