

# INTRO TO SCIENTIFIC COMPUTING

---

QUANTITATIVE ECONOMICS 2025

Piotr Żoch

October 21, 2025

# INTRODUCTION

- We will mostly deal with numbers.
- Important to know how computers store and manipulate them.
- We will also introduce some basic concepts related to numerical analysis.

## FLOATING POINT NUMBERS

- Floating point numbers are ubiquitous in scientific computing.
- Useful to have a basic understanding of them.

## FLOATING POINT NUMBERS

- $\mathbb{R}$  (real numbers) is large, but computers have finite memory.
- We need to find some representation of  $\mathbb{R}$ .
- Fix a **base**  $\beta$  and integers  $p, m, M$ . Define a **floating point system** as

$$\pm \left( \sum_{n=1}^p d_n \beta^{-n} \right) \times \beta^e,$$

where  $d_n \in \{0, 1, \dots, \beta - 1\}$ ,  $m \leq e \leq M$ .

- We call elements of the **floating point system** **floating point numbers** or **floats**.

## FLOATING POINT NUMBERS

$$\pm \left( \sum_{n=1}^p d_n \beta^{-n} \right) \times \beta^e, \quad \text{where } d_n \in \{0, 1, \dots, \beta - 1\}, m \leq e \leq M.$$

- Why “floating point”? By varying  $e$  we can represent numbers by shifting the decimal point.
- Example: Consider  $\beta = 10$  and two numbers  $x_1 = 0.12 \times 10^1$  and  $x_2 = 0.12 \times 10^2$ .  
These are  $x_1 = 1.200$  and  $x_2 = 12.00$ . The location of the decimal point is different.

## FLOATING POINT NUMBERS

- Focus on  $\beta = 2$ . We have **binary** floating point numbers.

$$\# = (-1)^s \times 2^e \times 1.f, \quad f = \left( \sum_{n=1}^p d_n 2^{-n} \right).$$

- We use the following names:
  - $f$  the **mantissa** or **significand**;
  - $e$  is the **exponent**;
  - $p$  is the **precision**;
  - $s$  is the **sign**.

## IEEE 754

- IEEE 754: technical standard for floating-point arithmetic established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE).
- Defines formats for floating point numbers with  $\beta = 2$ .
- Two formats: **single precision** (stored in 32 bits) and **double precision** (stored in 64 bits).
- A bit is a binary digit, i.e., 0 or 1.
- `Float32` and `Float64` in Julia.

## DOUBLE PRECISION FORMAT

- The format for a double precision number is:

$$\# = (-1)^s \times 2^{e-1023} \times 1.f$$

- $s$  is the sign bit (1 bit),  $e$  is the exponent (11 bits), and  $f$  is the fraction (52 bits).
- Note that only combinations of powers of 2 can be expressed exactly.
- Note the bias of 1023 in the exponent.



## DOUBLE PRECISION FORMAT

- Consider the number 1.0.
- It has  $s = 0$ ,  $e = 1023$ , and  $f = 0$ .

$$1.0 = (-1)^0 \times 2^{(1023-1023)} \times 1.0$$

$$\rightarrow \underbrace{0}_{\text{sign}} \underbrace{001111111111}_{1023 \text{ in base 2}} 0000 \dots 0000$$

## DOUBLE PRECISION FORMAT

- Consider the number 2.0.
- It has  $s = 0$ ,  $e = 1024$ , and  $f = 0$ .

$$2.0 = (-1)^0 \times 2^{(1024-1023)} \times 1.0$$

$$\rightarrow \underbrace{0}_{\text{sign}} \underbrace{100000000000}_{1024 \text{ in base 2}} 0000 \dots 0000$$

## DOUBLE PRECISION FORMAT

- Consider the number 0.5.
- It has  $s = 0$ ,  $e = 1022$ , and  $f = 0$ .

$$0.5 = (-1)^0 \times 2^{(1022-1023)} \times 1.0$$

$$\rightarrow \underbrace{0}_{\text{sign}} \underbrace{001111111110}_{1022 \text{ in base 2}} 0000 \dots 0000$$

## DOUBLE PRECISION FORMAT

- Consider the number 0.2.
- it has  $s = 0$ ,  $e = 1020$ , and  $f = 0.6$ .

$$0.2 = (-1)^0 \times \underbrace{2^{(1020-1023)}}_{=\frac{1}{8}} \times (1 + 0.6)$$

- This one **does not** have an exact binary representation.
- The problem is 0.6. How is it represented in binary?
- Note: it is similar to representing  $1/3$  in the decimal system.

## DOUBLE PRECISION FORMAT

- How to represent fractions (here 0.6) in binary?
  1. Multiply by 2 ( $2 \times 0.6 = 1.2$ ), record the integer part (1)
  2. Multiply the fraction part by 2 ( $2 \times 0.2 = 0.4$ ), record the integer part (0).
  3. Multiply the fraction part by 2 ( $2 \times 0.4 = 0.8$ ), record the integer part (0).
  4. Multiply the fraction part by 2 ( $2 \times 0.8 = 1.6$ ), record the integer part (1).
  5. Multiply the fraction part by 2 ( $2 \times 0.6 = 1.2$ ), record the integer part (1).
  6. Continue until the fraction part is 0.
  7. The binary representation is the integer parts of the results.

## DOUBLE PRECISION FORMAT

- Note that the binary representation of 0.6 is **infinite** ( $0.\overline{1001}$ ).
- We use only 52 bits for the fraction part.
- We then write 0.6 as  $0.\underbrace{1001\ 1001\ \dots\ 1010}_{52\ 0s\ and\ 1s}$  where we have rounded the repetitive end to the nearest binary number 1010.

## DOUBLE PRECISION FORMAT

- The largest  $e$  value is  $1111\ 1111\ 111 = 2047$ .
  - When  $f = 0$  we use this to represent  $\infty$ .
  - When  $f \neq 0$  we use this to represent NaN (not a number).
- The largest positive double precision number has  $s = 0, e = 2046$ ,  
 $f = 1111 \dots 1111 = 1 - 2^{-52}$ . It is  $\approx 1.797710^{308}$
- **Overflow** occurs when a number is too big to be represented.
- Usually the result will be represented as  $\infty$ .

## DOUBLE PRECISION FORMAT

- The smallest  $e$  value is  $0000\ 0000\ 000 = 0$ .
- It is reserved to represent numbers for which representation changes from  $1.f$  to  $0.f$  (*denormalized numbers*)
- The smallest positive double precision number has  $s = 0, e = 1, f = 0000 \dots 0000$ . It is  $\approx 2.225110^{-308}$
- **Underflow** occurs when a (positive) number is too small to be represented.
- Usually the result will be represented as 0.0.



## MACHINE EPSILON

- Machine epsilon,  $\epsilon$  is the distance between 1 and the next largest number that can be represented.
- For any  $0 < \delta < \epsilon/2$  we have  $1 + \delta$  represented as 1.
- In double precision:  $\epsilon \approx 2.2204 \times 10^{-16}$ .
- In Julia `eps(Float64)`.
- Note: the distance between 1 and the next smallest number (i.e., the number just below 1) is  $\epsilon/2$ .
- Generally: numbers in double precision are not equally spaced.

## DIGITS OF PRECISION

- In double precision format we have 52 bits for the mantissa.
- $2^{52} = 4,503,599,627,370,496$ . We can represent all numbers with 15 digits and some with 16 digits.
- Exponent just shifts the decimal point.
- That means doubles have between 15 and 16 digits of precision.
- Implication: numbers like 10000000000000000001 and 10000000000000000002 are stored as the same float.

## ERRORS

- **Takeaway:** numbers stored on a computer are approximations.
  - Let  $\tilde{x}$  be the approximation of  $x$  (for example, as a floating point number).
  - The **absolute error** is  $|\tilde{x} - x|$ .
  - The **relative error** is  $|\tilde{x} - x|/|x|$ .
- 
- Let  $fl(x)$  be the floating point representation of  $x$ .
  - We have  $fl(x) = x(1 + \delta)$ , with  $|\delta| \leq \epsilon/2$ .  $\delta$  is the **relative error**.
  - Similarly, let  $\odot$  be  $+$ ,  $-$ ,  $\cdot$ ,  $/$ . We have  $fl(x \odot y) = (x \odot y) (1 + \delta)$ .

## ROUNDING ERROR

- Suppose we add two numbers  $x$  and  $y$ . The result will be represented as  $(x + y)(1 + \delta)$ . What is  $\delta$ ?

$$\begin{aligned}(x + y)(1 + \delta) &= fl(fl(x) + fl(y)) \\ &= fl(x(1 + \delta_x) + y(1 + \delta_y)) \\ &= [x(1 + \delta_x) + y(1 + \delta_y)](1 + \delta_{x+y})\end{aligned}$$

so

$$\delta = \frac{x\delta_x + y\delta_y + (x + y)\delta_{x+y}}{x + y}.$$

## ROUNDING ERROR

- Let  $|\delta_x|, |\delta_y|, |\delta_{x+y}| \leq \epsilon$ . Then

$$|\delta| \leq \frac{|x| + |y| + |x + y|}{|x + y|} \epsilon.$$

- Caution: the relative error can be **much larger** than  $\epsilon$ .
- **Catastrophic cancellation.**
- This happens when  $x \approx -y$ .

## ORDER OF OPERATIONS MIGHT MATTER

- By representing numbers as floats, operations **cease** to be associative and distributive.
- We do not necessarily have  $(x + y) + z = x + (y + z)$ .
- We do not necessarily have  $(x + y) \cdot z = x \cdot z + y \cdot z$ .
- Order of operations might matter. Start addition from small numbers.

## EXAMPLE

- Suppose we want to find the roots of the quadratic equation

$$ax^2 + bx + c = 0.$$

- We could use the formula:  $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ .
- Problems:
  1.  $b^2$  and  $4ac$  might be close to each other.
  2. If  $4ac \approx 0$ ,  $b$  and  $\sqrt{b^2 - 4ac}$  might be close to each other.
- Solution?

## LOSS OF PRECISION

- Interested in solving  $x^2 - 26x + 1 = 0$ . True solution:

$$x_1 = 13 - \sqrt{168} \approx 0.03852, \quad x_2 = 13 + \sqrt{168} \approx 25.961.$$

- Assume we use a format with 5 significant digits:  $\sqrt{168}$  is represented as 12.961.
- We calculate the first root as  $13.000 - 12.961 = 0.03900$  instead of 0.03852.
- Only two digits (0 and 3) are correct. We lost three digits of precision.
- Instead, we can use  $13 - \sqrt{168} = \frac{1}{13 + \sqrt{168}}$  to get a more accurate result: 0.03852.



## CONDITION NUMBERS

- Let  $x \in \mathbb{R}$  be data and  $f : \mathbb{R} \rightarrow \mathbb{R}$  be a function.
- Recall we have  $f(x) = x(1 + \delta)$  with  $|\delta| \leq \epsilon/2$ .
- We are interested in how much the output of  $f$  changes when the input changes.
- We can measure it as

$$\frac{\frac{|f(x) - f(x(1+\delta))|}{|f(x)|}}{\frac{|x - x(1+\delta)|}{|x|}}$$

- It looks like elasticity of  $f$  with respect to  $x$ .

## CONDITION NUMBERS

- The expression can be simplified to

$$\frac{|f(x + \delta x) - f(x)|}{|\delta f(x)|}$$

- Take the limit as  $\delta \rightarrow 0$  and suppose  $f$  is differentiable. We define the **relative condition number** as

$$\kappa_f(x) = \left| \frac{xf'(x)}{f(x)} \right|$$

- For small  $\delta$  we have

$$\frac{|f(x + \delta x) - f(x)|}{|f(x)|} \approx \kappa_f(x) |\delta|.$$

- The relative perturbation in the input,  $\delta$ , is amplified by the relative condition number.

## CONDITION NUMBERS

- If the relative condition number is large, we call a problem **ill-conditioned**. Otherwise, we call it **well-conditioned**.
- In an ill-conditioned problem, small perturbations in the input can lead to large changes in the output.
- If  $\kappa_f(x) = 10^k$ , you might lose up to  $k$  digits of accuracy due to  $f$  itself.
- For example, if  $\kappa_f(x) = 10^{16}$  `Float64` is useless.

## CONDITION NUMBERS

- Most problems have more than one input and output.
- We can generalize the concepts of relative condition numbers to accomodate these cases.
- We will also see later how to extend the concept of condition numbers to matrices.

## CONDITION NUMBERS

- Consider again the quadratic equation  $ax^2 + bx + c = 0$ .
- Let's pick one root,  $x_1$  and consider what happens to it as we vary  $a$ .
- We have  $f(a) = x_1$  with  $f'(a) = -\frac{x_1^2}{2ax_1 + b}$ .
- The condition number is

$$\kappa_f(a) = \left| \frac{ax_1}{2ax_1 + b} \right| = \left| \frac{x_1}{x_1 - x_2} \right|$$

- The problem is ill-conditioned when  $x_1 \approx x_2$ .
- Think of the extreme case with a repeated root.

## LITTLE “OH” - BIG “OH”

- Let  $f : \mathbb{N} \rightarrow \mathbb{R}_+$  and  $g : \mathbb{N} \rightarrow \mathbb{R}_+$ .
- We say that  $f = \mathcal{O}(g)$  ( $f$  is “big-Oh” of  $g$ ) if there exists  $c > 0$  and  $n_0 \in \mathbb{N}$  such that

$$f(n) \leq cg(n) \text{ for all } n \geq n_0.$$

- This says that the ratio  $f(n)/g(n)$  is bounded from above as  $n \rightarrow \infty$ .
- $f(n)$  might be much smaller than  $g(n)$ , this is just a bound.

## LITTLE “OH” - BIG “OH”

- We say that  $f = o(g)$  ( $f$  is “little-oh” of  $g$ ) if for any  $c > 0$  there exists  $n_0 \in \mathbb{N}$  such that

$$f(n) \leq cg(n) \text{ for all } n \geq n_0.$$

- For strictly positive  $g$  this says that the ratio  $f(n)/g(n)$  goes to zero as  $n \rightarrow \infty$ .

## LITTLE “OH” - BIG “OH”

- Let  $f(n) = a_1n^3 + b_1n^2 + c_1n$  and  $g(n) = a_2n^3$ .
- We have

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \frac{a_1}{a_2}.$$

- This means that  $f = \mathcal{O}(g)$ .
- At the same time  $f$  is not  $o(g)$ .



# FLOPS

- A **flop** is a **floating point operation**.
- We count flops to measure the complexity of an algorithm.
- Suppose  $A$  is an  $n \times n$  matrix and  $b$  is an  $n \times 1$  vector.
- We want to calculate  $Ab$ .
- To calculate one element of  $Ab$ :

$$\sum_{i=1}^n A_{ij}b_j.$$

- There are  $n$  multiplications and  $n - 1$  additions.
- We need to do it  $n$  times. In total we have  $n(n + n - 1)$  flops, or  $\mathcal{O}(n^2)$ .

## FLOPS

- If the run time of an algorithm is dominated by flops, we expect

$$\text{run time} \approx c \times \text{flops}$$

for some constant  $c$ .

- In our example, if  $n_1 = 1000$  and  $n_2 = 2000$ , we expect the run time of the algorithm for  $n_2$  to be four times longer than for  $n_1$ .

## FLOPS

- Suppose  $A, B$  are  $n \times n$  matrices. We want to calculate  $AB$ .
- To calculate one element of  $AB$ :

$$\sum_{k=1}^n A_{ik} B_{kj}.$$

- There are  $n$  multiplications and  $n - 1$  additions.
- We need to do it  $n^2$  times. In total we have  $2n^3$  flops, or  $\mathcal{O}(n^3)$ .
- In our example, if  $n_1 = 1000$  and  $n_2 = 2000$ , we expect the run time of the algorithm for  $n_2$  to be eight times longer than for  $n_1$ .
- Matrix multiplication using this algorithm is expected to take  $n$  times longer than matrix-vector multiplication.