II.4 PLU and Cholesky factorisations

In this chapter we consider the following factorisations for square invertible matrices A:

1. The *LU factorisation*:

$A = LU$ where L is lower triangular and U is upper triangular. This is equivalent to Gaussian elimination without pivoting, so may not exist (e.g. if $A[1,1] = 0$).

1. The *PLU factorisation*:

$A = P^\top LU$ where P is a permutation matrix, L is lower triangular and U is upper triangular. This is equivalent to Gaussian elimination with pivoting. It always exists but may in extremely rare cases be unstable. 2. For a real square *symmetric positive definite* ($A \in \mathbb{R}^{n\times n}$ such that $A^\top = A$ and $\mathbf{x}^\top A\mathbf{x} > 0$ for all $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x} \neq 0$) matrix the LU decompostion has a special form which is called the *Cholesky factorisation*: $A = LL^\top$. This provides an algorithmic way to *prove* that a matrix is symmetric positive definite. 3. We also discuss timing and stability of the different factorisations.

In [1]:
```
using LinearAlgebra, Plots, BenchmarkTools
```

1. LU Factorisation

Just as Gram–Schmidt can be reinterpreted as a reduced QR factorisation, Gaussian elimination can be interpreted as an LU factorisation. Write a matrix $A \in \mathbb{C}^{n\times n}$ as follows:

$$A = \begin{bmatrix} \alpha_1 & \mathbf{w}_1^\top \\ \mathbf{v}_1 & A_2 \end{bmatrix}$$

where $\alpha_1 = a_{11}$, $\mathbf{v}_1 = A[2:n, 1]$ and $\mathbf{w}_1 = A[1, 2:n]$. Gaussian elimination consists of taking the first row, dividing by α and subtracting from all other rows. That is equivalent to multiplying by a lower triangular matrix:

$$\begin{bmatrix} 1 & \\ -\mathbf{v}_1/\alpha_1 & I \end{bmatrix} A = \begin{bmatrix} \alpha_1 & \mathbf{w}_1^\top \\ & K - \mathbf{v}_1\mathbf{w}_1^\top/\alpha_1 \end{bmatrix}$$

where $A_2 := K - \mathbf{v}_1\mathbf{w}_1^\top/\alpha_1$ happens to be a rank-1 perturbation of K. We can write this another way:

$$A = \underbrace{\begin{bmatrix} 1 & \\ \mathbf{v}_1/\alpha_1 & I \end{bmatrix}}_{L_1} \begin{bmatrix} \alpha_1 & \mathbf{w}_1^\top \\ & A_2 \end{bmatrix}$$

Now assume we continue this process and manage to deduce $A_2 = L_2U_2$. Then

$$A = L_1 \begin{bmatrix} \alpha_1 & \mathbf{w}_1^\top \\ & L_2U_2 \end{bmatrix} = \underbrace{L_1 \begin{bmatrix} 1 & \\ & L_2 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} \alpha_1 & \mathbf{w}_1^\top \\ & U_2 \end{bmatrix}}_{U}$$

Note we can multiply through to find

$$L = \begin{bmatrix} 1 & \\ \mathbf{v}_1/\alpha_1 & L_2 \end{bmatrix}$$

This procedure implies an algorithm:

Algorithm 1 (LU)

In [2]:
```
function mylu(A)
    n,m = size(A)
    if n ≠ m
        error("Matrix must be square")
    end
    T = eltype(A)
    L = LowerTriangular(zeros(T,n,n))
    U = UpperTriangular(zeros(T,n,n))

    σ = Vector(1:n)

    Aⱼ = copy(A)

    for j = 1:n-1
        α,v,w = Aⱼ[1,1],Aⱼ[2:end,1],Aⱼ[1,2:end]
        K = Aⱼ[2:end,2:end]

        # populate data
        L[j,j] = 1
        L[j+1:end,j] = v/α
        U[j,j] = α
        U[j,j+1:end] = w

        # this is the "recursion": Aⱼ is now the next block
        # We use transpose(w) instead of w' incase w is complex

        Aⱼ = K - v*transpose(w)/α
    end
    # j = n case
    L[n,n] = 1
    U[n,n] = Aⱼ[1,1]

    L,U
end

A = randn(5,5) + 100I # need + 100I so that the matrix is (probably) diagona
L,U = mylu(A)
@test A ≈ L*U
```

Out[2]: **Test Passed**

Example 1 (by-hand)

Consider the matrix

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 4 & 8 \\ 1 & 4 & 9 \end{bmatrix}$$

We write

$$A = \underbrace{\begin{bmatrix} 1 & & \\ 2 & 1 & \\ 1 & & 1 \end{bmatrix}}_{L_1} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 6 \\ 0 & 3 & 8 \end{bmatrix} = L_1 \underbrace{\begin{bmatrix} 1 & & \\ & 1 & \\ & 3/2 & 1 \end{bmatrix}}_{L_2} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 6 \\ 0 & 0 & 5 \end{bmatrix}$$

$$= \underbrace{\begin{bmatrix} 1 & & \\ 2 & 1 & \\ 1 & 3/2 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 6 \\ 0 & 0 & 5 \end{bmatrix}}_{U}$$

2. PLU Factorisation

We learned in first year linear algebra that if a diagonal entry is zero when doing Gaussian elimnation one has to *row pivot*. For stability, in implementation one *always* pivots: swap the largest in magnitude entry for the entry on the diagonal.

We will see this is equivalent to a PLU decomposition:

Theorem 1 (PLU) A matrix $A \in \mathbb{C}^{n\times n}$ is invertible if and only if it has a PLU decomposition:

$$A = P^\top LU$$

where the diagonal of L are all equal to 1 and the diagonal of U are all non-zero.

Proof

If we have a PLU decomposition of this form then L and U are invertible and hence the inverse is simply $A^{-1} = U^{-1}L^{-1}P$.

If $A \in \mathbb{C}^{1\times 1}$ we trivially have an LU decomposition $A = [1] * [a_{11}]$ as all 1×1 matrices are triangular. We now proceed by induction: assume all invertible matrices of lower dimension have a PLU factorisation. As A is invertible not all entries in the first column are zero. Therefore there exists a permutation P_1 so that $\alpha := (P_1A)[1,1] \neq 0$. Hence we write

$$P_1 A = \begin{bmatrix} \alpha & \mathbf{w}^\top \\ \mathbf{v} & K \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & \\ \mathbf{v}/\alpha & I \end{bmatrix}}_{L_1} \begin{bmatrix} \alpha & \mathbf{w}^\top \\ & K - \mathbf{v}\mathbf{w}^\top/\alpha \end{bmatrix}$$

We deduce that $A_2 := K - \mathbf{v}\mathbf{w}^\top/\alpha$ is invertible because A and L_1 are invertible (Exercise).

By assumption we can write $A_2 = P^\top L U$. Thus we have:

$$\begin{aligned} \underbrace{\begin{bmatrix} 1 & \\ & P \end{bmatrix}}_{P} P_1 A &= \begin{bmatrix} 1 & \\ & P \end{bmatrix} \begin{bmatrix} \alpha & \mathbf{w}^\top \\ \mathbf{v} & A_2 \end{bmatrix} = \begin{bmatrix} 1 & \\ & P \end{bmatrix} L_1 \begin{bmatrix} \alpha & \mathbf{w}^\top \\ & P^\top L U \end{bmatrix} \\ &= \begin{bmatrix} 1 & \\ P\mathbf{v}/\alpha & P \end{bmatrix} \begin{bmatrix} 1 & \\ & P^\top L \end{bmatrix} \begin{bmatrix} \alpha & \mathbf{w}^\top \\ & U \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} 1 & \\ P\mathbf{v}/\alpha & L \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} \alpha & \mathbf{w}^\top \\ & U \end{bmatrix}}_{U}. \end{aligned}$$

∎

In the above we neglected to state which permutation is used as for the proof of existence it is immaterial. For *stability* however, we choose one that puts the largest entry: let $\sigma_{\max} : \mathbb{R}^n \to S_n$ be the permutation that swaps the first row with the row of $\mathbf{a}$ whose absolute value is maximised. In cycle notation we then have:

$$\sigma_{\max}(\mathbf{a}) = (1, \operatorname{indmax}|\mathbf{a}|)$$

where indmax gives the index of the entry of a vector which is its maximum.

This inductive proof encodes an algorithm. Note that in the above, just like in Householder QR, P is a product of all permutations that come afterwards, that is, we can think of P as:

$$P = P_{n-1} \cdots P_3 P_2 \qquad \text{for} \qquad P_j = \begin{bmatrix} I_{j-2} & \\ & P_j \end{bmatrix}$$

where P_j is a single permutation corresponding to the first column of A_j. That is, we have

$$P\mathbf{v} = P_{n-1} \cdots P_3 P_2 \mathbf{v}.$$

Algorithm 2 (PLU) This can be implemented in Julia as follows:

In [3]:
```
function σ_max(a)
    n = length(a)
    mx, ind = findmax(abs.(a)) # finds the index of the maximum entry
    if ind == 1
        1:n
```

```
    else
        [ind; 2:ind-1; 1; ind+1:n]
    end
end

function plu(A)
    n,m = size(A)
    if n ≠ m
        error("Matrix must be square")
    end
    T = eltype(A)
    L = LowerTriangular(zeros(T,n,n))
    U = UpperTriangular(zeros(T,n,n))

    σ = Vector(1:n)

    Aⱼ = copy(A)

    for j = 1:n-1
        σ₁ = σ_max(Aⱼ[:,1])
        P₁Aⱼ = Aⱼ[σ₁,:] # permute rows of Aⱼ
        α,v,w = P₁Aⱼ[1,1],P₁Aⱼ[2:end,1],P₁Aⱼ[1,2:end]
        K = P₁Aⱼ[2:end,2:end]

        # populate data
        L[j,j] = 1
        L[j+1:end,j] = v/α
        U[j,j] = α
        U[j,j+1:end] = w

        # apply permutation to previous L
        # and compose the permutations
        L[j:n,1:j-1] = L[(j:n)[σ₁],1:j-1]
        σ[j:n] = σ[j:n][σ₁]

        # this is the "recursion": Aⱼ is now the next block
        # We use transpose(w) instead of w' incase w is complex
        Aⱼ = K - v*transpose(w)/α
    end
    # j = n case
    L[n,n] = 1
    U[n,n] = Aⱼ[1,1]

    L,U,σ
end

A = randn(5,5)
L,U,σ = plu(A)
@test L*U ≈ A[σ,:]
```

Out[3]: **Test Passed**

Example 2

Again we consider the matrix

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 4 & 8 \\ 1 & 4 & 9 \end{bmatrix}$$

Even though $a_{11} = 1 \neq 0$, we still pivot: placing the maximum entry on the diagonal to mitigate numerical errors. That is, we first pivot and upper triangularise the first column:

$$\underbrace{\begin{bmatrix} 0 & 1 & \\ 1 & 0 & \\ & & 1 \end{bmatrix}}_{P_1} A = \begin{bmatrix} 2 & 4 & 8 \\ 1 & 1 & 1 \\ 1 & 4 & 9 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & & \\ 1/2 & 1 & \\ 1/2 & & 1 \end{bmatrix}}_{L_1} \begin{bmatrix} 2 & 4 & 8 \\ & -1 & -3 \\ & 2 & 5 \end{bmatrix}$$

That is we have $\alpha_1 = 2$, $\mathbf{v}_1 = [1, 1]$, and $\mathbf{w}_1 = [4, 8]$. We now pivot for A_2:

$$\underbrace{\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}}_{P_2} \underbrace{\begin{bmatrix} -1 & -3 \\ 2 & 5 \end{bmatrix}}_{A_2} = \begin{bmatrix} 2 & 5 \\ -1 & -3 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & \\ -1/2 & 1 \end{bmatrix}}_{L_2} \begin{bmatrix} 2 & 5 \\ & -\frac{1}{2} \end{bmatrix}$$

Note that $P_2 \mathbf{v}_1 = \mathbf{v}_1$ and

$$P = P_2 P_1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

Hence we have

$$PA = \begin{bmatrix} 1 & & \\ 1/2 & 1 & \\ 1/2 & -1/2 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & 8 \\ & 2 & 5 \\ & & -1/2 \end{bmatrix}$$

We see how this example is done on a computer:

In [4]:
```
A = [1 1 1;
     2 4 8;
     1 4 9]
L,U,σ = lu(A) # σ is a vector encoding the permutation

@test L == [1    0   0 ;
            1/2  1   0 ;
            1/2 -1/2 1 ]
@test U == [2 4 8   ;
            0 2 5   ;
            0 0 -1/2]
@test I(3)[σ,:] == [0 1 0 ;
                    0 0 1 ;
                    1 0 0 ]
```

Out[4]: **Test Passed**

To invert a system we can do:

In [5]:
```
b = randn(3)
@test U\(L\b[σ]) == A\b
```

Out[5]: **Test Passed**

Note the entries match exactly because this is precisely what `\` is using.

3. Cholesky Factorisation

Cholesky Factorisation is a form of Gaussian elimination (without pivoting) that exploits symmetry in the problem, resulting in a substantial speedup. It is only relevant for *symmetric positive definite* (SPD) matrices.

Definition 1 (positive definite) A square matrix $A \in \mathbb{R}^{n\times n}$ is *positive definite* if for all $\mathbf{x} \in \mathbb{R}^n, x \neq 0$ we have

$$\mathbf{x}^\top A\mathbf{x} > 0$$

First we establish some basic properties of positive definite matrices:

Proposition 3 (conj. pos. def.) If $A \in \mathbb{R}^{n\times n}$ is positive definite and $V \in \mathbb{R}^{n\times n}$ is non-singular then

$$V^\top AV$$

is positive definite.

Proposition 4 (diag positivity) If $A \in \mathbb{R}^{n\times n}$ is positive definite then its diagonal entries are positive: $a_{kk} > 0$.

Theorem 1 (subslice pos. def.) If $A \in \mathbb{R}^{n\times n}$ is positive definite and $\mathbf{k} \in \{1, \dots, n\}^m$ is a vector of m integers where any integer appears only once, then $A[\mathbf{k}, \mathbf{k}] \in \mathbb{R}^{m\times m}$ is also positive definite.

We leave the proofs to the problem sheets. Here is the key result:

Theorem 2 (Cholesky and SPD) A matrix A is symmetric positive definite if and only if it has a Cholesky factorisation

$$A = LL^\top$$

where the diagonals of L are positive.

Proof If A has a Cholesky factorisation it is symmetric ($A^\top = (LL^\top)^\top = A$) and for $\mathbf{x} \neq 0$ we have

$$\mathbf{x}^\top A\mathbf{x} = (L\mathbf{x})^\top L\mathbf{x} = \|L\mathbf{x}\|^2 > 0$$

where we use the fact that L is non-singular.

For the other direction we will prove it by induction, with the 1×1 case being trivial. Assume all lower dimensional symmetric positive definite matrices have Cholesky decompositions. Write

$$A = \begin{bmatrix} \alpha & \mathbf{v}^\top \\ \mathbf{v} & K \end{bmatrix} = \underbrace{\begin{bmatrix} \sqrt{\alpha} & \\ \frac{\mathbf{v}}{\sqrt{\alpha}} & I \end{bmatrix}}_{L_1} \underbrace{\begin{bmatrix} 1 & \\ & K - \frac{\mathbf{v}\mathbf{v}^\top}{\alpha} \end{bmatrix}}_{A_1} \underbrace{\begin{bmatrix} \sqrt{\alpha} & \frac{\mathbf{v}^\top}{\sqrt{\alpha}} \\ & I \end{bmatrix}}_{L_1^\top}.$$

Note that $A_2 := K - \frac{\mathbf{v}\mathbf{v}^\top}{\alpha}$ is a subslice of $L_1^{-1} A L_1^{-\top}$, hence by the previous propositions is itself symmetric positive definite. Thus we can write

$$A_2 = K - \frac{\mathbf{v}\mathbf{v}^\top}{\alpha} = LL^\top$$

and hence $A = LL^\top$ for

$$L = L_1 \begin{bmatrix} 1 & \\ & L \end{bmatrix} = \begin{bmatrix} \sqrt{\alpha} & \\ \frac{\mathbf{v}}{\sqrt{\alpha}} & L \end{bmatrix}$$

satisfies $A = LL^\top$. ∎

Note hidden in this proof is a simple algorithm form computing the Cholesky factorisation.

Algorithm 3 (Cholesky)

In [6]:
```
function mycholesky(A)
    T = eltype(A)
    n,m = size(A)
    if n ≠ m
        error("Matrix must be square")
    end
    if A ≠ A'
        error("Matrix must be symmetric")
    end
    T = eltype(A)
    L = LowerTriangular(zeros(T,n,n))
    Aⱼ = copy(A)
    for j = 1:n
        α,v = Aⱼ[1,1],Aⱼ[2:end,1]
        if α ≤ 0
            error("Matrix is not SPD")
        end
        L[j,j] = sqrt(α)
        L[j+1:end,j] = v/sqrt(α)

        # induction part
        Aⱼ = Aⱼ[2:end,2:end] - v*v'/α
    end
    L
end
```

```
A = Symmetric(rand(100,100) + 100I)
L = mycholesky(A)
@test A ≈ L*L'
```

Out[6]: **Test Passed**

This algorithm succeeds if and only if A is symmetric positive definite.

Example 3 (Cholesky by hand) Consider the matrix

$$A = \begin{bmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix}$$

Then $\alpha_1 = 2$, $\mathbf{v}_1 = [1, 1, 1]$, and

$$A_2 = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} - \frac{1}{2}\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}\begin{bmatrix} 1 & 1 & 1 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 3 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 3 \end{bmatrix}.$$

Continuing, we have $\alpha_2 = 3/2$, $\mathbf{v}_2 = [1/2, 1/2]$, and

$$A_3 = \frac{1}{2}\left(\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} - \frac{1}{3}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\begin{bmatrix} 1 & 1 \end{bmatrix}\right) = \frac{1}{3}\begin{bmatrix} 4 & 1 \\ 1 & 4 \end{bmatrix}$$

Next, $\alpha_3 = 4/3$, $\mathbf{v}_3 = [1]$, and

$$A_4 = [4/3 - 3/4 * (1/3)^2] = [5/4]$$

i.e. $\alpha_4 = 5/4$.

Thus we get

$$L = \begin{bmatrix} \sqrt{\alpha_1} & & & \\ \frac{\mathbf{v}_1[1]}{\sqrt{\alpha_1}} & \sqrt{\alpha_2} & & \\ \frac{\mathbf{v}_1[2]}{\sqrt{\alpha_1}} & \frac{\mathbf{v}_2[1]}{\sqrt{\alpha_2}} & \sqrt{\alpha_3} & \\ \frac{\mathbf{v}_1[3]}{\sqrt{\alpha_1}} & \frac{\mathbf{v}_2[2]}{\sqrt{\alpha_2}} & \frac{\mathbf{v}_3[1]}{\sqrt{\alpha_3}} & \sqrt{\alpha_4} \end{bmatrix} = \begin{bmatrix} \sqrt{2} & & & \\ \frac{1}{\sqrt{2}} & \sqrt{\frac{3}{2}} & & \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} & \frac{2}{\sqrt{3}} & \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{12}} & \frac{\sqrt{5}}{2} \end{bmatrix}$$

We can check if our answer is correct:

In [7]:
```
A = ones(4,4) + I
# The inbuilt cholesky returns a special type whose field L is the factor
@test cholesky(A).L ≈ [  sqrt(2)   0         0           0;
                         1/sqrt(2) sqrt(3/2) 0           0;
                         1/sqrt(2) 1/sqrt(6) 2/sqrt(3)   0;
                         1/sqrt(2) 1/sqrt(6) 1/sqrt(12)  sqrt(5)/2]
```

Out[7]: **Test Passed**

4. Timings and Stability

The different factorisations have trade-offs between speed and stability. First we compare the speed of the different factorisations on a symmetric positive definite matrix, from fastest to slowest:

In [8]:
```
n = 100
A = Symmetric(rand(n,n)) + 100I # shift by 10 ensures positivity
@btime cholesky(A);
@btime lu(A);
@btime qr(A);
```

```
  237.748 μs (3 allocations: 78.20 KiB)
  67.238 μs (4 allocations: 79.08 KiB)
  219.580 μs (7 allocations: 134.55 KiB)
```

On my machine, `cholesky` is ~1.5x faster than `lu`,
which is ~2x faster than QR.

In terms of stability, QR computed with Householder reflections (and Cholesky for positive definite matrices) are stable, whereas LU is usually unstable (unless the matrix is diagonally dominant). PLU is a very complicated story: in theory it is unstable, but the set of matrices for which it is unstable is extremely small, so small one does not normally run into them.

Here is an example matrix that is in this set.

In [9]:
```
function badmatrix(n)
    A = Matrix(1I, n, n)
    A[:,end] .= 1
    for j = 1:n-1
        A[j+1:end,j] .= -1
    end
    A
end
A =1badmatrix(5)
```

Out[9]:
```
5×5 Matrix{Int64}:
  1   0   0   0  1
 -1   1   0   0  1
 -1  -1   1   0  1
 -1  -1  -1   1  1
 -1  -1  -1  -1  1
```

Note that pivoting will not occur (we do not pivot as the entries below the diagonal are the same magnitude as the diagonal), thus the PLU Factorisation is equivalent to an LU factorisation:

In [10]:
```
L,U = lu(A)
```

Out[10]:
```
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
5×5 Matrix{Float64}:
  1.0   0.0   0.0   0.0  0.0
 -1.0   1.0   0.0   0.0  0.0
 -1.0  -1.0   1.0   0.0  0.0
 -1.0  -1.0  -1.0   1.0  0.0
 -1.0  -1.0  -1.0  -1.0  1.0
U factor:
5×5 Matrix{Float64}:
 1.0  0.0  0.0  0.0   1.0
 0.0  1.0  0.0  0.0   2.0
 0.0  0.0  1.0  0.0   4.0
 0.0  0.0  0.0  1.0   8.0
 0.0  0.0  0.0  0.0  16.0
```

But here we see an issue: the last column of `U` is growing exponentially fast! Thus when `n` is large we get very large errors:

In [11]:
```
n = 100
b = randn(n)
A = badmatrix(n)
norm(A\b - qr(A)\b) # A \ b still uses lu
```

Out[11]:
```
4.162792838983333
```

Note `qr` is completely fine:

In [12]:
```
norm(qr(A)\b - qr(big.(A)) \b) # roughly machine precision
```

Out[12]:
```
8.153023773640683532197510463047686204980820308149611625292169906520130876318862e-15
```

Amazingly, PLU is fine if applied to a small perturbation of `A`:

In [13]:
```
ε = 0.000001
Aε = A .+ ε .* randn.()
norm(Aε \ b - qr(Aε) \ b) # Now it matches!
```

Out[13]:
```
9.365183599121149e-15
```

The big *open problem* in numerical linear algebra is to prove that the set of matrices for which PLU fails has extremely small measure.