Introduction to numerical analysis

E. Burovski

October 2020

Part I

1 Machine arithmetics

1.1 A simple worked example

Let's talk a little bit about peculiarly of using floating-point numbers on the computer. At first, we'll try to divide 2 by 3. So let's see what's the answer is.

```
In [1]: 2.0 / 3
Out[1]: 0.6666666666666666
```

The answer is 0.66666... It's clear that this number is quite close to $\frac{2}{3}$, but how close is decided by the computer, not us.

Let's take one more simple example. Imagine, we want to take a very large number like 10^{20}, add 1 and subtract 10^{20}. We expect the result to equal 1.

```
In [2]: 1e20 + 1 - 1e20
Out[2]: 0.0
```

We get 0 not 1. Actually that's not wrong. There is a reason why this simple arithmetic operations may produce unexpected results.

Let's take one more example. We know that for usual real numbers the difference of squares is identical equal to the sum times the difference:

$$a^2 - b^2 \equiv (a+b)(a-b).$$

Let's check

$$\frac{1}{\sqrt{1+x} - \sqrt{x}} = \sqrt{1+x} + \sqrt{x}$$

using the previous equality. We have the sum of square roots and it should equal one over the difference of the square roots. Let's compare the right hand side and the left hand side evaluated numerically.

```
In [10]: from math import sqrt

         def func(x):
             num = sqrt(1 + x) + sqrt(x)
             den = sqrt(1 + x) - sqrt(x)
             return 1./den - num
```

We take the right hand side - the numerator, the left hand side - 1 over the denominator, we subtract them and expect the result to be 0. Let's see what it will be numerically.

```
In [11]: for x in [1e-20, 1, 1e3, 1e10, 1e20]:
             print(x, "---", func(x))
```

We take this difference between the right hand side and the left hand side, evaluated for the range of values: 10^{-20}, 1, 1000, 10^{10}, 10^{20} and compute the result. Remember: in all cases, we expect the result to be 0. Here is the result:

```
1e-20 --- 0.0
1 --- -4.440892098500626e-16
1000.0 --- -7.219114195322618e-12
10000000000.0 --- 0.22332640280365013
---------------------------------------------------
ZeroDivisionError Traceback (most recent call last)
<ipython-input-11-0e30b6976494> in <module>
      1 for x in [1e-20, 1, 1e3, 1e10, 1e20]:
----> 2     print(x, "---", func(x))

<ipython-input-10-57b366266d1e> in func(x)
      4     num = sqrt(1 + x) + sqrt(x)
      5     den = sqrt(1 + x) - sqrt(x)
----> 6     return 1./den - num

ZeroDivisionError: float division by zero
```

Let's see, what we have. For x equal 10^{-20}, the result is 0 as expected. For x equal 1 the result is -4.44×10^{-16} that is reasonably close to 0, but how close is close. For 1000 the result is -7.22×10^{-12}. Is it close to 0? We do not know. If we take 10^{10}, the result is about 0.22, it is still larger, definitely not 0. For the last one 10^{20} we have zero division error, so, denominator is 0. The problem is $\sqrt{1+x}$ apparently is equal to $\sqrt{x}$ and that is strange.

1.2 Machine arithmetics. Representation of real numbers

Floating-point numbers are represented by a mantissa and an exponent:

$$m \times 2^p,$$

where m is a mantissa and p is an exponent, with

$$\frac{1}{2} \leq |m| < 1.$$

Now let's see how we represent both mantissa and exponent. Everything is binary in the computer. So the way we represent mantissa also known as significand is by a string of t binary digits:

$$m = \pm\mu_1\mu_2...\mu_t.$$

The first digit is the sign bit: 0 is for "+" sign, 1 is for "−" sign. Each other digit has the meaning of the corresponding power of 2, because everything is binary:

$$m = \pm(\mu_1 \times 2^{-1} + \mu_2 \times 2^{-2} + ... + \mu_t \times 2^{-t}).$$

It could be clearly seen that with this meaning the mantissa is indeed between $\frac{1}{2}$ and 1 by the absolute value.
The way we represent the exponent, a fixed-width integer, is similar:

$$p = \gamma_0\gamma_1\gamma_2...\gamma_L,$$

here γ_0 is a sign bit and the rest are the digits of the binary presentation of fixed-width integers.
Here is the way we store a number in a computer:

$$\pm\gamma_0\gamma_1\gamma_2...\gamma_L\mu_1\mu_2...\mu_t.$$

There is an IEEE standard that describes how it must work and defines several data types in terms of the bit width of the binary word:

- single precision (32 bits) - 23 bit mantissa and 8 bit exponent (float)
- double precision (64 bits) - 52 bit mantissa and 11 bit exponent (double)
- extended precision

In fact, in the vast majority of applications you would just use the double precision datatype.

The IEEE standard defines several special floating-point numbers. The first one is infinity. There are two sign advantages as positive or negative infinity.
Signed *infinities*: inf, defined as

$$x < inf, \ \forall \text{ finite } x$$

and the corresponding $-inf$.
The standard also decrease the existence of a curious object called NAN:
Not-a-number: nan, defined as

$$x = nan, \text{ if } x \neq x.$$

Is meant to represent the results of invalid computations, e.g. $1/0$. Any arithmetic computation involving a NAN results in a NAN.

1.3 Machine epsilon. Over- and underflow

The first immediate observation is that not all real numbers can be represented exactly. So we should be able to say what's the closest number which can be represented exactly.
For two numbers to be as close to each other as possible, they must only differ in the lowest bit of the mantissa:

$$\pm(\mu_1 \times 2^{-1} + \mu_2 \times 2^{-2} + ... + \underline{\mu_t \times 2^{-t}}) \times 2^p.$$

So the distance between two consecutive numbers with exponent is

$$2^{p-t}.$$

The granularity depends on the *bit-width* of the mantissa and the *value* of the exponent. As a consequence, any sort of computation incurs *rounding errors*. Therefore, we need to be aware of this rounding error in all sorts of calculation.

As the absolute value of a rounding error 2^{p-t} depends on p, there is a point to consider the relative *rounding error*

$$\frac{2^{p-t}}{2^p} = 2^{-t},$$

which only depends on the bit width of mantissa.

Definition:
Machine epsilon is the minimal number $\epsilon > 0$, for which

$$1 + \epsilon \neq 1.$$

It's clear that this epsilon is directly related to the bit width of the mantissa. Here is a ballpark estimate: for double precision $\epsilon \sim 10^{-16}$, and for single precision $\epsilon \sim 10^{-8}$. Notice that this machine epsilon is not the smallest number we can represent as a floating-point number.

$$\pm(\mu_1 \times 2^{-1} + \mu_2 \times 2^{-2} + ... + \mu_t \times 2^{-t}) \times 2^p.$$

The exponent, p, is a signed integer of L bits. The maximum value of the the exponent is

$$p_{max} = 2^{L-1},$$

L - *1* is because we use the sign bit for the exponent - it is a signed integer.

Therefore, nonzero numbers less than

$$X_0 = 2^{-p_{max}} \times 2^{-1} = 2^{-p_{max}-1}$$

cannot be represented as floating-point values.

Definition:
Machine zero is the minimum nonzero value X_0, such that

$$\frac{X_0}{2} = 0.$$

The word underflow means exactly this, it is the result of an operation where the result should be finite but gets to equal 0 because of the granularity of the floating-point numbers.

Notice that normally

$$X_0 << \epsilon.$$

The maximum value of exponent is $p_{max} = 2^{L-1}$. Therefore,

$$x > X_\infty = 2^{p_{max}}$$

cannot be represented in floating-point numbers. And here we say that the computation overflows to infinity.

Notice that X_∞ differs from the special IEEE 754 value inf.

Definition:
Machine infinity is the minimal nonzero value X_∞, such that

$$X_\infty \times 2$$

overflows.

1.4 A crude estimate of the machine epsilon

Let's estimate the value of the machine epsilon. Epsilon is the smallest possible number which we can add to a unity so that the result is still distinguishable from unity. In exact arithmetic, this loop never terminates. Let's try this in normal a double precision.

```
In [1]: eps = 1.0
        while 1 + eps != 1:
            print(eps)
            eps = eps / 2
```

```
1.0
0.5
0.25
 ...
8.881784197001252e-16
4.440892098500626e-16
2.220446049250313e-16
```

Finally, the loop terminates at the value of the order of 10^{-16}, which is actually reasonable.

So, if you are doing computations in double-precision and your results are of the order of 10^{-16}, it means that what you get is an indistinguishable from 0. In another words, 10^{-16} means 0.

2 Systems of linear algebraic equations

2.1 Systems of linear equations. Cramer's rule

Let's say we have a system of m linear equations for m unknowns $x_1, x_2, ..., x_m$

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1m}x_m = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2m}x_m = b_2 \\ ... \\ a_{m1}x_1 + a_{m2}x_2 + ... + a_{mm}x_m = b_m \end{cases}$$

where a_{ij} are real numbers, i labels the equation and j labels the unknown, and b_i are also numbers.
It's usually convenient to write the system in a matrix form:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & ... & a_{1m} \\ a_{21} & a_{22} & a_{23} & ... & a_{2m} \\ ... & & & & \\ a_{m1} & a_{m2} & a_{m3} & ... & a_{mm} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_m \end{pmatrix}$$

$$or\ \mathbf{Ax} = \mathbf{b},$$

where matrix $\mathbf{A}$ consists of coefficients a_{ij}, the column vector of unknowns is $\mathbf{x}$ and $\mathbf{b}$ consists of b_i.

Cramer's rule:
The formal solution for given system of linear equations $\mathbf{Ax} = \mathbf{b}$ is

$$x_k = \frac{\det\mathbf{A}_{(k)}}{\det\mathbf{A}},\ k = 1, 2, ..., m.$$

Here we require $\det\mathbf{A} \neq 0$ and each matrix $\mathbf{A}_{(k)}$ we form by replacing the k_{th} column by the vector $\mathbf{b}$.

The computational complexity of Cramer's rule:
A standard way to compute the determinants is using a *co-factor expansion* – you walk along the first row, take the element a_{1j}, cross out the 1st row and the i_{th} column and multiply this element a_{1j} by what remains, by the co-factor which is a determinant of order $m-1$, and then you carry on walking along the first row.
Let C_m be the number of operations to compute the determinant of order $m \times m$. Clearly

$$C_m = mC_{m-1},$$

as you need to compute m determinants of order $m-1$.
For given value of m the complexity is a factorial

$$C_m = O(m!).$$

The problem is that a factorial grows very fast. If we have a gigaflop machine that can do 10^9 operations per second, then it will take about 3.6×10^{-3} seconds to compute the determinant of order 10 and about 760 years for the determinant of order 20.
Obviously, Cramer's rule is simply not useable, so this problem requires other methods.

2.2 Gaussian elimination

Have a system of m equations for m unknowns: $x_1, x_2, ..., x_m$

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1m}x_m = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2m}x_m = b_2 \\ ... \\ a_{m1}x_1 + a_{m2}x_2 + ... + a_{mm}x_m = b_m \end{cases}$$

Or, equivalently

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & ... & a_{1m} \\ a_{21} & a_{22} & a_{23} & ... & a_{2m} \\ ... & & & & \\ a_{m1} & a_{m2} & a_{m3} & ... & a_{mm} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_m \end{pmatrix}$$

$$\mathbf{Ax} = \mathbf{b}.$$

We know that for the solution of this system to exist and be unique, we need the matrix $\mathbf{A}$ to be non-degenerate ($\det\mathbf{A} \neq 0$). One way we do it is the so-called Gaussian elimination, which is a two-step process: first, we do a forward sweep and then back substitution.

Forward sweep:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & ... & a_{1m} \\ a_{21} & a_{22} & a_{23} & ... & a_{2m} \\ ... & & & & \\ a_{m1} & a_{m2} & a_{m3} & ... & a_{mm} \end{pmatrix}$$

We want to eliminate the first unknown from the second and subsequent equations. Firstly, we need to set the element in front of x_1 to 0. Next, we need to multiply the first equation by γ_{21} ($\gamma_{21} = \frac{a_{21}}{a_{11}}$, $a_{11} \neq 0$) and subtract it from the second one.

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1m}x_m = b_1$$

$$(a_{21} - \gamma_{21}a_{12})x_1 + (a_{22} - \gamma_{21}a_{11})x_2 + ... + (a_{2m} - \gamma_{21}a_{1m})x_m = b_2 - \gamma_{21}b_1$$

Let's look at the coefficients of the second equation:

$$a_{21} - \gamma_{21}a_{12} = 0$$

$$a_{22} - \gamma_{21}a_{11} = a_{22}^{(1)}$$

$$b_2 - \gamma_{21}b_1 = b_2^{(1)}$$

Then we continue working through the first column all the way up to γ_{m1} ($\gamma_{m1} = \frac{a_{m1}}{a_{11}}$). Once we have done all these, we have eliminated all elements under a_{11}: $a_{22}, ..., a_{m1}$.

After $m-1$ steps, we arrive at

$$\mathbf{A}^{(1)}\mathbf{x} = \mathbf{b}^{(1)}$$

with

$$\mathbf{A}^{(1)} = \begin{pmatrix} \times & \times & \times & ... & \times \\ 0 & \times & \times & ... & \times \\ ... & & & & \\ 0 & \times & \times & ... & \times \end{pmatrix}$$

We managed to transform our matrix into the form, where the first unknown only enters the first equation. Then we consider smaller sub matrix (matrix $\mathbf{A}^{(1)}$ without the first row and the first column) and do the same thing. We define γ_{32} ($\gamma_{32} = \frac{a_{32^{(1)}}}{a_{22}^{(1)}}$, $a_{22}^{(1)} \neq 0$), γ_{42}, ..., γ_{m2} and eliminate all elements under $a_{22}^{(1)}$ the same way.

After $m-2$ steps, we arrive at

$$\mathbf{A}^{(2)}\mathbf{x} = \mathbf{b}^{(2)}$$

with

$$\mathbf{A}^{(2)} = \begin{pmatrix} \times & \times & \times & ... & \times \\ 0 & \times & \times & ... & \times \\ 0 & 0 & \times & ... & \times \\ ... & & & & \\ 0 & 0 & \times & ... & \times \end{pmatrix}$$

We have transformed our system of equations into the form, where all elements in the second column below the main diagonal are 0. Then we proceed with the smallest sub matrix and so on.
Having annihilated $m-1$ columns, we arrive at

$$\mathbf{A}^{(m-1)}\mathbf{x} = \mathbf{b}^{(m-1)}$$

with an *upper triangular* matrix and call it $\mathbf{U}$

$$\mathbf{A}^{(m-1)} = \begin{pmatrix} \times & \times & \times & ... & \times & \times \\ 0 & \times & \times & ... & \times & \times \\ 0 & 0 & \times & ... & \times & \times \\ ... & & & & & \\ 0 & 0 & 0 & ... & \times & \times \\ 0 & 0 & 0 & ... & 0 & \times \end{pmatrix} = \mathbf{U}$$

and this concludes the forward sweep.

Back substitution:

$$\begin{pmatrix} \times & \times & \times & ... & \times & \times \\ 0 & \times & \times & ... & \times & \times \\ 0 & 0 & \times & ... & \times & \times \\ ... & & & & & \\ 0 & 0 & 0 & ... & \times & \times \\ 0 & 0 & 0 & ... & 0 & \times \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ ... \\ x_{m-1} \\ x_m \end{pmatrix} = \begin{pmatrix} \times \\ \times \\ \times \\ \times \\ ... \\ \times \end{pmatrix}$$

Let's look at the last equation. From this, we quiet easily get x_m, which is just a ratio of two numbers. Then we will look at the ultimate equation. It contains the last and the second to last unknowns, but we already know the last one, so, from this equation we get x_{m-1}. Next, we move upwards all the way up to x_1 .

Let's count the computational complexity of these Gaussian elimination.
Forward sweep:
1st row: multiply $m-1$ elements, $a_{1k}, k = 2, ..., m$ by $\gamma_{21} \rightarrow m-1$ multiplies.
2nd row: $m-1$ multiplies.
...

Total for the 1st column: $(m-1)^2$ operations.
Total for the 2nd column: $(m-2)^2$ operations.
...

Total for the whole forward sweep:

$$(m-1)^2 + (m-2)^2 + ... \sim O(m^3)$$

Back substitution:
m-th row: 1 multiplication.
$(m-1)$-th row: 2 multiplications.
...

Total is $\sim O(m^2)$.
Comared to the forward sweep, back substitution is cheap.

2.3 LU decomposition: the matrix form of the Gaussian elimination

Remembering the algorithm of Gaussian elimination, we have $\mathbf{Ax} = \mathbf{b} \rightarrow \mathbf{A}^{(1)}\mathbf{x} = \mathbf{b}^{(1)}$, where

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & ... & a_{1m} \\ a_{21} & a_{22} & a_{23} & ... & a_{2m} \\ ... & & & & \\ a_{m1} & a_{m2} & a_{m3} & ... & a_{mm} \end{pmatrix},$$

and we have our gammas $\gamma_{21} = a_{21}/a_{11}, \gamma_{31} = a_{31}/a_{11}, ..., \gamma_{m1} = a_{m1}/a_{11}$, which were used to introduce zeros to the first column.
Now we can construct a lower triangular matrix $\mathbf{L_1}$:

$$\mathbf{L_1} = \begin{pmatrix} 1 & 0 & 0 & ... & 0 \\ -\gamma_{21} & 1 & 0 & ... & 0 \\ -\gamma_{31} & 0 & 1 & ... & 0 \\ ... & & & & \\ -\gamma_{m1} & 0 & 0 & ... & 1 \end{pmatrix}.$$

It is easy to see that the transformation of the first column of the Gaussian forward sweep is equivalent to left multiplying both the left hand-side and the right-hand side of the system by $\mathbf{L_1}$:

$$\mathbf{L_1 Ax} = \mathbf{L_1 b}.$$

Likewise, we can construct a lower triangular matrix $\mathbf{L_2}$:

$$\mathbf{L_2} = \begin{pmatrix} 1 & 0 & 0 & 0 & ... & 0 \\ 0 & 1 & 0 & 0 & ... & 0 \\ 0 & -\gamma_{32} & 1 & 0 & ... & 0 \\ 0 & -\gamma_{42} & 0 & 1 & ... & 0 \\ ... & & & & & \\ 0 & -\gamma_{m2} & 0 & 0 & ... & 1 \end{pmatrix}.$$

Here the transformation of the second column of the Gaussian forward sweep is also equivalent to left multiplying both the left hand-side and the right-hand side of the system by $\mathbf{L_2}$:

$$\mathbf{L_2 L_1 A x} = \mathbf{L_2 L_1 b}.$$

The whole forward sweep is equivalent to

$$\mathbf{L_{m-1} ... L_1 A x} = \mathbf{L_{m-1} ... L_1 b}.$$

Now we have a string of this lambdas acting on $\mathbf{A}$, and the result is an upper triangular matrix $\mathbf{U}$ because that's what a Gaussian elimination is for, so

$$\mathbf{L_{m-1} ... L_1 A} = \mathbf{U},\ \mathbf{L_{m-1} ... L_1 b} = \mathbf{b}^{(m-1)}.$$

What we get is

$$\mathbf{U x} = \mathbf{b}^{(m-1)}.$$

From this point on, we can start with our back substitution in the usual way. We know that

$$\mathbf{A} = (\mathbf{L_{m-1} ... L_1})^{-\mathbf{1}} \mathbf{U}.$$

Then

$$\mathbf{A} = \mathbf{L_1^{-1} ... L_{m-1}^{-1}} \mathbf{U}.$$

The point is that we can simplify this expression. Now we'll see what $\mathbf{L_k}$ looks like.
Actually, it's very easy to check that the inverse of $\mathbf{L_1}$ looks very similar to $\mathbf{L_1}$ itself:

$$\mathbf{L_1^{-1}} = \begin{pmatrix} 1 & 0 & 0 & ... & 0 \\ \gamma_{21} & 1 & 0 & ... & 0 \\ \gamma_{31} & 0 & 1 & ... & 0 \\ ... & & & & \\ \gamma_{m1} & 0 & 0 & ... & 1 \end{pmatrix}.$$

Likewise, the structure of $\mathbf{L_2^{-1}}$ is very similar to $\mathbf{L_2}$:

$$\mathbf{L_2} = \begin{pmatrix} 1 & 0 & 0 & 0 & ... & 0 \\ 0 & 1 & 0 & 0 & ... & 0 \\ 0 & \gamma_{32} & 1 & 0 & ... & 0 \\ 0 & \gamma_{42} & 0 & 1 & ... & 0 \\ ... & & & & & \\ 0 & \gamma_{m2} & 0 & 0 & ... & 1 \end{pmatrix}.$$

And so on.
If we take the product of these matrices we'll get following:

$$\mathbf{L_1^{-1} L_2^{-1} ... L_{m-1}^{-1}} = \begin{pmatrix} 1 & 0 & 0 & ... & 0 \\ \gamma_{21} & 1 & 0 & ... & 0 \\ \gamma_{31} & \gamma_{32} & 1 & ... & 0 \\ ... & & & & \\ \gamma_{m1} & \gamma_{m2} & \gamma_{m3} & ... & 1 \end{pmatrix}.$$

It is a lower triangular matrix $\mathbf{L}$.
Now what we have is $\mathbf{A} = \mathbf{LU}$.
To conclude, what we do to solve a system of linear equations $\mathbf{Ax} = \mathbf{b}$:

- Decompose the l.h.s. matrix $\mathbf{A}$: $\mathbf{A} = \mathbf{LU}$
- Transform the r.h.s., compute $\mathbf{L^{-1}b}$
- Solve $\mathbf{Ux} = \mathbf{L^{-1}b}$

The first step is cubic in complexity and next two steps are actually cheap.

2.4 When does the Gaussian elimination work?

Let's see, what are the requirements for Gaussian elimination to work. We will be dividing things by this leading element a_{11}. So obviously we need to require that $a_{11} \neq 0$.

$\mathbf{A} \to \mathbf{A}^{(1)}$:

$$\begin{matrix} a_{11} & a_{12} & ... \\ a_{21} & a_{22} & ... \\ ... & ... & ... \end{matrix}$$

When working on the second column, we will be dividing things by the leading element, which here is $a_{22}^{(1)}$, so, it is also need not to be 0. And it is explicitly given by the linear combination of the elements from the top left corner of our matrix.

$\mathbf{A}^{(1)} \to \mathbf{A}^{(2)}$:

$$\begin{matrix} a_{11} & a_{12} & ... \\ 0 & a_{22}^{(1)} & ... \\ ... & ... & ... \end{matrix}$$

remember: $a_{22}^{(1)} = a_{22} - \frac{a_{21}}{a_{11}} a_{12} \neq 0$.
That is, we need to require that the leading minor of the order 2 is non-zero.

In fact, this whole process of Gaussian elimination requires that all leading minors of our matrix are non-zero.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & ... & a_{1m} \\ a_{21} & a_{22} & a_{23} & ... & a_{2m} \\ ... & & & & \\ a_{m1} & a_{m2} & a_{m3} & ... & a_{mm} \end{pmatrix}$$

This is quite a bit of a limitation. However, there is a way around it and that is coming up in the next paragraph.

2.5 LU decomposition with pivoting. Permutation matrices

- Solution of $\mathbf{A}x = b$ exists and is unique iff $det\mathbf{A} \neq 0$;
- However, LU factorization requires all leading minors to be non zero;
- If a leading minor close to zeo, it is also bad for stability.

The main idea: at each step, find a coefficient a_{ij} with the largest absolute value (pivot), and eliminate the corresponding unknown x_j.
Column pivoting (or partial pivoting): look for a max element along columns of $\mathbf{A}$:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & ... & a_{1m} \\ a_{21} & a_{22} & a_{23} & ... & a_{2m} \\ ... & & & & \\ a_{m1} & a_{m2} & a_{m3} & ... & a_{mm} \end{pmatrix}$$

If you work with the first column you scan the first column to find the maximum element. And then you swap the first and the this rows and then you proceed as usual. Complexity: $O(m)$ comparisons per column, it is cheap.

One other possibility is the so-called full pivoting: look for a max element in the whole matrix $\mathbf{A}$. Swap rows and relabel the unknowns. Complexity: $O(m^2)$ comparisons per column and this is not very cheap. Because of this complexity, full pivoting is very rarely used.

Permutation matrices. An elementary permutation matrix, P_{kl}, is a square matrix obtained from an identity matrix by swapping rows k and l.
$P_{kl}\mathbf{A}$ has rows k and l swapped, while $\mathbf{A}P_{kl}$ has columns k and l swapped.

Then we can cast the Gaussian elimination with pivoting and the sequence of multiplication by this triangular matrices and permutation matrices. The process of constructing the LU factorization looks like this: for the k-th column, find a pivot (row i_k), left-multiply by $P_k = P_{k,i_k}$, then left-multiply by $\mathbf{\Lambda}_k$. Proceed for $k = 1, ..., m-1$.

$$\mathbf{A}x = b$$

$$\mathbf{\Lambda}_1 P_1 \mathbf{A}x = \mathbf{\Lambda}_1 P_1 b$$

$$\mathbf{\Lambda}_2 P_2 \mathbf{\Lambda}_1 P_1 \mathbf{A}x = \mathbf{\Lambda}_2 P_2 \mathbf{\Lambda}_1 P_1 b$$

...

Finally,

$$\mathbf{A}^{(m-1)} = \mathbf{\Lambda}_{m-1} P_{m-1} ... \mathbf{\Lambda}_2 P_2 \mathbf{\Lambda}_1 P_1 \mathbf{A}.$$

For any matrix $\mathbf{A}$ with $det\mathbf{A} \neq 0$, factorization

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

exists and is unique.

In other words, if your matrix **A** is non-singular then there exists a permutation which makes all leading minors to be non-zero and then **LU** decomposition works.