Floating-Point Arithmetic

2026-05-06 · cheerful mango Haubentaucher

Getting used to Errors Everywhere

The Why

We saw that numerical methods introduce errors through approximation.

Where $f(x)$ be a continuous function on $[a, b]$. The discretized version $f(x_i)$ approximates $f(x)$ at discrete points x_i with an error that depends on the step size h and the smoothness of f. Numerical values can further only be stored approximately in a computer's memory, in floating-point representation. This leads to rounding errors when performing arithmetic operations, adding up to the truncation errors we already experienced when approximating infinite processes with finite ones, it is also called approximation error.

Modeling Error occurs as all models are simplifications of reality, and the difference between the model and the real-world system introduces an error. We will not dive deeper into this type of error in this course. Yet, mind the fine difference between what we term $f(x)$ and what actually is a $f(\tilde{x})$.

This gives us good reason to understand these errors and their interplay with our machines.

- Numerical methods introduce rounding and truncation errors.
- Based on our machines these errors play out differently, and can amplify and accumulate.

In machine learning and artificial intelligence, you already are aware thatnumerical methods are crucial for training models, but of course floating-point arithmetic

is as relevant in model inference. Especially in compute-sparse environments on the edge, or when deploying quantized models [1], understanding floating-point arithmetic and the underlying mechanics comes handy.

On the Edge models use lower-precision arithmetic, such as 8-bit integers or even binary weights and activations.

Binary neural networks (BNNs) use weights and activations constrained to $\{-1, +1\}$, drastically reducing memory and computation requirements, but making floating-point representation and rounding effects critical.

[1]

Hands On Experience

Let's get a feeling for rounding error with a simple example.

Consider dividing 10 by 3. Write out the decimal expansion:

$$
\begin{aligned}
&10 \div 3 = 3 \quad \text{remainder: } 1 \\
&\text{Bring down a 0: } 10 \div 3 = 3 \quad \text{remainder: } 1 \\
&\text{Bring down a 0: } 10 \div 3 = 3 \quad \text{remainder: } 1 \\
&\text{Bring down a 0: } 10 \div 3 = 3 \quad \text{remainder: } 1 \\
&\text{... and so on} \\
&\Rightarrow \frac{10}{3} = 3.333\ldots
\end{aligned}
$$

The 3s repeat forever. But when you write it down or enter it into a calculator or computer, you have to stop at some point:

$$\frac{10}{3} \approx 3.333 \quad \text{(rounded or truncated after 3 digits)} \tag{1}$$

What you will end up with is the rounding error: The difference between the true value and the value you get when you cut off (truncate) the expansion. No matter how many digits you write, as soon as you stop, you introduce an error:

$$\text{Rounding error} = |\, 3.333333\ldots - 3.333 \,| = 0.000333\ldots \tag{2}$$

The more digits you keep, the smaller the error, but it never disappears completely unless you write infinitely many digits, which well, you know is impossible.

Types of numerical representations in computers include:
Integer (int):
3
Floating-point (float):
3.3333333
Double precision (double):
3.3333333333333333
Fixed-point (e.g. 4-digits):
3.3333

Impossible? Mathematical annotation helps us with period: $3.\overline{3}$.

This is rounding error: Computers always store numbers with a finite number of digits, so rounding errors will inevitably show up and need to be managed.

Because as you will see, these small errors can accumulate and amplify and lead to significant inaccuracies in computations, especially in iterative algorithms common in numerical methods and machine learning.

The Learning Objectives of this chapter aim at providing you with the abilities to:

- Understand the different types of numerical errors: Modeling, truncation, and rounding errors.
- Comprehend floating-point representation, machine epsilon, and loss of significance.
- Be able to handle numerical errors in practical computations.

Floating Point Representation and Precision

FLOATING-POINT NUMBERS are a way for computers to represent real numbers using a finite number of bits. The IEEE 754[2] standard is the most widely used format. A floating-point number is typically stored as:

$$x = (-1)^s \cdot m \cdot 2^e, \tag{3}$$

where s is the sign bit, m is the mantissa (or significand) which determines the precision, and e is the exponent which determines the scale (or magnitude) of the number. This allows for a wide range of values, but only a finite set of real numbers can be represented exactly. In IEEE 754 single precision (float), 32 bits are divided into 1 sign bit, 8 exponent bits, and 23 mantissa bits.

[2]

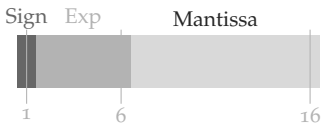

Figure 1: Bit layout of IEEE 754 **HALF (16)**: sign (dark), exponent (medium), mantissa (light).

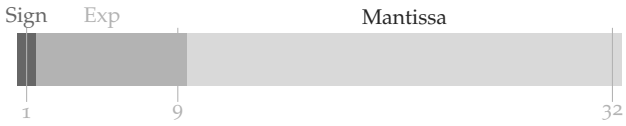

Figure 2: Bit layout of IEEE 754 **SINGLE (32)**: sign (dark), exponent (medium), mantissa (light).

Figure 3: Bit layout of IEEE 754 **DOUBLE (64)**: sign (dark), exponent (medium), mantissa (light).

A BIT, short for binary digit, is the most basic unit of information in computing and digital communications. It can have a value of either 0 or 1.

THE EXPONENT is stored with a bias to allow both positive and negative exponents. This allows floating-point numbers to represent both very small and very large magnitudes. For example, in IEEE 754 single precision, the exponent uses 8 bits and a bias of 127. The stored exponent E is related to the true exponent e by $e = E - 127_{10}$. The reason for the bias of 127 is that with 8 bits, the exponent field can store values from 0 ($2^0 - 1$) to 255 ($2^8 - 1$). By subtracting the bias (127), the actual exponent e can take both positive and negative values, centered around zero. This makes encoding and comparison of floating-point numbers easier in hardware.

CONSTRUCTING SCALE in floating-point representation:

$$10^1 = 10_{10} = 1010_2 = 1.010 \times 2^3, \quad E = 10000010_2$$
$$10^2 = 100_{10} = 1100100_2 = 1.100100 \times 2^6, \quad E = 10000101_2$$
$$10^3 = 1000_{10} = 1111101000_2 = 1.111101000 \times 2^9, \quad E = 10001000_2$$

THE LARGEST NUMBER IN SINGLE PRECISION is about 10^{38}, set by the largest exponent $e = +127$. The decimal exponent 38 comes from $\log_{10}(2^{128}) \approx 38.5$.

REMEMBER, mega (10^6), giga (10^9), tera (10^{12}), peta (10^{15}), exa (10^{18}), zetta (10^{21}), yotta (10^{24}), ronna (10^{27}), quetta (10^{30}),
no official SI prefix for (10^{33}).

THE MANTISSA determines how finely numbers can be represented between powers of two.

2-BIT MANTISSA EXAMPLE (UNNORMALIZED):

Mantissa bit combinations: 00, 01, 10, 11

Unnormalized significand: $0.xx_2$

$$\begin{aligned}
00: \quad 0.00_2 &= 0 + 0 \times 2^{-1} + 0 \times 2^{-2} = 0.0 \\
01: \quad 0.01_2 &= 0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 0.25 \\
10: \quad 0.10_2 &= 0 + 1 \times 2^{-1} + 0 \times 2^{-2} = 0.5 \\
11: \quad 0.11_2 &= 0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 0.75
\end{aligned}$$

A TWO BIT MANTISSA means we can represent four distinct values between any two powers of two. A three bit mantissa allows eight distinct values, and so on. In general, with a mantissa of t bits, we can represent 2^t distinct values between any two powers of two, scaled up by the exponent.

MACHINE EPSILON ($\varepsilon_{\text{mach}}$) is the smallest positive number such that $1 + \varepsilon_{\text{mach}} \neq 1$ in the computer's arithmetic. It quantifies the upper bound on relative error due to rounding in floating-point arithmetic:

$$\varepsilon_{\text{mach}} = 2^{-t}, \tag{4}$$

where t is the number of bits in the mantissa.
In our 2-bit Mantissa example this is:

$$\varepsilon_{\text{mach}} = 2^{-2} = 0.25$$

For IEEE 754 single precision, $\varepsilon_{\text{mach}} \approx 1.19 \times 10^{-7}$; for double precision, $\varepsilon_{\text{mach}} \approx 2.22 \times 10^{-16}$.

THIS MEANS that the relative precision of floating-point numbers is approximately 2^{-t}, while the absolute precision depends on the magnitude of the number being represented:

$$\varepsilon_{\text{mach}} \cdot |x|. \tag{5}$$

ABSOLUTE PRECISION is just about to become clear, $1 : 2 = 0.5$ but $10 : 2 = 5$. Same number of steps (relative precision), but gaps of 0.5 vs 5.

In other words large numbers have larger absolute gaps between representable values than small numbers.

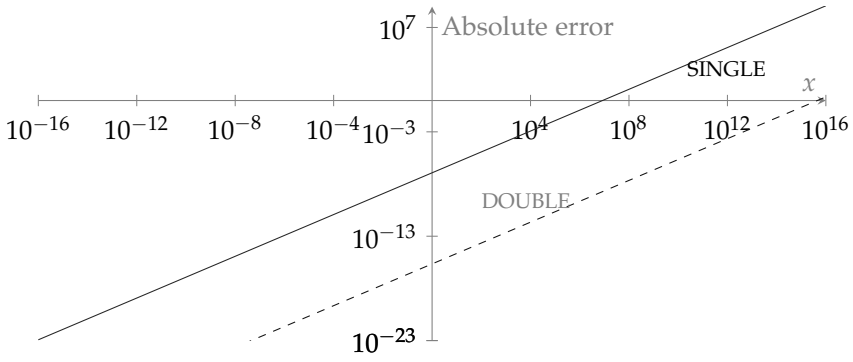

Figure 4: Absolute error for single (black, labeled SINGLE) and double (gray, dashed, labeled DOUBLE) precision as a function of the represented value x. The error grows linearly with x and is proportional to machine epsilon for each format.

Fixed-point representation is another way to store real numbers in computers, especially when you want predictable precision and performance. In fixed-point, you decide in advance how many bits are used for the integer part and how many for the fractional part. This means the gap between representable numbers (the precision) is always the same, no matter how large or small the value. Which gives more control.

Example: Suppose you want to store numbers between -1000 and 1000 using a 32-bit signed integer. Normally, a 32-bit integer can represent values from -2147483648 to 2147483647, which is much more than you need. To get more precision, you can use a scaling factor: For example, multiply every real number by 10^6 and store the result as an integer. So, the number 1.234567 becomes 1234567 in storage.

Precision, with a scaling factor of 10^{-6}, is equal to the smallest difference you can represent is 0.000001. The maximum rounding error is half a step, or $0.5 \times 10^{-6} = 0.0000005$.

Examples & Excercises

PSEUDO-ACCURACY in general is a unjustifiably high level of detail, creating a misleading, and artificial sense of accuracy.

LET'S TAKE A CLOSER LOOK at loss of significance, also called catastrophic cancellation.

This occurs when subtracting two nearly equal numbers, causing leading digits to cancel and leaving only the less significant, rounding-error-prone digits. This can greatly amplify rounding errors. Let's say we have:

SUPPOSE WE HAVE TWO NUMBERS a and b that are both stored in a computer with limited precision. Let's say each is rounded to 8 significant digits as a fixed-point representation:

$$a = 12345678.5$$
$$b = 12345678.0$$

But with 8 significant digits, they are stored as:

$$\tilde{a} = 12345679$$
$$\tilde{b} = 12345678$$

Now, subtract:

$$\tilde{a} - \tilde{b} = 12345679 - 12345678 = 1$$

Compare to the true difference:

$$a - b = 12345678.5 - 12345678.0 = 0.5$$

THE ERROR IN THE RESULT is 0.5, which is equal to the rounding error in $\tilde{a}$ or $\tilde{b}$ individually at the machine epsilon level 0.5. Let's have a look at what happens with a minimal deviation in the numbers.

$$a = 12345678.4$$
$$b = 12345678.0$$

But with 8 significant digits, they are stored as:

$$\tilde{a} = 12345678$$
$$\tilde{b} = 12345678$$

Now, subtract:

$$\tilde{a} - \tilde{b} = 12345678 - 12345678 = 0$$

Compare to the true difference:

$$a - b = 12345678.4 - 12345678.0 = 0.4$$

INFINITY in mathematics there is an infinity between 0 and 1. The difference between something and nothing.

THE DIFFERENCE IN THE TRUE RESULTS is 0.1, but comparing the machine results, we see that the result is 0 in one case and 1 in the other, which is a huge relative error. This shows how loss of significance can lead to large errors in computations, especially when the numbers being subtracted are very close to each other.

HANDS ON MACHINE EPSILON. But just before you head over to your machine, think about how you would determine the machine epsilon of any given system,

for a specific floating-point format, by deploying a program. Revisit how machine epsilon is defined. Write down your thoughts in pseudo-code. What would such a program show when run on a fixed-point system vs a floating-point system? Write down your thoughts. Then try different types on your machine in code. Write down things that you observe and reflect on them.

When you would build a calculator, which type would you choose and why [3]?

CODE is again to be found here https://github.com/Quillstacks/lecturecode_numericalmethods.git.

OVERFLOW, occurs when numbers with large magnitude are approximated as $+\infty$ or $-\infty$.

UNDERFLOW, occurs when numbers near zero are rounded to zero.

IS DOUBLE ENOUGH?

[3]

LOSS OF SIGNIFICANCE EXAMPLE. Now, think about how you would demonstrate loss of significance on your machine. Write down your thoughts in pseudo-code, before you read beyond this point.

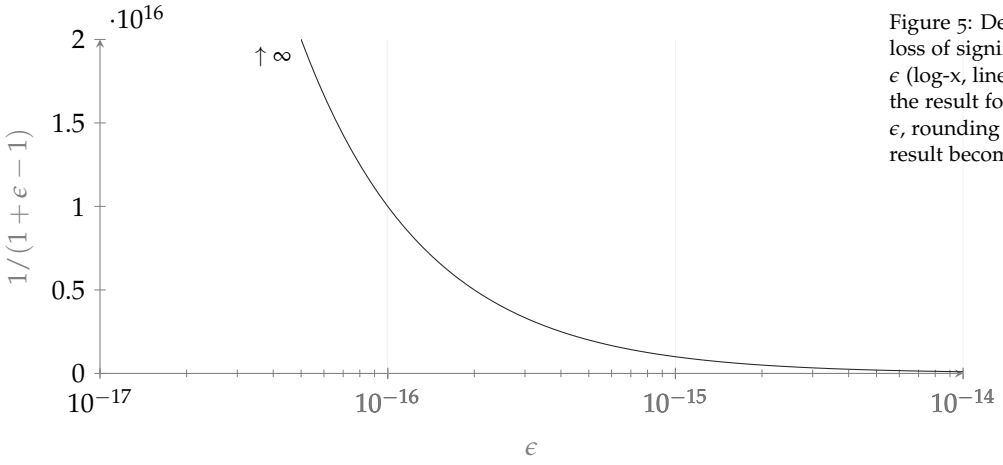

Figure 5: Demonstration of catastrophic loss of significance: $1/(1+\epsilon-1)$ vs ϵ (log-x, linear-y scale). For large ϵ, the result follows $1/\epsilon$. For very small ϵ, rounding error dominates and the result becomes infinite.

REASON ABOUT how computing $f(\epsilon) = 1/(a+\epsilon-b)$ for small ϵ and $a = b$ would do the job. What do you expect to see when ϵ is very small?

WHAT HAPPENS for $a > b$?

Think along the lines of significant digits.

Self-Reflection and Recap

Self-Reflection Questions which can guide your thoughts during the excercises and afterwards:

- How is floating-point representation structured, and what are its components?
- What is machine epsilon, and how does it relate to numerical precision?
- How do these concepts impact numerical computations in practice?

Recap of Key Concepts:

- Floating-point representation allows computers to store a wide range of real numbers using a finite number of bits, but introduces rounding errors.
- Machine epsilon quantifies the smallest difference that can be represented in floating-point arithmetic, affecting the precision of numerical computations.
- Loss of significance occurs when subtracting nearly equal numbers, amplifying rounding errors and leading to inaccurate results.

Knowing what can go wrong. We are now close to understanding how we can define what is good and the quality of our methods. We now know that there is a true function f and an approximated function $\hat{f}$, further we have a true input x and a rounded input $\tilde{x}$. These effect and characterize our numerical methods.

Teaser. Can you think of metrics for numerical methods, based on the approximations and errors we discussed?