Click here to view the PDF version.

```
begin
    using LinearAlgebra
    using SparseArrays
    using PlutoUI
    using PlutoTeachingTools
    using HypertextLiteral: @htl, @htl_str
    using Plots
end
```

Table of Contents

Direct methods for linear systems

In the previous chapter on polynomial interpolation we were already confronted with the need to solve linear systems, that is a system of equations of the form

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{1}$$

where we are given a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ as well as a right-hand side $\mathbf{b} \in \mathbb{R}^n$. As the solution we seek the unknown $\mathbf{x} \in \mathbb{R}^n$.

A difficult example

Solving linear equations is a standard exercise in linear algebra and you probably have already done it in previous courses. However, you might also know that solving such linear systems is not always equally easy.

Let us consider a family of innocent-looking matrices, which are famously ill-conditioned, the **Hilbert matrices**. Here we show the 10 by 10 case. Feel free to increase the `nmax` to make the problem even more challenging:

- `nmax` = 10

```
M_difficult =
10×10 Matrix{Float64}:
 0.5        0.333333   0.25       0.2        …  0.111111   0.1        0.0909091
 0.333333   0.25       0.2        0.166667      0.1        0.0909091  0.0833333
 0.25       0.2        0.166667   0.142857      0.0909091  0.0833333  0.0769231
 0.2        0.166667   0.142857   0.125         0.0833333  0.0769231  0.0714286
 0.166667   0.142857   0.125      0.111111      0.0769231  0.0714286  0.0666667
 0.142857   0.125      0.111111   0.1        …  0.0714286  0.0666667  0.0625
 0.125      0.111111   0.1        0.0909091     0.0666667  0.0625     0.0588235
 0.111111   0.1        0.0909091  0.0833333     0.0625     0.0588235  0.0555556
 0.1        0.0909091  0.0833333  0.0769231     0.0588235  0.0555556  0.0526316
 0.0909091  0.0833333  0.0769231  0.0714286     0.0555556  0.0526316  0.05
```

```
1 M_difficult = [ 1/(i+j) for i=1:nmax, j=1:nmax ]
```

We take the simple right-hand side of all ones:

```
b_difficult = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

```
1 b_difficult = ones(nmax)
```

And solve the system using `\`:

```
x_difficult =
▸ [-110.043, 5942.0, -1.02991e5, 841063.0, -3.78469e6, 1.00923e7, -1.63396e7, 1.57558e7, -8
```

```
1 x_difficult = @time M_difficult \ b_difficult
```

```
  0.000076 seconds (6 allocations: 1.203 KiB)
```

Looks like a reasonable answer, **but is it ?**

Let's check against a computation using Julia's `BigFloat` number. This is usually between 10 and 100 times more expensive, so not useful for practical computations. But it will give us a reference answer to much higher precision.

```
▸ [-110.0, 5940.0, -102960.0, 840840.0, -3.78378e+06, 1.00901e+07, -1.63363e+07, 1.57529e+0
```

```
1 begin
2     M_big = [ 1/(big(i)+big(j)) for i=1:nmax, j=1:nmax ]
3     x_big = @time M_big \ b_difficult
4 end
```

```
  0.007060 seconds (1.01 k allocations: 91.328 KiB, 97.24% compilation tim
e)
```

Looking at the second entry we already see some **significant deviations in the standard `Float64` answer**, which actually get way worse as we increase `nmax`:

```
▸ [0.0429388, -2.00331, 30.5705, -222.996, 906.665, -2205.02, 3281.26, -2927.26, 1437.35, -2
```

```
1 x_big - x_difficult
```

While for `nmax = 5` the answer still kind of ok, **the result of the standard `\`-operator become numerical garbage** from `nmax = 12` onwards.

The related questions we want to ask here are:

- How does Julia's `\`-operator work ?
- How can we quantify when a linear system is more challenging to solve than another ?
- Based on this: When can we trust the results we get ?

This is what we want to explore in this part of the course.

Motivation: The \ (backslash) operator

For solving linear systems, we so far employed Julia's backslash `\` operator. So let's find out what it actually does under the hood. We take the problem

```
begin
    A = Float64[-4  3 -1;
                2  1  0;
                4 -3  4]
    b = [2, 4, -2]
end;
```

as an example. Normally we would now just do `A \ b`, so let's check what this calls

```
# 1 method for generic function "\" from [90mBase [39m:
```

- \(A::**AbstractMatrix**, B::**AbstractVecOrMat**) in LinearAlgebra at /opt/hostedtoolcache/julia/1.12.4/x64/share/julia/stdlib/v1.12/LinearAlgebra/src/generic.jl:1220

```
methods(\, (Matrix, Vector))
```

Ok, so this calls into Julia's linear algebra library. The code is located here. Essentially it performs an **LU factorisation** for square matrices:

```
LU = LinearAlgebra.LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
3×3 Matrix{Float64}:
  1.0  0.0  0.0
 -0.5  1.0  0.0
 -1.0  0.0  1.0
U factor:
3×3 Matrix{Float64}:
 -4.0  3.0  -1.0
  0.0  2.5  -0.5
  0.0  0.0   3.0
```

```
LU = lu(A)
```

which as we can see *factorises* the matrix $\mathbf{A}$ into a **lower triangular** matrix $\mathbf{L}$ and an **upper triangular** matrix $\mathbf{U}$. More formally:

Definition: LU factorisation

Given a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ find a lower triangular matrix $\mathbf{L} \in \mathbb{R}^{n \times n}$ and an upper triangular matrix $\mathbf{U} \in \mathbb{R}^{n \times n}$, such that

$$\mathbf{A} = \mathbf{LU} \tag{2}$$

We can easily check that by multiplying $\mathbf{L}$ and $\mathbf{U}$ we indeed revover $\mathbf{A}$:

```
3×3 Matrix{Float64}:
 -4.0   3.0  -1.0
  2.0   1.0   0.0
  4.0  -3.0   4.0
```

```
LU.L * LU.U
```

```
3×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0
```

```
A - LU.L * LU.U
```

Now we are a step closer to what happens, but why is this useful ?

Solving triangular systems

The factorisation into two triangular matrices is useful, because it is especially easy to solve a system where the matrix is triangular. For example consider the *lower* triangular system

$$\begin{pmatrix} 4 & 0 & 0 & 0 \\ 3 & -1 & 0 & 0 \\ -1 & 0 & 3 & 0 \\ 1 & -1 & -1 & 2 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 8 \\ 5 \\ 0 \\ 1 \end{pmatrix}.$$

- The first row of this system simply states $4x_1 = 8$, which is very easy to solve, namely $x_1 = 8/4 = 2$.
- The second row states $3x_1 - x_2 = 5$. However, x_1 is already known and can be inserted to find $x_2 = -(5 - 3 \cdot 2) = 1$.
- Following the same idea the third row gives $x_3 = (0 + 1 \cdot 2)/3 = 2/3$ and the last row $x_4 = (1 - 1 \cdot 2 + 1 \cdot 1 + 1 \cdot 2/3)/2 = 1/3$.
- In total we found

$$\mathbf{x} = \begin{pmatrix} 2 \\ 1 \\ 2/3 \\ 1/3 \end{pmatrix}.$$

Generalising to an arbitrary 4x4 lower-rectangular matrix

$$\begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix} \mathbf{x} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

this solution algorithm can be expressed as

$$\begin{aligned} x_1 &= \frac{b_1}{L_{11}}, \\ x_2 &= \frac{b_2 - L_{21}x_1}{L_{22}}, \\ x_3 &= \frac{b_3 - L_{31}x_1 - L_{32}x_2}{L_{33}}, \\ x_4 &= \frac{b_4 - L_{41}x_1 - L_{42}x_2 - L_{43}x_3}{L_{44}}. \end{aligned} \tag{3}$$

We obtain

Algorithm 1: Forward substitution

Given a lower-triangular matrix $\mathbf{L} \in \mathbb{R}^{n \times n}$ a linear system $\mathbf{Lx} = \mathbf{b}$ can be solved by looping from $i = 1, \dots n$ and computing

$$x_i = \frac{1}{L_{ii}} \left(b_i - \sum_{j=1}^{i-1} L_{ij} x_j \right)$$

or in form of an implementation

```
forward_substitution (generic function with 1 method)
```

```julia
function forward_substitution(L, b)
    n = size(L, 1)
    x = zeros(n)
    x[1] = b[1] / L[1, 1]
    for i in 2:n
        row_sum = 0.0
        for j in 1:i-1
            row_sum += L[i, j] * x[j]
        end

        x[i] = 1 / L[i, i] * (b[i] - row_sum)
    end
    x
end
```

▶ **Teacher hint: Live coding**

For upper triangular matrices we solve linear systems proceeds following the same idea — just in this case we start from the last row and not the first. For example a system

$$\begin{pmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{pmatrix} \mathbf{x} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

we solve by starting with x_4 and working our way forward to x_1:

$$\begin{aligned} x_4 &= \frac{b_4}{U_{44}}, \\ x_3 &= \frac{b_3 - U_{34}x_4}{U_{33}}, \\ x_2 &= \frac{b_2 - U_{23}x_3 - U_{24}x_4}{U_{22}}, \\ x_1 &= \frac{b_1 - U_{12}x_2 - U_{13}x_3 - U_{14}x_4}{U_{11}}. \end{aligned} \tag{4}$$

More formally this algorithm is called

Algorithm 2: Backward substitution

Given an upper-triangular matrix $\mathbf{U} \in \mathbb{R}^{n \times n}$ a linear system $\mathbf{U}\mathbf{x} = \mathbf{b}$ can be solved by looping in reverse order $i = n, \ldots, 1$ and computing

$$x_i = \frac{1}{U_{ii}} \left(b_i - \sum_{j=i+1}^{n} U_{ij} x_j \right)$$

or in code

```
backward_substitution (generic function with 1 method)
```

```
function backward_substitution(U, b)
    n = size(U, 1)
    x = zeros(n)
    x[n] = b[n] / U[n, n]
    for i in n-1:-1:1  # Note that this loop goes *backwards* n-1, n-2, ..., 1
        row_sum = 0.0
        for j in i+1:n
            row_sum += U[i, j] * x[j]
        end
        x[i] = (b[i] - row_sum) / U[i, i]
    end
    x
end
```

Based on our discussion we now understand why it is advantageous to perform an LU factorisation when solving a linear system: For both the resulting triangular matrix $\mathbf{L}$ as well as the matrix $\mathbf{U}$, simple solution algorithms are available. In summary we thus obtain

Algorithm 3: Solving linear systems by LU factorisation / Gaussian elimination

We are given a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, a right-hand side $\mathbf{b} \in \mathbb{R}^n$. This algorithm computes the solution $\mathbf{x} \in \mathbb{R}^n$ to $\mathbf{A}\mathbf{x} = \mathbf{b}$:

1. Factorise $\mathbf{A} = \mathbf{L}\mathbf{U}$.
2. Solve $\mathbf{L}\mathbf{z} = \mathbf{b}$ for $\mathbf{z}$ using **forward substitution** (Algorithm 1).
3. Solve $\mathbf{U}\mathbf{x} = \mathbf{z}$ for $\mathbf{x}$ using **backward substitution** (Algorithm 2).

A few remarks are in order:

- We have so far not discussed how to even compute the LU factorisation. As we will see in the next section this step will actually be accomplished by the **Gaussian elimination algorithm**, which probably already know from your linear algebra lectures.
- A severe **drawback of this naive algorithm** is immediately apparent from our 4x4 example problem, where (3) and (4) provide explicit solution expressions. If by any chance an **element**

L_{ii} **or** U_{ii} **happens to be zero**, the **algorithm cannot work**. So one needs to come up with ways around this. We will mention a few ideas in the section on **pivoting**.

- Note, that **steps 2. and 3.** of the algorithm **do not depend on the original matrix** $\mathbf{A}$, but only on the right hand side $\mathbf{b}$. In other words once the factorisation $\mathbf{A} = \mathbf{LU}$ has been computed, solving $\mathbf{Ax} = \mathbf{b}$ for different right hand sides $\mathbf{b}$ only requires the execution of steps 2. and 3. As we will see later, step 1. is the most expensive step. Therefore **once** $\mathbf{A}$ **has been factorised**, the **cost for solving an *arbitrary* linear system** involving $\mathbf{A}$ **is reduced** as its factorised form $\mathbf{LU}$ can be used in its place.

LU factorisation

The missing piece in our discussion is how the LU factorisation can be computed. As it turns out this is the Gaussian elimination algorithm, which you already learned in previous linear algebra classes. Indeed, this algorithm reduces a matrix to upper triangular form — now we just need to be careful with the book-keeping to also extract the factor $\mathbf{L}$.

Algorithm 4: LU factorisation

Input: $\mathbf{A} \in \mathbb{R}^{n \times n}$

Output: $\mathbf{U} \in \mathbb{R}^{n \times n}$, $\mathbf{L} \in \mathbb{R}^{n \times n}$

- $\mathbf{U} = \mathbf{A}$
- for $k = 1, \dots, n-1$ *(algorithm steps)*
  - $L_{kk} = 1$
  - for $i = k+1, \dots, n$ *(Loop over rows)*
    - $L_{ik} = \frac{U_{ik}}{U_{kk}}$
    - for $j = k, \dots n$ *(Loop over columns)*
      - $U_{ij} \leftarrow U_{ij} - L_{ik}U_{kj}$ *(modify* U_{ij} *to* $U_{ij} - L_{ik}U_{kj}$*)*
- $L_{nn} = 1$ *(the loop above only runs until* $n-1$*)*

Example: Manual LU factorisation

We will run this algorithm manually for solving the linear system

$$\underbrace{\begin{pmatrix} 2 & 1 & 0 \\ -4 & 3 & -1 \\ 4 & -3 & 4 \end{pmatrix}}_{=\mathbf{A}} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{=\mathbf{x}} = \underbrace{\begin{pmatrix} 4 \\ 2 \\ -2 \end{pmatrix}}_{=\mathbf{b}}.$$

That is for factorising the matrix $\mathbf{A}$. Before we start the loop over `k`, the matrix $\mathbf{L}$ is empty and $\mathbf{U}$ is just equal to $\mathbf{A}$:

$$\mathbf{L} = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix} \qquad \mathbf{U} = \begin{pmatrix} \mathbf{2} & 1 & 0 \\ -4 & 3 & -1 \\ 4 & -3 & 4 \end{pmatrix}$$

- `k=1` **(Step 1):** In Gaussian elimination we would first zero out the **2nd and 3rd row** of the **1st column** of matrix $\mathbf{A}$. Here we do the same in a loop over rows starting at `k+1 = 2`.
  - `i = 2` **(Row 2):** To zero out the first entry of the second row by subtraction, we need to multiply the first row with this factor:
    - $L_{21} = \frac{U_{21}}{U_{11}} = \frac{-4}{2} = -2$
    - The *loop over columns* now just uses this factor to update the second row by subtracting $L_{21} = -2$ times the first — or equivalently adding 2 times the first.
    - After this step we have:

$$\mathbf{L} = \begin{pmatrix} 1 & & \\ \mathbf{-2} & & \\ & & \end{pmatrix} \qquad \mathbf{U} = \begin{pmatrix} \mathbf{2} & 1 & 0 \\ \mathbf{0} & \mathbf{5} & \mathbf{-1} \\ 4 & -3 & 4 \end{pmatrix}$$

      where **bold** highlights the elements, that have changed.
  - `i = 3` **(Row 3):** Here we zero out U_{31}. We determine the factor
    - $L_{31} = \frac{U_{31}}{U_{11}} = \frac{4}{2} = 2$
    - The *loop over columns* updates the third row by subtracting $L_{31} = 2$ times the first row.
    - After this step:

$$\mathbf{L} = \begin{pmatrix} 1 & & \\ -2 & & \\ \mathbf{2} & & \end{pmatrix} \qquad \mathbf{U} = \begin{pmatrix} \mathbf{2} & 1 & 0 \\ 0 & 5 & -1 \\ \mathbf{0} & \mathbf{-5} & \mathbf{4} \end{pmatrix}$$

    - In *loop over rows* `i` only runs until `n = 3`, so we are done with it.
- `k=2` **(Step 2):** Our goal is now to zero out the 2nd column in all rows below the diagonal. We thus start another loop over rows, this time starting at `k+1 = 3`:

- `i = 3` **(Row 3):** Our factor is now
  - $L_{32} = \frac{U_{31}}{U_{22}} = \frac{-5}{5} = -1$
  - After subtracting $L_{32} = -1$ times the second row from the 3rd in the *loop over columns*, i.e. we add 2nd and 3rd row, we get

$$\mathbf{L} = \begin{pmatrix} 1 & & \\ -2 & 1 & \\ 2 & \mathbf{-1} & \end{pmatrix} \qquad \mathbf{U} = \begin{pmatrix} 2 & 1 & 0 \\ 0 & \mathbf{5} & -1 \\ 0 & \mathbf{0} & \mathbf{3} \end{pmatrix}$$

  - Again we have reached the end of the *loop over rows* as `i = n = 3`.
- Since `k = n-1 = 2` we also reached the end of the loop over `k`
- Finally we set the missing $L_{33} = 1$ to obtain the final result:

$$\mathbf{L} = \begin{pmatrix} 1 & & \\ -2 & 1 & \\ 2 & -1 & \mathbf{1} \end{pmatrix} \qquad \mathbf{U} = \begin{pmatrix} 2 & 1 & 0 \\ 0 & 5 & -1 \\ 0 & 0 & 3 \end{pmatrix}$$

This is the LU factorisation of $\mathbf{A}$, which is easily verified by multiplying out the matrices:

$$\mathbf{LU} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 2 & -1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 0 \\ 0 & 5 & -1 \\ 0 & 0 & 3 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 0 \\ -4 & 3 & -1 \\ 4 & -3 & 4 \end{pmatrix} = \mathbf{A}$$

▸ Expand for an explict Gaussian elimination procedure, which works directly working on the equations in x

We see that indeed the LU factorisation algorithm can be seen as a formalisation of the Gaussian elimination procedure, reducing the matrix to triangular form $\mathbf{U}$.

Finally, for completeness, we provide a Julia implementation of LU factorisation (Algorithm 4):

```
factorise_lu (generic function with 1 method)
```

```
function factorise_lu(A)
    n = size(A, 1)
    L = zeros(n, n)      # Initialise L and U by zeros
    U = float(copy(A))   # Make a copy of A and ensure that all entries
                         # are converted to floating-point numbers

    for k in 1:n-1           # Algorithm steps
        L[k, k] = 1.0
        for i in k+1:n       # Loop over rows
            L[i, k] = U[i, k] / U[k, k]
            for j in k:n     # Loop over columns
                U[i, j] = U[i, j] - L[i, k] * U[k, j]  # Update U in-place
            end
        end
    end
    L[n, n] = 1.0            # Since the loop only runs until n-1

    # Return L and U using Julia datastructures to indicate
    # their special lower-triangular and upper-triangular form.
    return LowerTriangular(L), UpperTriangular(U)
end
```

We stay with the example where we performed manual Gaussian elimination:

```
A_manual = 3×3 Matrix{Float64}:
  2.0   1.0   0.0
 -4.0   3.0  -1.0
  4.0  -3.0   4.0
```

```
A_manual = Float64[ 2  1  0;
                   -4  3 -1;
                    4 -3  4]
```

With this slider you can stop `factorise_lu` after 1, 2 or 3 steps, checking that agrees with the steps we computed manually. The matrices displayed below show the situation after `k = nstep_lu_A` in **Algorithm 4** has finished.

- `nstep_lu_A =`  0

```
▸ (U = 3×3 Matrix{Float64}:, L = 3×3 Matrix{Float64}:)
```

Running Algorithm 3

With this we have the missing ingredient to run Algorithm 3 and numerically solve a linear system.

We stay with the problem

$$\underbrace{\begin{pmatrix} 2 & 1 & 0 \\ -4 & 3 & -1 \\ 4 & -3 & 4 \end{pmatrix}}_{=\mathbf{A}_{\text{manuel}}} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{=\mathbf{x}} = \underbrace{\begin{pmatrix} 4 \\ 2 \\ -2 \end{pmatrix}}_{=\mathbf{b}_{\text{manuel}}}$$

which we solved manually beforehand:

```
3×3 Matrix{Float64}:
  2.0   1.0   0.0
 -4.0   3.0  -1.0
  4.0  -3.0   4.0
```

```
A_manual
```

```
b_manual = [4, 2, -2]
```

```
b_manual = [4, 2, -2]
```

We follow **Algorithm 3**. First we need to find the LU factorisation:

```
(3×3 LowerTriangular{Float64, Matrix{Float64}}:, 3×3 UpperTriangular{Float64, Matrix{Fl
   1.0    •    •                                  2.0  1.0   0.0
```

```
L, U = factorise_lu(A_manual)
```

which agrees what we obtained manually. Next we forward substitute:

```
z = [4.0, 10.0, 0.0]
```

```
z = forward_substitution(L, b_manual)
```

Finally we backward substitute:

```
x = [1.0, 2.0, 0.0]
```

```
x = backward_substitution(U, z)
```

and check the result:

```
[0.0, 0.0, 0.0]
```

```
A * x - b
```

Hooray ! This suceeded!

Unfortunately **this simple algorithm does not always work**.

Consider the matrix:

```
D = 3×3 Matrix{Int64}:
     1  2  3
     2  4  5
     7  8  9
```

If we apply our `factorise_lu` to this matrix we obtain:

```
3×3 LowerTriangular{Float64, Matrix{Float64}}:
 1.0     ⋅    ⋅
 2.0    1.0   ⋅
 7.0  -Inf   1.0
```

```
let
    L, U = factorise_lu(D)
    L
end
```

This `-Inf` is suspicious and points to a problem in the algorithm, which we will investigate in the next section.

LU factorisation with pivoting

We stay with the problem we identified at the end of the previous section. That is factorising the matrix

$$\mathbf{D} = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 7 & 8 & 9 \end{pmatrix}$$

Applying **Algorithm 4** the first step ($k = 1$) will zero out the first column in the second and third row by subtracting the 2 times (7 times) the first row from the second (third) row. When entering the loop for $k = 2$ this results in:

$$\textbf{at beginning of } k = 2\textbf{:} \qquad \mathbf{U} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & -1 \\ 0 & -6 & -12 \end{pmatrix}, \qquad \mathbf{L} = \begin{pmatrix} 1 & & \\ 2 & & \\ 7 & & \end{pmatrix}.$$

The first step of the $k = 2$ iteration in **Algorithm 4** will be to compute $L_{ik} = \frac{U_{ik}}{U_{kk}}$. However, the element U_{kk} is zero as marked **in red**. We thus divide by zero, which is exactly what leads to the introduction of a `-Inf` in the matrix `L`

To see this we use the slider to advance the algorithm step by step, the matrices below show the situtation after the `k = nstep_lu_D` iteration has finished.

- `nstep_lu_D =` 0

```
▸ (U = 3×3 Matrix{Float64}:, L = 3×3 Matrix{Float64}:)
       1.0  2.0  3.0             0.0  0.0  0.0
```

Due their central role in the Gaussian elimination algorithm the denominators U_{kk} in the computation of the L_{ik} are usually referred to as **pivots**.

In summary we observe that from step $k \geq 2$ and onwards our computations are numerical garbage because we have used a 0 pivot.

However, if instead of $\mathbf{D}$ we factorise the **permuted matrix**

$$\mathbf{PD} = \begin{pmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \\ 2 & 4 & 5 \end{pmatrix}$$

which is the matrix $\mathbf{D}$ in which the last two rows are swapped, the algorithm goes through as expected:

```
3×3 Matrix{Int64}:
 1  2  3
 7  8  9
 2  4  5
```

```
1 P * D
```

- `nstep_lu_PD =` 0

```
▸ (U = 3×3 Matrix{Float64}:, L = 3×3 Matrix{Float64}:)
       1.0  2.0  3.0             0.0  0.0  0.0
```

Let us also check that the LU factorisation indeed yields P * U:

```
3×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0
```

```
let
    L, U = factorise_lu(P * D)
    L * U - P * D   # show that L * U = P * D
end
```

We remark that the permutation matrix $\mathbf{P}$ is given by ...

```
P = 3×3 Matrix{Int64}:
 1  0  0
 0  0  1
 0  1  0
```

```
P = [1 0 0;
     0 0 1;
     0 1 0]
```

... and exactly achieves the task of swapping the last two rows, but leaving the rest of $\mathbf{D}$ intact as we saw above.

We notice that even though $\mathbf{D}$ cannot be permuted it is possible to obtain an LU factorisation $\mathbf{LU} = \mathbf{PD}$ if we additionally allow the freedom to cleverly permute the rows of $\mathbf{D}$.

This is in fact a general result:

Theorem 1

Every non-singular matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ admits a factorisation

$$\mathbf{PA} = \mathbf{LU}$$

where $\mathbf{L}$ is lower-triangular, $\mathbf{U}$ upper-triangular and $\mathbf{P}$ is a permutation matrix.

That is to say, that while in general **factorising a matrix $\mathbf{A}$ can fail**, **it always suceeds** for non-singular matrices if we **give ourselves the freedom to permute the rows** of $\mathbf{A}$.

Finding a suitable $\mathbf{P}$ can be achieved by a small modification of **Algorithm 4**. Essentially this modification boils down to **selecting a permutation of rows** of the factorised matrix on the fly, **such that the pivots** of the permuted matrix $\mathbf{PA}$ **are always nonzero**. The precise way how this is achieved is called **pivoting strategy** and the details are beyond the scope of this course. The interested reader can find some discussion in the optional subsection below.

Note, that Julia's implementation of LU factorisation (the `lu` Julia function) does indeed implement one such pivoting strategies, such that it works flawlessly on the problematic matrix `D`:

```
facD = lu(D);
```

Notably beyond the factors $\mathbf{L}$ and $\mathbf{U}$

```
3×3 Matrix{Float64}:
 1.0       0.0  0.0
 0.285714  1.0  0.0
 0.142857  0.5  1.0
```

```
facD.L
```

```
3×3 Matrix{Float64}:
 7.0  8.0      9.0
 0.0  1.71429  2.42857
 0.0  0.0      0.5
```

```
facD.U
```

this factorisation object also contains the employed permutation matrix $\mathbf{P}$:

```
3×3 Matrix{Float64}:
 0.0  0.0  1.0
 0.0  1.0  0.0
 1.0  0.0  0.0
```

```
facD.P
```

Such that as expected $\mathbf{L} * \mathbf{U} = \mathbf{P} * \mathbf{D}$:

```
3×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0
```

```
facD.L * facD.U - facD.P * D
```

However, we notice that Julia's pivoting strategy did end up with a different permutation than our example.

Solving linear systems based on LU factorisation ∈

The result of Theorem 1 is clearly that a factorisation $\mathbf{PA} = \mathbf{LU}$ can always be achieved if the linear system $\mathbf{Ax} = \mathbf{b}$ has a unique solution, that is that $\mathbf{A}$ is non-singular. To employ pivoted LU factorisation to solve this linear system we note that we can always multiply both left and right hand sides by $\mathbf{P}$, therefore:

$$\mathbf{Ax} = \mathbf{b} \quad \Longleftrightarrow \quad \mathbf{PAx} = \mathbf{Pb} \quad \Longleftrightarrow \quad \mathbf{LUx} = \mathbf{Pb}$$

which leads to the following algorithm:

Algorithm 5: Solving linear systems with pivoted LU factorisation

Given a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, a right-hand side $\mathbf{b} \in \mathbb{R}^n$ the linear system $\mathbf{Ax} = \mathbf{b}$ can be solved for $\mathbf{x} \in \mathbb{R}^n$ as follows:

1. Factorise $\mathbf{PA} = \mathbf{LU}$, that is obtain $\mathbf{P}$, $\mathbf{L}$ and $\mathbf{U}$ *(The `lu` Julia function).*
2. Solve $\mathbf{Lz} = \mathbf{Pb}$ for $\mathbf{z}$ using *forward* substitution.
3. Solve $\mathbf{Ux} = \mathbf{z}$ for $\mathbf{x}$ using *backward* substitution.

When we use Julia's backslash `\`-operator, effectively this **Algorithm 5** is executed under the hood.

Let us understand this algorithm by executing the steps manually for solving the problem

$$\mathbf{D}\,\mathbf{x}_D = \mathbf{b} \qquad \text{with} \quad \mathbf{D} = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 7 & 8 & 9 \end{pmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} 2 \\ 4 \\ -2 \end{pmatrix}$$

We already defined defined $\mathbf{D}$ and $\mathbf{b}$ before:

```
3×3 Matrix{Int64}:
 1  2  3
 2  4  5
 7  8  9
```

```
1 D
```

```
▸ [2, 4, -2]
```

```
1 b
```

Step 1: Perform LU factorisation of $\mathbf{D}$:

```
▸ (P = 3×3 Matrix{Float64}:, L = 3×3 Matrix{Float64}:, U = 3×3 Matrix{Float64}:  )
      0.0  0.0  1.0            1.0       0.0  0.0        7.0  8.0       9.0
```

```
1 begin
2     fac = lu(D)
3     (; fac.P, fac.L, fac.U)
4 end
```

Step 2: Compute $\mathbf{z}$ by forward substitution of $\mathbf{Pb}$ wrt. $\mathbf{L}$

```
zD = [-2.0, 4.57143, 0.0]
```

```
zD = forward_substitution(fac.L, fac.P * b)
```

Step 3: Backward-substitute $\mathbf{z}$ wrt. $\mathbf{U}$:

```
xD = [-3.33333, 2.66667, 0.0]
```

```
xD = backward_substitution(fac.U, zD)
```

Verify result: This solves the problem $\mathbf{D}\mathbf{x}_D = \mathbf{b}$ as desired:

```
[0.0, 0.0, 0.0]
```

```
D * xD - b
```

Optional: More details on pivoting

In this section we provide some details of one common form of pivoting from LU factorisation, namely **row pivoting**.

In this approach we allow ourselves some flexibility in **Algorithm 4** by allowing ourselves to change the order in the last few rows and thus *choose* the pivot amongst the entries U_{ik} with $i \geq k$ in each step k. The resulting change of row order will then define the permutation matrix $\mathbf{P}$ we employed in our above discussion.

As a guiding example we again consider the problem of factorising

$$\mathbf{D} = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 7 & 8 & 9 \end{pmatrix}$$

where we saw previously non-pivoted LU factorisation to fail. Without any pivoting / row swapping after the first LU factorisation step (i.e when $k = 2$ will start) the situation is

$$\textbf{at beginning of } k = 2\text{:} \qquad \mathbf{U} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & -1 \\ 0 & -6 & -12 \end{pmatrix}, \qquad \mathbf{L} = \begin{pmatrix} 1 & & \\ 2 & & \\ 7 & & \end{pmatrix}.$$

Looking at this $\mathbf{U}$ it seems very reasonable to just swap the second and the third column in and thus move the -6 to become the new pivot. For consistency we not only need to swap $\mathbf{U}$, but also $\mathbf{L}$. This gives us

$$\textbf{after row swap:} \qquad \mathbf{U} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & -6 & -12 \\ 0 & 0 & -1 \end{pmatrix}, \qquad \mathbf{L} = \begin{pmatrix} 1 & & \\ 7 & & \\ 2 & & \end{pmatrix}.$$

If we continue the algorithm now, the -6 sits in the position of the pivot $\left(A^{(k)}\right)_{kk}$ and the division by zero is avoided.

In fact in this particular case the matrix $\mathbf{U}$ is already in upper triangular form after step $k = 2$, such that in the step $k = 3$ nothing will change, in fact we will just get $L_{32} = 0$. After $k = 3$ we thus obtain from our algorithm:

$$\mathbf{U} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & -6 & -12 \\ 0 & 0 & -1 \end{pmatrix} \qquad \mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ 7 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix}.$$

Due to the additional row permutation we performed multiplying out $\mathbf{LU}$ will not yield $\mathbf{A}$, but a matrix where the second and third rows of $\mathbf{A}$ are swapped:

$$\mathbf{LU} = \begin{pmatrix} 1 & 0 & 0 \\ 7 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 0 & -6 & -12 \\ 0 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \\ 2 & 4 & 5 \end{pmatrix}$$

This is not surprising as with pivoting we expect to get $\mathbf{LU} = \mathbf{PA}$, so our missing piece is to find the permutation matrix $\mathbf{P}$.

Here the correct matrix is

$$\mathbf{P} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix},$$

as we saw before. We can now understand how this matrix has been constructed: Namely if we take the identity matrix and perform exactly the same row permutations as during the pivoted LU factorisation. That is we swap the second and third row. With this matrix we can easily check that

$$\mathbf{LU} = \mathbf{PA}.$$

If we thus extend our `factorise_lu` function to additionally perform such pivoting permutations, the Gaussian elimination algorithm would always terminate successfully.

But pivoting brings additional opportunities. As it turns out numerical stability of LU factorisation can improve if one permuts the rows of $\mathbf{U}$ not only if a pivot U_{kk} is zero, but in fact during each iteration k, ensuring that the **pivot** is **as large as possible**.

In other words in the k-th step of LU factorisation we always exchange row k with the row l where l satisfies

$$|U_{lk}| \leq |U_{ik}| \qquad \text{for all } i = k, \dots n.$$

The appropriate swaps are tracked and returned as well.

In practical algorithms instead of returning a permutation matrix $\mathbf{P}$ (which requries to store n^2 elements) it is usually more convenient to store a permutation vector $\mathbf{p}$, which has only n elements. This vector tracks the indices of the rows of $\mathbf{A}$ in the order they are used as pivots. In other words if

```
idmx = 4×4 Matrix{Float64}:
       1.0  0.0  0.0  0.0
       0.0  1.0  0.0  0.0
       0.0  0.0  1.0  0.0
       0.0  0.0  0.0  1.0
```

```
1 idmx = diagm(ones(4))
```

is the identity matrix and

```
perm = [1, 3, 2, 4]
```

```
1 perm = [1, 3, 2, 4]
```

the permutation vector, then

```
4×4 Matrix{Float64}:
 1.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0
 0.0  1.0  0.0  0.0
 0.0  0.0  0.0  1.0
```

```
1 idmx[perm, :]
```

is the permutation matrix.

The code below presents an implementation of row-pivoted LU factorisation.

```
factorise_lu_pivot (generic function with 1 method)
```

```
function factorise_lu_pivot(A)
    n = size(A, 1)
    L = zeros(n, n)
    U = zeros(n, n)
    p = fill(0, n)
    Ak = float(copy(A))  # Make a copy of A and ensure that all entries
                         # are converted to floating-point numbers

    for k in 1:n-1             # Algorithm steps
        p[k] = argmax(abs.(Ak[:, k]))  # Find row with maximal pivot

        U[k, :] = Ak[p[k], :]  # Copy pivot row to U, use U now instead of Ak,
                               # which is again updated in-place
        for i in 1:n           # Row loop: Note the full range as any row may
                               #           be non-zero
            L[i, k] = Ak[i, k] / U[k, k]
            for j = 1:n        # Column loop: Again full range
                Ak[i, j] = Ak[i, j] - L[i, k] * U[k, j]
            end
        end
    end
    p[n] = argmax(abs.(Ak[:, n]))
    U[n, n] = Ak[p[n], n]
    L[:, n] = Ak[:, n] / U[n, n]

    # To simplify assembling L we so far kept the rows in the same order
    # as in A. To make the matrix upper triangular we also apply the column
    # permutation p before returning the results.
    (; L=LowerTriangular(L[p, :]), U=UpperTriangular(U), p=p)
end
```

We check our implementation of pivoted LU factorisation against Julia's `RowMaximum` pivoting strategy, which implements the same algorithm.

```
reference = LinearAlgebra.LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
3×3 Matrix{Float64}:
 1.0       0.0  0.0
 0.285714  1.0  0.0
 0.142857  0.5  1.0
U factor:
3×3 Matrix{Float64}:
 7.0  8.0      9.0
 0.0  1.71429  2.42857
 0.0  0.0      0.5
```

```
reference = lu(D, RowMaximum())
```

The resulting L, U and pivot vector values are:

```
3×3 Matrix{Float64}:
 1.0       0.0  0.0
 0.285714  1.0  0.0
 0.142857  0.5  1.0
```

```
fac.L
```

```
3×3 Matrix{Float64}:
 7.0  8.0      9.0
 0.0  1.71429  2.42857
 0.0  0.0      0.5
```

```
fac.U
```

```
▸ [3, 2, 1]
```

```
fac.p
```

In contrast we obtain:

```
3×3 LowerTriangular{Float64, Matrix{Float64}}:
 1.0       ⋅    ⋅
 0.285714  1.0   ⋅
 0.142857  0.5  1.0
```

```
factorise_lu_pivot(D).L
```

```
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 7.0  8.0      9.0
  ⋅   1.71429  2.42857
  ⋅    ⋅       0.5
```

```
factorise_lu_pivot(D).U
```

```
▸ [3, 2, 1]
```

```
factorise_lu_pivot(D).p
```

Memory usage and fill-in

Sparse matrices

The matrices and linear systems one encounters in physics and engineering applications not infrequently reach **huge sizes**. For example in the numerical solution of partial differential equations using finite difference or finite element methods the vector of unknowns frequently has a **dimension on the order of** $n \simeq 10^6$.

Taking $n = 10^6$ thus as an example this implies that matrices, i.e. entities from $\mathbb{R}^{n \times n}$, hold around 10^{12} **elements**. In double precision, i.e. 64-bit numbers, each entry requires 8 bytes. As a result **storing all elements** of such a matrix explicitly in memory reqires

```
8000000000000
```

```
1 10^12 * 8
```

bytes or

```
7.275957614183426
```

```
1 10^12 * 8 / 1024^4  # TiB
```

tibibyes of storage. **This is a challenge** even for modern compute clusters, where typically a node has around 512 GiB. Laptops nowadays feature around 32 GiB, so would be completely out of the question to solve such problems.

However, **in many applications** the arising **matrices are sparse**, meaning that they contain a large number of zero elements, which do not need to be stored. Let us discuss a few examples.

Definition: Full matrix

A matrix $A \in \mathbb{R}^{n \times n}$ is called **full** if the number of non-zero elements is at the order of n^2. For such matrices almost all elements are non-zero and need to be stored.

Full matrices are the "standard case" and for them memory constraints usually set the limit of the size of problems, which can be tackled. For example an 32 GiB memory laptop can store around $4 \cdot 10^9$ double-precision numbers, which means that linear problems with more than around $n \simeq 60000$ unknows cannot be treated.

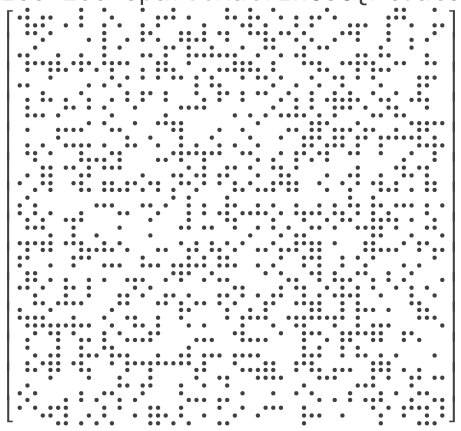
Definition: Sparse matrix

A matrix $A \in \mathbb{R}^{n \times n}$ is called **sparse** if the number of non-zero elements is at the order of n.

If we know which elements are zero, we can thus save memory by only storing those elements, which are non-zero. For this purpose the `SparseArrays` Julia package implements many primitives for working with sparse arrays.

This includes, generating random sparse arrays:

```
Asp = 100×100 SparseMatrixCSC{Float64, Int64} with 1458 stored entries:
```



```
1 Asp = sprand(100, 100, 0.15)
```

or simply sparsifying a dense array using the `sparse` function, which converts a full matrix to a sparse matrix by dropping the explicit storage of all zero entries.

```
M = 4×4 Matrix{Int64}:
  1  0  0   5
 -2  0  1   0
  0  0  6   0
  0  1  0  -1
```

```
1 M = [ 1 0 0  5;
2      -2 0 1  0;
3       0 0 6  0;
4       0 1 0 -1]
```

```
4×4 SparseMatrixCSC{Int64, Int64} with 7 stored entries:
  1   ⋅   ⋅   5
 -2   ⋅   1   ⋅
  ⋅   ⋅   6   ⋅
  ⋅   1   ⋅  -1
```

```
1 sparse(M)
```

Using the `SparseArray` data structure from `SparseArrays` consistently allows to fully exploit sparsity when solving a problem. As a result the **storage costs scale only as $O(n)$**. With our laptop of 32 GiB memory we can thus tackle problems with around $n \simeq 4 \cdot 10^9$ unknowns — much better than the 60000 we found when using full matrices.

Finally we introduce a special kind of sparse matrix:

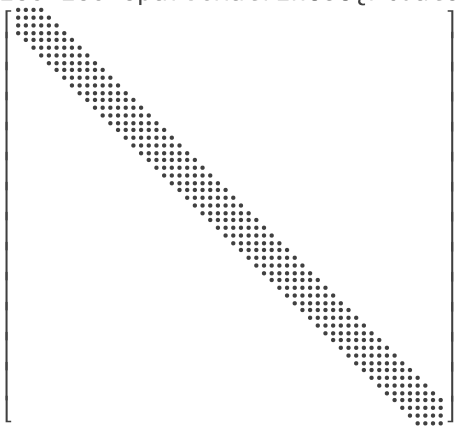
Definition: Band matrix

A matrix $A \in \mathbb{R}^{n \times n}$ is called a **band matrix** with bandwidth d if $A_{ij} = 0$ when $|j - i| > d$. Every line of the matrix contains at most $2d + 1$ non-zero elements and the number of non-

zeros thus scales as $O(nd)$.

An example for a banded matrix with bandwidth 5 is:

```
band = 105×105 SparseMatrixCSC{Float64, Int64} with 817 stored entries:
```



Memory: LU factorisation of full matrices

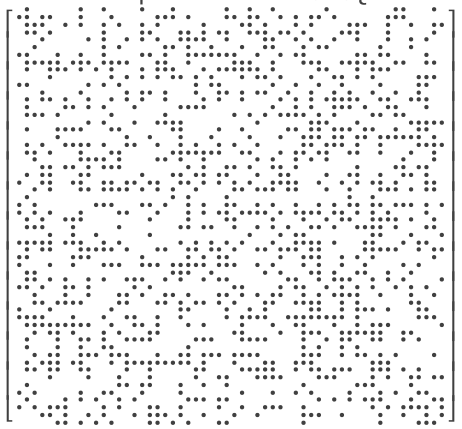
Since the amount of available memory can put hard constraints on the size of linear systems which can be solved, we now want to investigate the memory requirement of LU factorisation $\mathbf{A} = \mathbf{LU}$.

If $\mathbf{A}$ is a **full matrix**, then we know that $\mathbf{L}$ and $\mathbf{U}$ are triangular, thus **L and U** contain less non-zero elements or **more zero elements** than $\mathbf{A}$ itself and thus **require together as much memory** to be stored in memory as $\mathbf{A}$ itself.

Memory: LU factorisation of sparse matrices

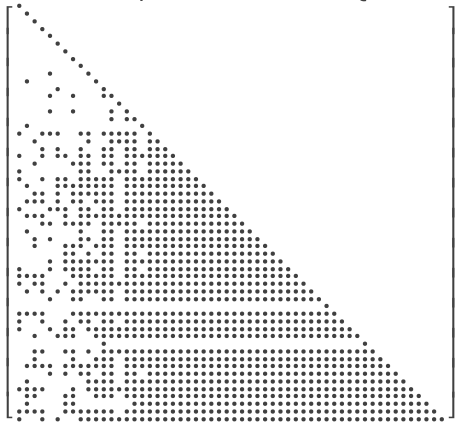
Let's **contrast** this **with the sparse matrices** we have considered above. First the random sparse matrix:

```
100×100 SparseMatrixCSC{Float64, Int64} with 1458 stored entries:
```



```
1 Asp
```

```
100×100 SparseMatrixCSC{Float64, Int64} with 2960 stored entries:
```



```
lu(Asp).L  # U looks similar
```

Storing both L and U thus requires a total number of

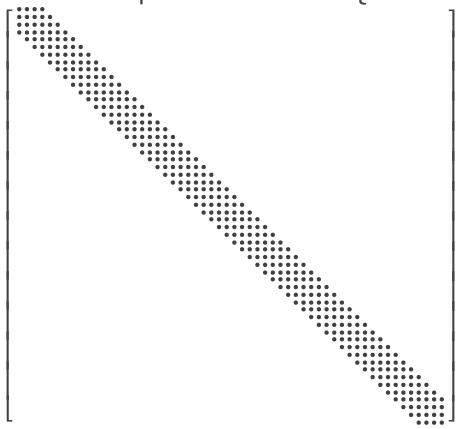
```
5738
```

```
nnz(lu(Asp).L) + nnz(lu(Asp).U)
```

non-zero elements, which is about **3.9 times as much** as the original matrix !

Now the banded matrix:

```
105×105 SparseMatrixCSC{Float64, Int64} with 817 stored entries:
```



```
band
```

```
105×105 SparseMatrixCSC{Float64, Int64} with 611 stored entries:
```

```
lu(band).L  # U looks similar
```

At least the banded structure is kept, but still we require

```
1222
```

```
nnz(lu(band).L) + nnz(lu(band).U)
```

nonzeros, which is about 1.5 times than the original in this case.

In both cases storing the $\mathbf{L}$ and the $\mathbf{U}$ factors require more non-zero elements than the original matrix. This phenomenon is usually referred to as **fill in**.

In particular for the sparse matrix `Asp` LU factorisation did not preserve the structure at all. In contrast it almost resulted in a full matrix in the lower right corner of the $\mathbf{L}$. For $\mathbf{U}$ this looks exactly the same. In fact one can show that in general even for sparse matrices the **memory usage of LU factorisation** is $O(n^2)$. Therefore while one may be able to store a sparse matrix $\mathbf{A}$ with a huge size in memory (due to the $O(n)$ memory cost), one may actually run out of memory while computing the factorisation.

Let us add that for banded matrices the situation is slightly better as one can show that in this case the fill-in takes place at most inside the band. As a result the memory requirement stays at $O(nd)$.

Overview of LU factorisation memory cost

We summarise the cost of LU factorisation in a table:

type of $n \times n$ matrix	memory usage	comment
`full matrix`	$O(n^2)$	
`general sparse matrix`	$O(n)$	`fill-in`

banded matrix, band width d	$O(nd)$	stays block-diagonal

Computational cost of LU factorisation

In this section we want to investigate how long it will take a computer to perform LU factorisation on a large matrix.

Modern laptop computers nowadays have a clock frequency of a few Gigahertz (GHz). This means they are **able to perform about 10^9 operations per second**, where for simplicity we assume that "one operation" is an elementary addition, multiplication, division and so on. If we can determine how many such operations are needed to perform LU factorisation we can estimate the computational cost.

Before we look at the LU algorithm (Algorithm 4) we first understand a few simpler cases from linear algebra:

Scalar product

Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ consider computing the scalar product

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T \mathbf{y} = \sum_{i=1}^{n} x_i \, y_i$$

which in code is achieved as

```
scalar_product (generic function with 1 method)
```

```
function scalar_product(x, y)
    result = 0.0
    for i in 1:length(x)
        result = result + x[i] * y[i]
    end
    result
end
```

In the loop for each iteration we require 1 multiplication and 1 addition. In total the `scalar_product` function therefore requires n multiplications and n additions. The number of elementary operations is thus $2n$. We say the **computational cost is $O(n)$** (i.e. on the order of n or linear in n).

If we take the dimensionality $n = 1000$ the number of operations is $O(1000)$. On a 1 GHz computer (with 10^9 operations per second) this therefore takes about 10^{-6} seconds ... hardly

noticable.

Matrix-vector product

Given a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ and a vector $\mathbf{x} \in \mathbb{R}^n$ the matrix-vector product $\mathbf{y} = \mathbf{A}\mathbf{x}$ is computed as

$$y_i = \sum_{j=1}^{n} A_{ij} x_j \qquad \text{for } i = 1, \ldots, n$$

or in code

```
matrix_vector_product (generic function with 1 method)
```

```
function matrix_vector_product(A, x)
    y = zeros(size(A, 1))  # Create a vector of zeros (#)

    for i in 1:size(A, 1)
        for j in 1:size(A, 2)
            y[i] = y[i] + A[i, j] * x[j]  # (*)
        end
    end
    result
end
```

In the innermost loop we observe again that we require 1 addition and 1 multiplication per iteration. This instruction is performed once for each combination of i and j, so we look at the limits of each of the nested loops. The loop over i runs over n values (the number of rows of $\mathbf{A}$) and the loop over j over n values as well (the number of columns of $\mathbf{A}$). In total the inner most instruction $(*)$ is thus run n^2 times, each times costing 2 operations. The total **cost is thus $O(n^2)$**.

For our case with $n = 1000$ and a 1 GHZ computer we thus now need $(10^3)^2/10^9 s = 10^{-3} s = 1\text{ms}$, which is again a rather short time.

▶ **Expert information: What about the allocation in `(#)` ?**

Overview of computational cost

For vectors $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ and matrices $\mathbf{A} \in \mathbb{R}^{n \times n}$ the computational cost is

operation	cost
dot product $\mathbf{v}^T\mathbf{w}$	$O(n)$
matriv-vector product $\mathbf{A}\mathbf{v}$	$O(n^2)$

matrix-matrix multiplication $\mathbf{AB}$	$O(n^3)$

Finally a few rough guidelines:

General guideline to estimate computational cost

1. Determine the instructions at the innermost (most deeply nested) loop level
2. Find the most expensive of these instructions and determine its cost in terms of *number of operations*
3. Determine the ranges of all loops in terms of the dimensionality of your problem. Typically for each loop level this is n
4. Multiply the result of 2. and all index ranges of 3 to get the total scaling. Typically for a single loop nesting the cost is $O(n)$ for a doubly nested loop $O(n^2)$ and so on.

Example Matrix-matrix multiplication

Let us code up an algorithm how to compute the product of two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$ and analyse its complexity.

```
matmul (generic function with 1 method)
```

```
function matmul(A, B)
    C = zeros(size(A, 1), size(B, 2))

    # loops ...

    C
end
```

LU factorisation

We revisit Algorithm 4:

Algorithm 4: LU factorisation

Input: $\mathbf{A} \in \mathbb{R}^{n \times n}$

Output: $\mathbf{U} \in \mathbb{R}^{n \times n}$, $\mathbf{L} \in \mathbb{R}^{n \times n}$

- $\mathbf{U} = \mathbf{A}$
- for $k = 1, \ldots, n-1$ *(algorithm steps)*

    - $L_{kk} = 1$
    - for $i = k+1, \dots, n$ *(Loop over rows)*
        - $L_{ik} = \frac{U_{ik}}{U_{kk}}$
        - for $j = k, \dots n$ *(Loop over columns)*
            - $U_{ij} \leftarrow U_{ij} - L_{ik}U_{kj}$ *(modify U_{ij} to $U_{ij} - L_{ik}U_{kj}$)*
- $L_{nn} = 1$ *(the loop above only runs until $n-1$)*

We notice that the instruction at the innermost loop level is

$$A_{ij}^{(k+1)} = A_{ij}^{(k)} - L_{ik}A_{kj}^{(k)},$$

which costs again 1 addition and 1 multiplication and which is executed once for each tuple (i,j,k). In the worst case k is takes $n-1$ elements, i also $n-1$ elements (assume $k=1$) and j again in its worst case takes $n-1$ elements. In total the cost is thus $(n-1)^3 = O(n^3)$.

For our case with $n = 1000$ and a 1 GHz computer the computation thus needs approximately $(10^3)^3/10^9 s = 1s$, which starts to be a noticable amount of time. If we even consider $n = 10^5$ all of a sudden it takes $10^3 s$, which is about 15 minutes. For **large matrices the cost of computing LU factorisation can thus become important** !

For **banded matrices** the computational scaling is a little improved. E.g. consider LU factorisation on a banded matrix A with band width d. Then bothe the loop over i (row loop) as well as the loop over j (column loop) in the LU factorisation algorithm can be truncated and at most run over d elements. As a result the computational cost at worst becomes $O(nd^2)$ which for a small band width (i.e. $d < n$) is substantially less than $O(n^3)$.

Overview of LU factorisation computational cost and memory cost

We summarise the cost of LU factorisation in a table:

type of $n \times n$ matrix	computational cost	memory usage
`full matrix`	$O(n^3)$	$O(n^2)$
`general sparse matrix`		
`banded matrix, band width` d	$O(n\,d^2)$	$O(n\,d)$

Numerical stability

To close off our discussion of Gaussian elimination we consider the question of its numerical stability, i.e. how is the **result of LU factorisation effected by small round-off errors** introduced when performing a calculation on a computer with its finite precision to represent numbers.

In this chapter our goal has been to solve a linear system of the form

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{5}$$

where we are given a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ as well as a right-hand side $\mathbf{b} \in \mathbb{R}^n$ and we wish to solve for $\mathbf{x}$ We will later refer to this system as the **exact system**.

Since the computer in general is unable to represent the matrix and right-hand side exactly in finite-precision floating point arithmetic, even **just providing this problem to a computer** will usually be associated with **making a small error**: in practice the computer only ever sees the **perturbed system**

$$\widetilde{\mathbf{A}}\widetilde{\mathbf{x}} = \widetilde{\mathbf{b}}, \tag{6}$$

where $\widetilde{\mathbf{A}}$ is an approximation to $\mathbf{A}$, $\widetilde{\mathbf{b}}$ is an approximation to $\mathbf{b}$. It will thus yield the solution $\widetilde{\mathbf{x}}$ by solving the perturbed system instead of providing $\mathbf{x}$.

To understand the numerical stability when solving linear systems our goal is thus to understand how far the solution $\widetilde{\mathbf{x}}$ — obtained on a computer — differs from $\mathbf{x}$ — the solution to (5), the true linear system we want to solve.

To provide a motivation **why this type of analysis matters in practice**, we first consider the following example:

Example: Rounding error in a 2x2 system

Consider the *exact* linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad \Leftrightarrow \quad \begin{pmatrix} 1 & 10^{-16} \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

with solution $(x_1, x_2) = (1, 0)$ and the *perturbed* linear system

$$\mathbf{A}\widetilde{\mathbf{x}} = \widetilde{\mathbf{b}} \quad \Leftrightarrow \quad \begin{pmatrix} 1 & 10^{-16} \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \widetilde{x}_1 \\ \widetilde{x}_2 \end{pmatrix} = \begin{pmatrix} 1 + 10^{-16} \\ 1 \end{pmatrix},$$

where only the first component of the right-hand side has been perturbed by 10^{-16}. This second system has solution $(\widetilde{x}_1, \widetilde{x}_2) = (1, 1)$. The second component of this solution is thus *completely wrong*!

The small perturbation of 10^{-16}, which may readily occurr just by inputing the data to a computer, can already have a very noticable effect on the solution and in fact render the solution we obtain from our our computation (i.e. $\widetilde{\mathbf{x}}$) rather far from the answer we are actually after (i.e. $\mathbf{x}$).

To make this more quantitative, we remind ourselves:

Definition: Absolute and relative error

Given two vectors $\mathbf{x} \in \mathbb{R}^n$ and $\tilde{\mathbf{x}} \in \mathbb{R}^n$, where $\tilde{\mathbf{x}}$ is thought to be an approximation of $\mathbf{x}$. Then the absolute error of $\tilde{\mathbf{x}}$ is $\|\mathbf{x} - \tilde{\mathbf{x}}\|$, while the relative error is $\frac{\|\mathbf{x}-\tilde{\mathbf{x}}\|}{\|\mathbf{x}\|}$.

Rounding error in a 2x2 system (continued)

We saw above that a small perturbation $1 \to 1 + 10^{-16}$ in one of the elements of the right hand side, introduces a change in the solution from $(x_1, x_2) = (1, 0)$ to $(\widetilde{x}_1, \widetilde{x}_2) = (1, 1)$. In the solution this is an absolute error of

$$\|\mathbf{x} - \widetilde{\mathbf{x}}\| = \left\| \begin{pmatrix} 0 \\ -1 \end{pmatrix} \right\| = 1$$

and thus a relative error of

$$\frac{\|\mathbf{x} - \widetilde{\mathbf{x}}\|}{\|\mathbf{x}\|} = \frac{1}{1} = 1.$$

while in the right-hand side this only was an absolute error of

$$\|\mathbf{b} - \widetilde{\mathbf{b}}\| = 10^{-16}$$

and a relative error of

$$\frac{\|\mathbf{b} - \widetilde{\mathbf{b}}\|}{\|\mathbf{b}\|} = \frac{10^{-16}}{\sqrt{2}}$$

As outlined in the introduction, in standard floating-point arithmetic the **relative error** in representing any number is around 10^{-16}. Mathematically we can find the following relationships between the elements of $\widetilde{\mathbf{b}}$ and $\mathbf{b}$ as well as $\widetilde{\mathbf{A}}$ and $\mathbf{A}$, respectively:

$$\begin{aligned} \tilde{b}_i &= b_i \left(1 + \varepsilon_i\right) \qquad && \textbf{where } \varepsilon_i \textbf{ is on the order of } 10^{-16} \\ \widetilde{A}_{ij} &= A_{ij} \left(1 + \eta_{ij}\right) \qquad && \textbf{where } \eta_{ij} \textbf{ is on the order of } 10^{-16} \end{aligned}$$

However, to simplify the treatment in this course we will not attempt to discuss the effect of such relative errors on the numerical stability in full generality. Much rather we will employ the **error model**:

Error model in this notebook

Between the exact system and the perturbed system we will assume the foollowing relationship

$$\begin{aligned} \tilde{b}_i &= b_i \left(1 + \varepsilon\right) \qquad && \textbf{where } \varepsilon \approx 10^{-16} \\ \widetilde{A}_{ij} &= A_{ij} \end{aligned}$$

In other words we assume that the system matrix $\mathbf{A}$ is exactly represented on the computer and that round-off errors only affect the right-hand side $\widetilde{\mathbf{b}}$. Moreover we assume that the relative error for all elements of $\mathbf{b}$ is identical to ε, i.e. the overall relative error in the right-hand side is

$$\frac{\|\mathbf{b} - \widetilde{\mathbf{b}}\|}{\|\mathbf{b}\|} = \frac{|1 - (1 + \varepsilon)| \, \|\mathbf{b}\|}{\|\mathbf{b}\|} = \varepsilon$$

as well.

For standard double-precision floating-point numbers we have $\epsilon \approx 10^{-16}$.

For an alternative and more detailed discussion see also chapter 2.8 of Driscoll, Brown: *Fundamentals of Numerical Computation*.

Matrix condition numbers and stability result

To analyse the error $\mathbf{x} - \widetilde{\mathbf{x}}$ mathematically, we first have to introduce some notation.

Recall that for a vector $\mathbf{v} \in \mathbb{R}^n$ its Euclidean norm is $\|\mathbf{v}\| = \sqrt{\sum_{i=1}^n v_i^2}$.

We define the following:

Definition: Relative error of linear systems

The **relative error** in $\widetilde{\mathbf{x}}$, the solution to the perturbed system (6), is the quantity

$$\frac{\|\mathbf{x}-\widetilde{\mathbf{x}}\|}{\|\mathbf{x}\|},$$

where $\mathbf{x}$ is the exact solution, i.e. the solution to the exact linear system (5) one actually wanted to solve.

We further need a generalisation of norms to matrices:

Definition: Matrix norm

Given $\mathbf{M} \in \mathbb{R}^{m\times n}$ a real matrix (not necessarily square), we define the matrix norm of $\mathbf{M}$ as

$$\|\mathbf{M}\| = \max_{\substack{\mathbf{v}\in\mathbb{R}^n \\ \mathbf{v}\neq\mathbf{0}}} \frac{\|\mathbf{M}\mathbf{v}\|}{\|\mathbf{v}\|}.$$

The previous definition implies in particular that

$$\|\mathbf{M}\mathbf{v}\| \leq \|\mathbf{M}\|\,\|\mathbf{v}\| \quad \forall \mathbf{x}\in\mathbb{R}^n. \tag{7}$$

With this definitions in place we begin our analysis for (5) and (6) where $\mathbf{A}\in\mathbb{R}^{n\times n}$. From the definition of the exact and perturbed linear systems

$$\mathbf{A}\mathbf{x}=\mathbf{b} \qquad \text{and} \qquad \mathbf{A}\widetilde{\mathbf{x}}=\widetilde{\mathbf{b}}$$

we obtain by subtraction and since $\mathbf{A}$ is invertible

$$\mathbf{A}\left(\mathbf{x}-\widetilde{\mathbf{x}}\right)=\mathbf{b}-\widetilde{\mathbf{b}} \qquad \Rightarrow \qquad \mathbf{x}-\widetilde{\mathbf{x}}=\mathbf{A}^{-1}\left(\mathbf{b}-\widetilde{\mathbf{b}}\right)$$

Using (7) we therefore have

$$\|\mathbf{x}-\widetilde{\mathbf{x}}\| \leq \left\|\mathbf{A}^{-1}\right\| \left\|\mathbf{b}-\widetilde{\mathbf{b}}\right\|.$$

Furthermore we have from applying (7) to $\mathbf{A}\mathbf{x}=\mathbf{b}$:

$$\|\mathbf{b}\| \leq \|\mathbf{A}\|\,\|\mathbf{x}\| \qquad \Rightarrow \qquad \frac{1}{\|\mathbf{x}\|} \leq \frac{\|\mathbf{A}\|}{\|\mathbf{b}\|}$$

such that with the above result we can bound the relative error as

$$\frac{\|\mathbf{x}-\widetilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \left\|\mathbf{A}^{-1}\right\| \left\|\mathbf{b}-\widetilde{\mathbf{b}}\right\| \cdot \frac{\|\mathbf{A}\|}{\|\mathbf{b}\|} = \underbrace{\left\|\mathbf{A}^{-1}\right\| \|\mathbf{A}\|}_{=\kappa(\mathbf{A})} \frac{\left\|\mathbf{b}-\widetilde{\mathbf{b}}\right\|}{\|\mathbf{b}\|}. \tag{8}$$

From as stability analysis point of view the quantity $\kappa(\mathbf{A}) = \left\|\mathbf{A}^{-1}\right\| \|\mathbf{A}\|$ thus relates the relative error in the right-hand side (input quantity) to the relative error in the solution $\mathbf{x}$ (output quantity). We call this the condition number for a linear system:

Definition: Condition number of a linear system

Given a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ with an invertible system matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ its **condition number** is defined as

$$\kappa(\mathbf{A}) = \left\|\mathbf{A}^{-1}\right\| \|\mathbf{A}\|.$$

Due to the importance of solving linear systems in linear algebra one usually calls $\kappa(\mathbf{A})$ also the **condition number of the matrix $\mathbf{A}$**.

But notably (8) can be interpreted even more generally as the way it the solution of linear systems are changed as we change the right-hand side as is summarised below:

Theorem 2: Change of solution when changing the right-hand side

Given a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$, which we solve for $\mathbf{x}$ and a related linear system $\mathbf{A}\widetilde{\mathbf{x}} = \widetilde{\mathbf{b}}$ which we solve for $\widetilde{\mathbf{x}}$, then the two solutions are related as

$$\frac{\|\mathbf{x}-\widetilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \kappa(\mathbf{A}) \frac{\left\|\mathbf{b}-\widetilde{\mathbf{b}}\right\|}{\|\mathbf{b}\|}.$$

In this notebook we employed an error model where storing $\mathbf{b}$ on the computer (which leads to $\widetilde{\mathbf{b}}$) introduces a small relative error:

$$\tilde{b}_i = b_i\,(1+\varepsilon).$$

This enables to simplify the expression of the absolute error of right-hand side:

$$\left\|\mathbf{b}-\widetilde{\mathbf{b}}\right\| = \sqrt{\sum_{i=1}^{n} b_i^2 \varepsilon^2} = \varepsilon \sqrt{\sum_{i=1}^{n} b_i^2} = \epsilon \left\|\mathbf{b}\right\|,$$

which leads to the following result:

Theorem 3: Stability of solving linear systems

Given a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ and $\widetilde{\mathbf{x}}$ the solution to a perturbed linear system $\mathbf{A}\widetilde{\mathbf{x}} = \widetilde{\mathbf{b}}$, where $\tilde{b}_i = b_i\left(1+\varepsilon\right)$ for $i = 1, \ldots, n$, then

$$\frac{\left\|\mathbf{x}-\widetilde{\mathbf{x}}\right\|}{\left\|\mathbf{x}\right\|} \leq \kappa(\mathbf{A})\,\varepsilon$$

This inequality shows that the condition number of the matrix $\mathbf{A}$ plays the role of *a worst-case amplification* of the round-off error introduced by the floating-point representation of the right-hand side.

Since for standard double-precision floating-point numbers we have $\epsilon \approx 10^{-16}$, this means that condition numbers of 10^{16} lead to a relative error of 1, i.e. 100% error — all precision is lost. In general condition numbers above 10^8 start to become problematic as in this case the relative error in the solution is at most 10^{-8}, i.e. roughly speaking no more than 8 digits.

Numerically computing the condition number. In Julia code the condition number of a matrix is computed using the `cond` function. It effectively uses the above formulas for its computation.

For example for our running example we obtain:

```
2.0000000000000004e16
```

```
let
    A = [1 1e-16;
         1  0   ]
    κ = cond(A)
end
```

Notice, that this is $2 \cdot 10^{16}$, which is a huge number !

Rounding error in a 2x2 system (continued)

Using Theorem 3 we can finally understand the behaviour we observe in our example. Previously we found that a small perturbation $1 \to 1 + 10^{-16}$ in one of the elements of the

right hand side, introduces a change in the solution from $(x_1, x_2) = (1, 0)$ to $(\tilde{x}_1, \tilde{x}_2) = (1, 1)$. We also found this to be a relative error of

$$\frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} = \frac{1}{1} = 1.$$

Note that in this example $\epsilon \approx 10^{-16}$ just like for the case of a rounding error in the right-hand side.

As we computed above

$$\kappa(\mathbf{A}) \approx 2 \cdot 10^{16},$$

such that this large relative error is explained by the large condition number of this matrix.

Computing condition numbers

Using the `cond` function of Julia enables us to easily compute condition numbers in practice.

However, the expression $\kappa(\mathbf{A}) = \left\|\mathbf{A}^{-1}\right\| \, \|\mathbf{A}\|$ is not extremly handy to provide a good intuition for what the condition number actually means or how it varies as the matrix $\mathbf{A}$ is changed.

In this section we thus discuss a few standard techniques how to compute condition numbers of matrices. These techniques are in fact exactly the algorithms that `cond` uses under the hood to do its computation.

First we need some notation:

Definition: Absolutely minimal and maximal eigenvalues

Given a diagonalisable matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$ we denote by $\lambda^{\text{abs}}_{\max}(\mathbf{M})$ the *largest absolute* eigenvalue of $\mathbf{M}$ and by $\lambda^{\text{abs}}_{\min}(\mathbf{M})$ the *smallest absolute* eigenvalue of $\mathbf{M}$, i.e.

$$\lambda^{\text{abs}}_{\max}(\mathbf{M}) = \max_{i=1,\ldots,n} |\lambda_i(\mathbf{M})| \qquad \text{and} \qquad \lambda^{\text{abs}}_{\min}(\mathbf{M}) = \min_{i=1,\ldots,n} |\lambda_i(\mathbf{M})|$$

where $\lambda_i(\mathbf{M})$ for $i = 1, \ldots, n$ are the eigenvalues of $\mathbf{M}$.

Examples

To develop some intuition about $\lambda^{\text{abs}}_{\max}$ we consider the diagonal matrices:

$$\mathbf{A} = \begin{pmatrix} 5 & 0 \\ 0 & 10 \end{pmatrix} \qquad \lambda^{\text{abs}}_{\max}(\mathbf{A}) = 10 \qquad \lambda^{\text{abs}}_{\min}(\mathbf{A}) = 5$$

$$\mathbf{B} = \begin{pmatrix} -5 & 0 \\ 0 & 10 \end{pmatrix} \qquad \lambda^{\text{abs}}_{\max}(\mathbf{B}) = 10 \qquad \lambda^{\text{abs}}_{\min}(\mathbf{B}) = 5$$

$$\mathbf{C} = \begin{pmatrix} -500 & 0 \\ 0 & 10 \end{pmatrix} \qquad \lambda^{\text{abs}}_{\max}(\mathbf{C}) = 500 \qquad \lambda^{\text{abs}}_{\min}(\mathbf{C}) = 10$$

It turns out that the minimal and maximal absolute eigenvalues provide a convenient way to compute matrix norms:

Lemma 4: Computing matrix norms

For any matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$

$$\|\mathbf{M}\| = \sqrt{\lambda^{\text{abs}}_{\max}(\mathbf{M}^T\mathbf{M})}.$$

If $\mathbf{M}$ is moreover square and invertible, then

$$\|\mathbf{M}^{-1}\| = \sqrt{\lambda^{\text{abs}}_{\max}(\mathbf{M}^{-T}\mathbf{M}^{-1})} = \frac{1}{\sqrt{\lambda^{\text{abs}}_{\min}(\mathbf{M}^T\mathbf{M})}}.$$

Based on this we can deduce a few useful formulas for computing condition numbers. We start with the general expression:

Corollary 5: Condition number for invertible matrices

If $\mathbf{A} \in \mathbb{R}^{n \times n}$ is invertible the condition number can be computed as

$$\kappa(\mathbf{A}) = \left\|\mathbf{A}^{-1}\right\| \|\mathbf{A}\| = \frac{\sqrt{\lambda^{\text{abs}}_{\max}(\mathbf{A}^T\mathbf{A})}}{\sqrt{\lambda^{\text{abs}}_{\min}(\mathbf{A}^T\mathbf{A})}}.$$

Furthemore if $\mathbf{A} \in \mathbb{R}^{n \times n}$ is invertible and **symmetric** then $\mathbf{A}^T = \mathbf{A}$. As a result we observe that

$$\lambda_i(\mathbf{A}^T\mathbf{A}) = \lambda_i(\mathbf{A}^2) = \lambda_i(\mathbf{A})^2 \qquad \text{for all } i = 1, \dots, n.$$

As a result

$$\kappa(\mathbf{A}) = \frac{\sqrt{\lambda_{\max}^{\text{abs}}(\mathbf{A}^T\mathbf{A})}}{\sqrt{\lambda_{\min}^{\text{abs}}(\mathbf{A}^T\mathbf{A})}} = \frac{\sqrt{\max_{i=1,\ldots,n}|\lambda_i(\mathbf{A})|^2}}{\sqrt{\min_{i=1,\ldots,n}|\lambda_i(\mathbf{A})|^2}} = \frac{\max_{i=1,\ldots,n}|\lambda_i(\mathbf{A})|}{\min_{i=1,\ldots,n}|\lambda_i(\mathbf{A})|}$$

We obtain:

Lemma 6: Condition number of symmetric matrices

If $\mathbf{A} \in \mathbb{R}^{n\times n}$ is symmetric, then

$$\kappa(\mathbf{A}) = \frac{\lambda_{\max}^{\text{abs}}(\mathbf{A})}{\lambda_{\min}^{\text{abs}}(\mathbf{A})}$$

Rounding error in a 2x2 system (continued)

As an example we now compute the condition number of the matrix analytically

$$\mathbf{A} = \begin{pmatrix} 1 & 10^{-16} \\ 1 & 0 \end{pmatrix}.$$

Recall that the value computed by Julia's `cond` function was roughly $2 \cdot 10^{16}$.

This matrix is not symmetric and we therefore use the expression

$$\kappa(\mathbf{A}) = \frac{\sqrt{\lambda_{\max}^{\text{abs}}(\mathbf{A}^T\mathbf{A})}}{\sqrt{\lambda_{\min}^{\text{abs}}(\mathbf{A}^T\mathbf{A})}},$$

which requires us to compute the eigenvalues of $\mathbf{A}^T\mathbf{A}$. We note

$$\mathbf{A}^T\mathbf{A} = \begin{pmatrix} 1 & 1 \\ 10^{-16} & 0 \end{pmatrix}\begin{pmatrix} 1 & 10^{-16} \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 10^{-16} \\ 10^{-16} & 10^{-32} \end{pmatrix}$$

and

$$\det(\mathbf{A}^T\mathbf{A} - \lambda I) = \lambda^2 - (2 + 10^{-32})\lambda + 10^{-32},$$

such that the two eigenvalues of $\mathbf{A}^T\mathbf{A}$ are

$$\lambda_{\pm}(\mathbf{A}^T\mathbf{A})) = \frac{1}{2}\left(2 + 10^{-32} \pm \sqrt{4 + 10^{-64}}\right).$$

We conclude that

$$\lambda^{\text{abs}}_{\max}(\mathbf{A}^T\mathbf{A}) \approx 2 \qquad \text{and} \qquad \lambda^{\text{abs}}_{\min}(\mathbf{A}^T\mathbf{A}) \approx \frac{10^{-32}}{2}$$

therefore

$$\kappa(\mathbf{A}) \approx 2 \cdot 10^{16},$$

— i.e. the same huge number we obtained before.

Lessons to learn about numerical stability

In the our previous discussion we noted that for a linear problem $\mathbf{Ax} = \mathbf{b}$ the condition number of the matrix $\kappa(\mathbf{A})$ provides the factor by which noise in $\mathbf{b}$ is at most amplified in the solution $\mathbf{x}$.

This concept of "condition number" or "conditioning" of a numerical problem is in fact more general as defined below:

Definition: Condition number of an algorithm

Consider an algorithm f, which relates an input quantity $\mathbf{b}$ to an output quantity $\mathbf{x}$, i.e. $\mathbf{x} = f(\mathbf{b})$. The (relative) **condition number** k of this algorithm is defined by the largest constant K satisfying the relation

$$\frac{\|f(\widetilde{\mathbf{b}}) - f(\mathbf{b})\|}{\|f(\mathbf{b})\|} \leq K \frac{\|\widetilde{\mathbf{b}} - \mathbf{b}\|}{\|\mathbf{b}\|}$$

for all valid inputs $\mathbf{b}$ and $\tilde{\mathbf{b}}$ to f.

In our case of the linear system we had that f is the LU factorisation algorithm 3 to obtain the solution $\mathbf{x}$ of the linear system $\mathbf{Ax} = \mathbf{b}$, i.e. $\widetilde{\mathbf{x}} = f(\widetilde{\mathbf{b}})$ and $\mathbf{x} = f(\mathbf{b})$. The condition number K of this algorithm (see Theorem 2) is than just $\kappa(\mathbf{A})$, the condition number of the matrix $\mathbf{A}$.

Potential confusion: Two related condition number concepts

Note that the term *condition number* both refers to the *condition number of a matrix* $\mathbf{A}$ and more generally to the *condition number of an algorithm*. While **for solving linear systems the two happen to coincide**, this is **not the case in general**.

Finally let us note:

Definiton: Well-conditioned

We call an algorithm *well-conditioned* if its condition number is small (i.e. close to $\mathbf{1}$). Else we call a problem badly conditioned.

Overview of matrix factorisations

Algorithms to factorise matrices is a **fundamental building block** of numerical methods. We so far discussed the pivoted LU factorisation $\mathbf{LU} = \mathbf{PA}$, which applies to all square matrices $\mathbf{A}$.

You know at least one more matrix factorisation, namely the eigendecomposition for diagonalisable matrices

$$\mathbf{U}\mathbf{\Lambda}\mathbf{U}^\dagger = \mathbf{A}$$

where $\mathbf{U}$ is the matrix of the eigenvectors of $\mathbf{A}$ as columns and $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues.

In fact beyond these two factorisations, there are at least 8-10 more, which you will encounter if you keep studying numerical methods for scientific or engineering problems, see the documentation of Julia functions such as `qr`, `cholesky`, `svd`, `ldlt`, `bunchkaufman` if you are curious.

Each of these factorisations have there respective advantages and disadvantages. Some are more computationally expensive to compute, some require additional properties of the matrix (e.g. positive-definite), explaining why there is such a zoo of methods to choose from.

Beyond LU and eigendecomposition we will only consider one additional matrix factorisation method in this class, namely **QR factorisation**. This factorisation applies to **rectangular matrices** $\mathbf{A} \in \mathbb{R}^{n \times m}$ with $n > m$ and factorises this matrix as $\mathbf{QR} = \mathbf{A}$ with $\mathbf{Q}$ being a matrix of m orthonormal vectors and $\mathbf{R}$ being an $m \times m$ upper-triangular matrix. We will discuss this method in detail in the context of solving Regression and curve fitting problems, where it is most commonly employed.

Comparing QR to LU factorisation the key difference is that LU only applies to square matrices, while QR can also be employed for rectangular matrices. In fact coming back to our question what Julia's `\` (backslash) operator does also make use of `qr` (see the last line of the implementation) — exactly for the case where $\mathbf{A}$ is not a square matrix.

Numerical analysis