

Lec 2: Numerics and Error Analysis

15-369/669/769: Numerical Computing

Instructor: Minchen Li

Table of Content

- Storing Numbers with Fractional Parts
- Understanding Error

Table of Content

- Storing Numbers with Fractional Parts
- Understanding Error

Storing Numbers with Fractional Parts

Fixed-Point Representation

- Most computers store data in binary format, e.g. we can convert 463 to binary:

1	1	1	0	0	1	1	1	1
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$$\begin{aligned} 463 &= 256 + 128 + 64 + 8 + 4 + 2 + 1 \\ &= 2^8 + 2^7 + 2^6 + 2^3 + 2^2 + 2^1 + 2^0 \end{aligned}$$

- Negative numbers can be represented by
 - introducing a leading sign bit, or
 - using a “two’s complement” trick.
- The binary system admits an extension to real numbers by including negative powers of two, e.g., 463.25 can be decomposed by adding two slots:

1	1	1	0	0	1	1	1	1.	0	1
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}

Storing Numbers with Fractional Parts

Fixed-Point Representation (An Example of Approximation)

- Writing the fraction $1/3$ in binary requires infinitely many digits: $\frac{1}{3} = 0.0101010101 \dots_2$
- All irrational numbers, e.g. π , have infinitely long expansions regardless of which (integer) base we use.
- Thus, we have to approximate, which leads to many points of confusion while coding:

```
double x = 1.0;
double y = x / 3.0;
if (x == y*3.0) cout << "They are equal!";
else cout << "They are NOT equal.";
```

```
double x = 1.0;
double y = x / 3.0;
if (fabs(x-y*3.0) < numeric_limits<double>::epsilon)
    cout << "They are equal!";
else cout << "They are NOT equal.";
```

Rarely if ever should the operator `==` and its equivalents be used on fractional values. Instead, some `tolerance` should be used to check if they are equal. Needs to be adjusted depending on context!

Storing Numbers with Fractional Parts

Fixed-Point Representation (Pros and Cons)

- Primary advantage: many efficient integer arithmetic operations can be reused, e.g. summation.
- Serious precision issues:
 - Suppose we include *one decimal point of precision*,

$$1/2 \cdot 1/2 = 1/4 \quad \text{calculated as} \quad 0.1_2 \times 0.1_2 = 0.01_2 \quad \text{represented as 0 after truncation!}$$

- Most programming languages do not include a fixed-point data type.
- Some lower-end GPUs implement only fixed-point operations and are still found in embedded or real-time systems with strict performance and power constraints.

Storing Numbers with Fractional Parts

Floating-Point Representations

- Motivation: extreme range of scales, e.g. in physics-based simulation:
 - Young's modulus of metal: $\sim 10^{10} Pa$, thickness of a shell: $\sim 10^{-4} m$
- Inspiration: scientific notation, e.g. $\underline{6.022} \times 10^{23} = 602, 200, 000, 000, 000, 000, 000, 000$
4 digits of precision
 - Avoids writing a lot of zeros, indicates precision
- Floating-point numbers: $a \times 10^e$ for some $a \sim 1$ and $e \in \mathbb{Z}$
 - a : significant
 - e : exponent

Storing Numbers with Fractional Parts

Floating-Point Representations (Parameters and Expansion)

Floating-point systems are defined using three parameters:

- The *base* or *radix* $b \in \mathbb{N}$. For scientific notation explained above, the base is $b = 10$; for binary systems the base is $b = 2$.
- The *precision* $p \in \mathbb{N}$ representing the number of digits used to store the significand.
- The range of exponents $[L, U]$ representing the allowable values for e .

The expansion looks like:

$$\underbrace{\pm}_{\text{sign}} \underbrace{(d_0 + d_1 \cdot b^{-1} + d_2 \cdot b^{-2} + \dots + d_{p-1} \cdot b^{1-p})}_{\text{significand}} \times \underbrace{b^e}_{\text{exponent}},$$

where each digit d_k is in the range $[0, b - 1]$ and $e \in [L, U]$.

**Can assume $d_0 = 1$ to save storage, but requires special treatment to represent 0.*

Storing Numbers with Fractional Parts

Floating-Point Representations (Machine Precision and Number Distribution)

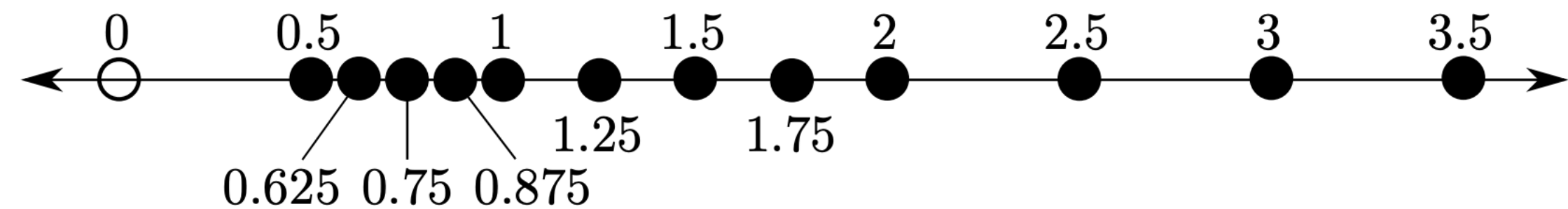
- Machine precision ϵ_m : the smallest $\epsilon_m > 0$ such that $1 + \epsilon_m$ is representable. (Numbers like $b + \epsilon_m$ are not expressible because ϵ_m is too small.)

Example 2.1 (Floating-point). Suppose we choose $b = 2$, $L = -1$, and $U = 1$. If we choose to use three digits of precision, we might choose to write numbers in the form

$$1.\square\square \times 2^\square.$$

Notice this number system does not include 0. The possible significands are $1.00_2 = 1_{10}$, $1.01_2 = 1.25_{10}$, $1.10_2 = 1.5_{10}$, and $1.11_2 = 1.75_{10}$. Since $L = -1$ and $U = 1$, these significands can be scaled by $2^{-1} = 0.5_{10}$, $2^0 = 1_{10}$, and $2^1 = 2_{10}$. With this information in hand, we can list all the possible values in our number system:

Significand	$\times 2^{-1}$	$\times 2^0$	$\times 2^1$
1.00_{10}	0.500_{10}	1.000_{10}	2.000_{10}
1.25_{10}	0.625_{10}	1.250_{10}	2.500_{10}
1.50_{10}	0.750_{10}	1.500_{10}	3.000_{10}
1.75_{10}	0.875_{10}	1.750_{10}	3.500_{10}



Here, $\epsilon_m = 0.25$.

Storing Numbers with Fractional Parts

IEEE 754 Floating-Point Standard (most widely used)

- Specifies several classes, e.g., double-precision in base $b=2$:
 - 1 sign bit
 - 52 bits for significand (fraction)
 - Exponent range: -1022 to 1023
- Supports special values:
 - $\pm\infty$, NaN (“not-a-number”)
 - Used for undefined results (e.g., $1/0$)
- Rounding Convention
 - Common default: round to nearest, ties to even
 - Breaks ties by choosing the value with an even least-significant bit
- Ensures consistent behavior across platforms
- Standardization enables reproducible results in scientific computing

Storing Numbers with Fractional Parts

Other Representations

- Rational numbers

- Motivation: rounding errors are sometimes unacceptable, e.g. hard to distinguish between nearly and completely parallel lines.

- Pros: basic arithmetic without any loss in precision, e.g. $\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$ $\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$

- Cons: the representation is not unique; may require many digits, e.g.

- Tracking the error

$$\frac{1}{100} + \frac{1}{101} + \frac{1}{102} + \frac{1}{103} + \frac{1}{104} + \frac{1}{105} = \frac{188463347}{3218688200}$$

$a, \varepsilon \in \mathbb{R}$; we think of the pair (a, ε) as the range $a \pm \varepsilon$

$$(x \pm \varepsilon_1) + (y \pm \varepsilon_2) = (x + y) \pm (\varepsilon_1 + \varepsilon_2 + \text{error}(x + y))$$

— Maintaining error bars keeps track of confidence in a given value.

Table of Content

- Storing Numbers with Fractional Parts
- Understanding Error

Understanding Error

Sources of Error (Rounding and Discretization)

- Rounding or truncation error
 - can only use a finite number of digits to represent values
 - e.g. impossible to write π exactly as an IEEE 754 floating-point value
- Discretization error
 - comes from computerized adaptations of continuous mathematics
 - e.g. $f'(t) \approx \frac{f(t + \epsilon) - f(t)}{\epsilon}$ is only accurate to some number of digits because of a finite $\epsilon > 0$

Understanding Error

Sources of Error (Modeling and Input)

- Modeling error
 - comes from incomplete or inaccurate descriptions of the problems we wish to solve
 - e.g. a simulation predicting motion of a heavy ball may choose to neglect air damping
- Input error
 - can come from user-generated approximations of parameters of a given system

Understanding Error

Sources of Error (Example)

Example 2.2 (Computational physics). Suppose we are designing a system for simulating planets as they revolve around the sun. The system essentially solves Newton's equation $F = ma$ by integrating forces forward in time. Examples of error sources in this system might include:

- *Rounding error*: Rounding the product ma to IEEE floating-point precision
- *Discretization error*: Using divided differences as above to approximate the velocity and acceleration of each planet
- *Modeling error*: Neglecting to simulate the moon's effects on the earth's motion within the planetary system
- *Input error*: Evaluating the cost of sending garbage into space rather than risking a Wall-E style accumulation on Earth, but only guessing the total amount of garbage to jettison monthly

Understanding Error

Absolute and Relative Error

Given our previous discussion, the following two numbers might be regarded as having the same amount of error:

$$1 \pm 0.01$$
$$10^5 \pm 0.01.$$

Both intervals $[1 - 0.01, 1 + 0.01]$ and $[10^5 - 0.01, 10^5 + 0.01]$ have the same width, but the latter appears to encode a more confident measurement because the error 0.01 is much smaller *relative* to 10^5 than to 1.

Definition 2.1 (Absolute error). The *absolute error* of a measurement is the difference between the approximate value and its underlying true value.

Definition 2.2 (Relative error). The *relative error* of a measurement is the absolute error divided by the true value.

e.g.: Absolute: 2 in ± 0.02 in
Relative: 2 in $\pm 1\%$

Understanding Error

Catastrophic Cancellation

Example 2.4 (Catastrophic cancellation). Suppose we wish to compute the difference $d \equiv 1 - 0.99 = 0.01$. Thanks to an inaccurate representation, we may only know these two values up to ± 0.004 . Assuming that we can carry out the subtraction step without error, we are left with the following expression for absolute error:

$$d = 0.01 \pm 0.008.$$

In other words, we know d is somewhere in the range $[0.002, 0.018]$. From an absolute perspective, this error may be fairly small. Suppose we attempt to calculate relative error:

$$\frac{|0.002 - 0.01|}{0.01} = \frac{|0.018 - 0.01|}{0.01} = 80\%.$$

Thus, although 1 and 0.99 are known with relatively small error, the difference has enormous relative error of 80%. This phenomenon, known as *catastrophic cancellation*, is a danger associated with subtracting two nearby values, yielding a result close to zero.

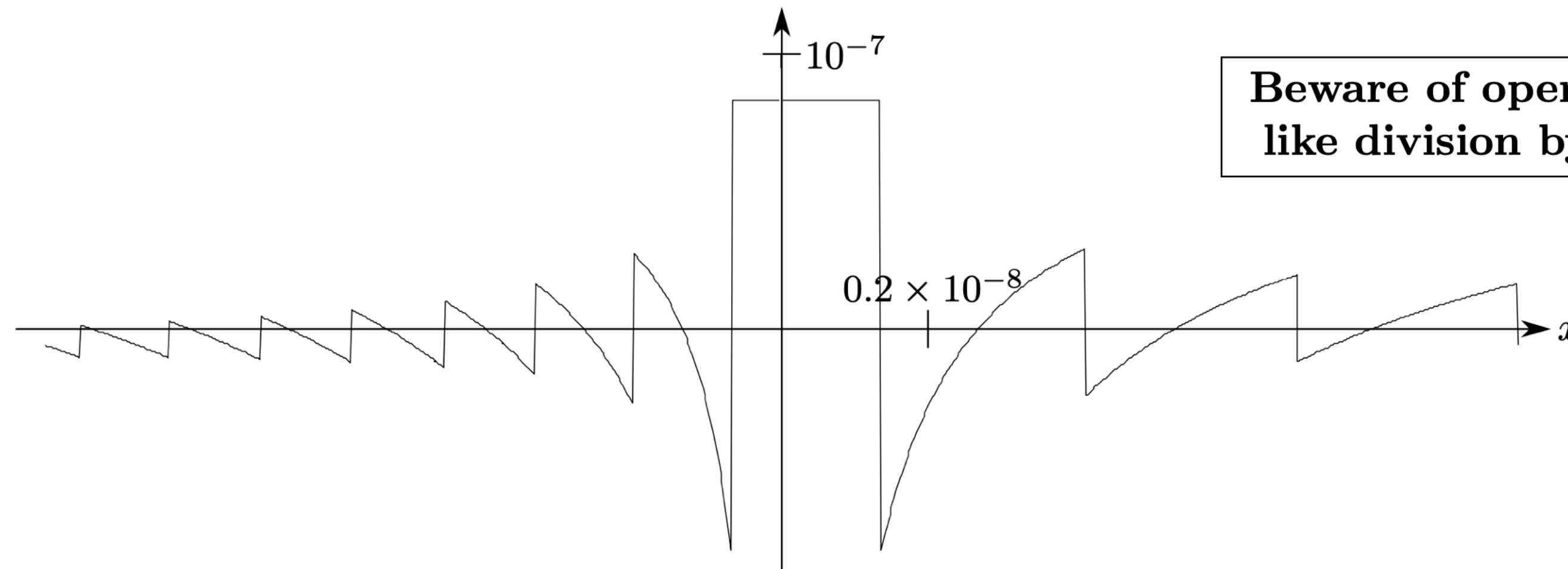
Understanding Error

Catastrophic Cancellation (A Practical Example)

Example 2.5 (Loss of precision in practice). Figure 2.2 plots the function

$$f(x) \equiv \frac{e^x - 1}{x} - 1,$$

for evenly spaced inputs $x \in [-10^{-8}, 10^{-8}]$, computed using IEEE floating-point arithmetic. The numerator and denominator approach 0 at approximately the same rate, resulting in loss of precision and vertical jumps up and down near $x = 0$. As $x \rightarrow 0$, in theory $f(x) \rightarrow 0$, and hence the relative error of these approximate values blows up.



Beware of operations that transition between orders of magnitude, like division by small values and subtraction of similar quantities.

Understanding Error

Forward and Backward Error

- Suppose we wish to solve the equation $f(x) = 0$ for x given a function $f : \mathbb{R} \rightarrow \mathbb{R}$.
- Our computational system may yield some x_{est} satisfying $f(x_{est}) = \varepsilon$ for some ε with $|\varepsilon| \ll 1$.
- If x_0 is the true root satisfying $f(x_0) = 0$, we may not be able to evaluate $|x_0 - x_{est}|$ since x_0 is unknown.
Backward error**Forward error**
- But by evaluating f we can compute $|f(x_{est}) - f(x_0)| \equiv |f(x_{est})|$ since $f(x_0) = 0$. This difference of f values gives a proxy for error that still is zero exactly when $x_{est} = x_0$.

Definition 2.3 (Backward error). The *backward error* of an approximate solution to a numerical problem is the amount by which the problem statement would have to change to make the approximate solution exact.

Understanding Error

Conditioning

- In nearly any numerical problem, zero backward error \Leftrightarrow zero forward error.
- In practice, we terminate our numerical solver when backward error is sufficiently small.
- But does small backward error \Rightarrow small forward error? **It depends!**
- A problem is *insensitive* or *well-conditioned* when small amounts of backward error imply small amounts of forward error. In other words, a small perturbation to the statement of a well-conditioned problem yields only a small perturbation of the true solution.
- A problem is *sensitive*, *poorly conditioned*, or *stiff* when this is not the case.

Understanding Error

Condition Number

Example 2.8 ($ax = b$). Suppose as a toy example that we want to find the solution $x_0 \equiv b/a$ to the linear equation $ax = b$ for $a, x, b \in \mathbb{R}$. Forward error of a potential solution x is given by $|x - x_0|$ while backward error is given by $|b - ax| = |a(x - x_0)|$. So, when $|a| \gg 1$, the problem is well-conditioned since small values of backward error $a(x - x_0)$ imply even smaller values of $x - x_0$; contrastingly, when $|a| \ll 1$ the problem is ill-conditioned, since even if $a(x - x_0)$ is small, the forward error $x - x_0 \equiv 1/a \cdot a(x - x_0)$ may be large given the $1/a$ factor.

Definition 2.4 (Condition number). The *condition number* of a problem is the ratio of how much its solution changes to the amount its statement changes under small perturbations. Alternatively, it is the ratio of forward to backward error for small changes in the problem statement.

Usually as hard as computing $x_0 \dots$

Example 2.9 ($ax = b$, continued). Continuing Example 2.8, we can compute the condition number exactly:

$$c = \frac{\text{forward error}}{\text{backward error}} = \frac{|x - x_0|}{|a(x - x_0)|} \equiv \frac{1}{|a|}.$$

Small c: well-conditioned;

Large c: bad-conditioned.

Understanding Error

An Example of Approximating Condition Number

Example 2.10 (Root-finding). Suppose that we are given a smooth function $f : \mathbb{R} \rightarrow \mathbb{R}$ and want to find roots x with $f(x) = 0$. By Taylor's theorem, $f(x + \varepsilon) \approx f(x) + \varepsilon f'(x)$ when $|\varepsilon|$ is small. Thus, an approximation of the condition number for finding the root x is given by

$$\frac{\text{forward error}}{\text{backward error}} = \frac{|(x + \varepsilon) - x|}{|f(x + \varepsilon) - f(x)|} \approx \frac{|\varepsilon|}{|\varepsilon f'(x)|} = \frac{1}{|f'(x)|}.$$

This approximation generalizes the one in Example 2.9. If we do not know x , we cannot evaluate $f'(x)$, but if we can examine the form of f and *bound* $|f'|$ near x , we have an idea of the worst-case situation.

Table of Content

- Storing Numbers with Fractional Parts
- Understanding Error