Floating Point Arithmetic

Atonu Roy Chowdhury,
ext.atonu.roy@bracu.ac.bd

February 4, 2026

We use computers all the time to do computations: checking your bank balance, rendering 3D graphics, training ML models, simulating physics, running navigation on a phone, or even computing grades. All of these tasks involve numbers, but a computer does not "see" numbers the way humans do. A CPU ultimately only stores and manipulates **bits** (`0/1`) and performs operations on fixed-size bit patterns.

So, to work with numbers in programs, we must choose a **representation**: a rule that maps bit patterns to numeric values. For example:

- a 32-bit signed integer maps 2^{32} different bit patterns to whole numbers in a fixed range;
- a floating-point number maps bit patterns to values that look like scientific notation (good range, limited precision).

You can think of many numbers (like π or $\sqrt{2}$) as mathematical ideas that exist in our minds. To make a computer *understand* and operate on those ideas, we encode them into finite memory. Because memory is finite, this encoding always has trade-offs:

- **range**: how big/small a magnitude we can represent,
- **precision**: how many digits are trustworthy,
- **speed**: how fast arithmetic is,
- **simplicity**: how easy it is to reason about correctness.

This is why numerical computing starts with representation: before worrying about algorithms, we must know what numbers the machine can actually store.

1 Number representation

When we say a number like 7, 0.1, or π, we are talking about a *mathematical idea*. Humans can reason about that idea directly. A computer cannot — it only stores **bit patterns**. So we must decide a rule that tells the machine:

Idea in our head	**Bits in memory**	**Meaning (rule)**
"the number 5"	`0101`	int: exact
"a probability"	`0x3fb999...`	float: approx

In other words: **representation** is the bridge between "numbers in our mind" and "bits in a machine".

A computer has *finite memory*. So for any numeric type, there are only finitely many bit patterns, meaning only finitely many values can be stored exactly. The goal is to pick a representation that is fast, predictable, and good enough for the task.

In practice, we usually choose between:

- Exactness in a limited range (integers), vs.
- Huge range with limited precision (floating point).

Integers can be represented exactly, up to some maximum size. In most programming languages, we use 32-bit or 64-bit system to store integers. If we use a N-bit system, the **first** bit is used to denote **sign** of the number. The rest of the bits signify the number's **magnitude** in binary.

Example 1.1

Think of a 64-bit signed integer as 64 "boxes" that can each store one bit (0 or 1).

- Bit 1 (the first box) is the sign bit: we will use `0` to mean "+" and `1` to mean "−".
- Bits 2–64 are the data bits: they form the number in **binary**.

So the stored value is: `sign bit`, `63-bit binary number`.

To get the **maximum positive** value, the sign bit must indicate "+" and then all the remaining 63 boxes must be `1`. That gives

$$1 \cdot 2^{62} + 1 \cdot 2^{61} + \cdots + 1 \cdot 2^1 + 1 \cdot 2^0 = 2^{63} - 1.$$

(Remember the geometric series formula $1 + r + r^2 + \cdots + r^n = \frac{r^{n+1}-1}{r-1}$.) Therefore, the maximum positive integer that can be stored in a 64-bit system is $2^{63} - 1$. Similarly, the maximum positive integer can be stored in an N-bit system is $2^{N-1} - 1$.

So we can't store integers beyond $2^{63} - 1$. But the addition/product of two numbers can exceed this limit. If such a case occurs, we call it **overflow**.

Intuition 1.1

This is why fixed-width integer types can overflow, but the representation is exact as long as you stay in range. Meaning, as long as we are in the range of $-2^{N-1} - 1$ to $+2^{N-1} - 1$, we can store every integer accurately and exactly. Overflow occurs when some numerical value goes out of this range.

Anecdote 1.1

In Python this is usually not a worry: even though a "normal" 32-bit integer has maximum size $2^{31} - 1$, larger results are automatically promoted to arbitrary-precision integers.

2 Floating points

In contrast to integers, only a *subset* of real numbers within any given interval can be represented exactly. In everyday life, we tend to use a **fixed-point** representation

$$x = \pm(d_1 d_2 \cdots d_{k-1}.d_k \cdots d_n)_\beta, \qquad d_1, \ldots, d_n \in \{0, 1, \ldots, \beta - 1\}. \tag{2.1}$$

Here β is the base (e.g. 10 for decimal arithmetic or 2 for binary).

Intuition 2.1: What "fixed" means

In fixed-point, the decimal point (more generally: the radix point) is in a *fixed location*. So the meaning of each digit position never changes. This is why fixed-point is easy to reason about: addition/subtraction behave like normal school arithmetic.

Example 2.1

$(10.1)_2 = 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} = 2.5$.

Anecdote 2.1: Why we sometimes prefer fixed-point

If you know your values stay within a limited range and you care about exactness (e.g., money, counters, IDs), fixed-point (or integers with a scale) avoids many floating-point surprises.

If we require that $d_1 \neq 0$ unless $k = 2$, then every number has a unique representation of this form, except for infinite trailing sequences of digits $\beta - 1$.

Example 2.2: Non-unique decimals

$0.999999\cdots = 1.000000\cdots$, $3.19999\cdots = 3.2$.

Intuition 2.2

Fixed-point is like "store an integer plus an implied scale factor". For example, storing money as cents (integer) is fixed-point with scale 100. It avoids many floating-point surprises for accounting-style computations.

Computers use a **floating-point** representation. Only numbers in a floating-point number system $F \subset \mathbb{R}$ can be represented exactly, where

$$F = \left\{ \pm (0.d_1 d_2 \cdots d_m)_\beta \, \beta^e \mid \beta, d_i, e \in \mathbb{Z},\ 0 \leq d_i \leq \beta - 1,\ e_{\min} \leq e \leq e_{\max} \right\}. \tag{2.2}$$

Here $(0.d_1 d_2 \cdots d_m)_\beta$ is called the **fraction** (also: significand or mantissa), β is the base, and e is the exponent.

Intuition 2.3: Why "floating"?

The radix point is not fixed. The fraction gives you a fixed amount of precision (a fixed number of digits/bits), while the exponent *moves* the radix point left/right. That is why floating point can represent very small and very large numbers with the same number of bits.

This can represent a much larger range of numbers than a fixed-point system of the same size, although at the cost that the numbers are not equally spaced. If $d_1 \neq 0$ then each number in F has a unique representation and F is called **normalized**.

Intuition 2.4

This is like scientific notation, but in base 2: you keep a fixed number of bits of "precision" (the fraction), and you slide the radix point around using the exponent.

Intuition 2.5: Practical consequence

Because spacing is not uniform, the same absolute error can be "small" for large values and "huge" for tiny values. This is the root reason why rounding, overflow/underflow, and comparison issues happen later in the chapter.

Example 2.3: Toy floating-point system

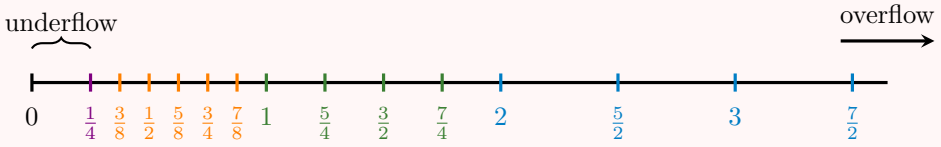
Consider a floating-point number system with $\beta = 2$, $m = 3$, $e_{\min} = -1$, $e_{\max} = 2$.

Important observation: The spacing between numbers jumps by a factor β at each power of β. The largest possible number is

$$(0.111)_2 \, 2^2 = \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} \right) \cdot 4 = \frac{7}{2}.$$

The smallest non-zero positive number is

$$(0.100)_2 \, 2^{-1} = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}.$$



So, in this toy system:

- anything with magnitude smaller than $\frac{1}{4}$ is treated as 0 (underflow),
- anything with magnitude larger than $\frac{7}{2}$ is treated as ∞ (overflow).

Anecdote 2.2

Suppose you have a ruler that is marked in millimeters and you cannot reliably measure anything smaller than 1mm. If an object has length 0.2mm, your measurement device will simply report 0mm, not because the object has zero length, but because your tool does not have enough *resolution* to distinguish such small values.
Underflow in floating-point arithmetic is the same idea. When numbers become smaller than the smallest non-zero value your system can represent, the computer loses the ability to distinguish them, and they collapse to 0.
Floating point behaves similarly: below a certain threshold, the machine cannot represent smaller non-zero magnitudes. So extremely small results may underflow and become 0. Underflow-to-zero is often fine (e.g., probabilities that are effectively 0), but it can also break algorithms if you assume every non-zero stays non-zero.

Near bigger numbers you have bigger "gaps" between representable values. That is why adding a small number to a huge number can do nothing.

Example 2.4: IEEE double precision (64-bit)

IEEE standard (1985) for double-precision (64-bit) arithmetic. Here $\beta = 2$, and there are 52 bits for the fraction, 11 for the exponent, and 1 for the sign. The actual format used is

$$\pm(1.d_1 \cdots d_{52})_2 \, 2^{e-1023} = \pm(0.1d_1 \cdots d_{52})_2 \, 2^{e-1022}, \qquad e = (e_1 e_2 \cdots e_{11})_2.$$

When $\beta = 2$, the first digit of a normalized number is always 1, so it does not need to be stored in memory.
The exponent bias avoids the need to store the sign of the exponent.
The exponent bias means that the actual exponents are in the range -1022 to 1025 (since $e \in [0, 2047]$). In practice, the exponents -1022 and 1025 are used to store ± 0 and $\pm\infty$ respectively.
The smallest non-zero number in this system is

$$(0.1)_2 \, 2^{-1021} \approx 2.225 \times 10^{-308},$$

and the largest number is

$$(0.1 \cdots 1)_2 \, 2^{1024} \approx 1.798 \times 10^{308}.$$

IEEE stands for Institute of Electrical and Electronics Engineers. This is the default system in Python/NumPy and in most programming languages. The automatic leading 1 is often called the **hidden bit**.

3 Overflow, underflow, and subnormals

Numbers outside the finite set F cannot be represented exactly. If a calculation falls below the lower non-zero limit (in absolute value), it is called **underflow**, and is usually set to 0. If it falls above the upper limit, it is called **overflow**, and usually results in a floating-point exception (or becomes `Inf`, depending on language/settings).

Example 3.1

In Python, overflow occurs when you try to compute 2^{1024} (in float), and underflow occurs when you try to compute 2^{-1075}.

```
[atonu@Atonus-MacBook-Air ~ % python3                                    ]
 Python 3.13.5 (v3.13.5:6cb20a219a8, Jun 11 2025, 12:23:45) [Clang 16.0.0 (clang-
 1600.0.26.6)] on darwin
 Type "help", "copyright", "credits" or "licensе" for more information.
[>>> 2.0 ** 1023                                                         ]
 8.98846567431158e+307
[>>> 2.0 ** 1024                                                         ]
 Traceback (most recent call last):
   File "<python-input-1>", line 1, in <module>
     2.0 ** 1024
     ~~~~^^~~~~
 OverflowError: (34, 'Result too large')
[>>> 2.0 ** -1074                                                        ]
 5e-324
[>>> 2.0 ** -1075                                                        ]
 0.0
 >>>
```

Anecdote 3.1

In 1996, the maiden flight of the Ariane 5 rocket ended in failure. Roughly 40 seconds after launch, at an altitude of about 3700m, the rocket veered off its intended trajectory and self-destructed.
A key cause was a software exception that occurred while converting a value stored as a 64-bit floating-point number into a 16-bit signed integer. The true value was too large to fit into the smaller integer type, so the conversion overflowed and triggered an exception. Because the flight-control software did not handle that exception safely in that situation, the rocket lost valid guidance data.
This is a classic engineering lesson for numerical computing: the representation is not just a theory topic. A wrong assumption about what a numeric type can represent (especially during type conversion) can crash real systems.

Subnormal numbers (idea). In IEEE arithmetic, some numbers in the "zero gap" can be represented using $e = 0$, since only two possible fraction values are needed for ± 0. The other fraction values may be used with first (hidden) bit 0 to store a set of so-called **subnormal** numbers.

Intuition 3.1

Subnormals extend the range closer to 0, but with less precision. They help avoid sudden "drop to zero" in some computations.

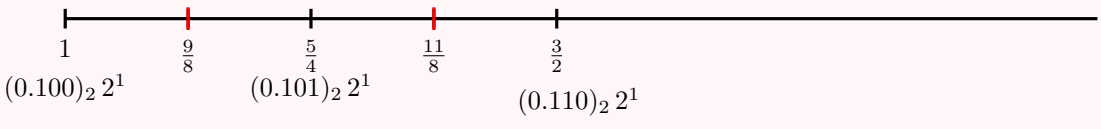
4 Rounding and tie-breaking

The mapping from $\mathbb{R}$ to F is called **rounding** and denoted $\mathrm{fl}(x)$. Usually it is simply the nearest number in F to x. If x lies exactly midway between two numbers in F, a method of breaking ties is required. The IEEE standard specifies **round to nearest even** (take the neighbour with last digit 0 in the fraction). This avoids statistical bias or prolonged drift.

Example 4.1: Rounding in the toy system

$\frac{9}{8} = (1.001)_2$ has neighbours $1 = (0.100)_2\, 2^1$ and $\frac{5}{4} = (0.101)_2\, 2^1$, so it is rounded down to 1.

$\frac{11}{8} = (1.011)_2$ has neighbours $\frac{5}{4} = (0.101)_2\, 2^1$ and $\frac{3}{2} = (0.110)_2\, 2^1$, so it is rounded up to $\frac{3}{2}$.



Anecdote 4.1

Vancouver Stock Exchange index (1982–1983). The Vancouver Stock Exchange introduced its composite index with an initial value of 1000. However, by November 1983 the published index value had drifted down to about 520, even though market activity did not justify such a collapse.
The reason was not an economic shock; it was a numerical design mistake. At each update, the index calculation was rounded down to three decimal digits and then that rounded value was used as the input to the next update. Because the rounding was systematically downward, the small truncation errors accumulated in the same direction over thousands of recomputations. Over time, those tiny one-step errors compounded into a very large bias.
When the exchange later recomputed the index using higher precision (or by avoiding repeated biased rounding during intermediate steps), the corrected index value was roughly double the published value. This is a standard cautionary tale in floating-point computation: small rounding choices that look harmless in isolation can create large, persistent errors when applied repeatedly.

Example 4.2

In pretty much all programming languages, `0.1+0.2` is not exactly equal to `0.3`.

```
atonu@Atonus-MacBook-Air ~ % python3
Python 3.13.5 (v3.13.5:6cb20a219a8, Jun 11 2025, 12:23:45) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 - 0.3
5.551115123125783e-17
>>>
```

Their difference is around 10^{-17}. When you check the binary value (the way numbers are stored), you'll see that `0.1+0.2` and `0.3` are slightly different.

```
>>> a = 0.1 + 0.2
>>> b = 0.3
>>> a.hex()
'0x1.3333333333334p-2'
>>> b.hex()
'0x1.3333333333333p-2'
>>>
```

This is because of rounding error. When you convert 0.1 and 0.2 to binary, you cannot convert them exactly in the 64-bit system, we have to round them to the nearest possible representable. This rounding leads to the error. We'll discuss rounding error soon.

4.1 Significant figures

When doing calculations without a computer, we often use the terminology of **significant figures**. To count the number of significant figures in a number x, start with the first non-zero digit from the left, and count all the digits thereafter, including final zeros if they are after the decimal point.

Example 4.3: Significant figures

3.1056, 31.050, 0.031056, 0.031050, and 3105.0 all have 5 significant figures (s.f.).

To round x to n s.f., replace x by the nearest number with n s.f. An approximation $\hat{x}$ of x is "correct to n s.f." if both $\hat{x}$ and x round to the same number to n s.f.

Intuition 4.1

Significant figures are basically a human-friendly way of talking about precision. Floating point behaves similarly: a `double` has about 15–16 decimal digits of precision, but the exponent allows huge range.

5 Rounding error

Even if a real number x lies within the representable range of our floating-point system (so it is not overflowed to ∞ and not underflowed to 0), it usually *still* cannot be stored exactly. Instead, it is rounded to the nearest representable floating-point number.

If $|x|$ lies between the smallest non-zero number in F and the largest number in F, then rounding can be written as

$$\mathrm{fl}(x) = x(1+\delta), \tag{5.1}$$

where the (dimensionless) **relative rounding error** is

$$|\delta| = \frac{|\mathrm{fl}(x) - x|}{|x|}. \tag{5.2}$$

Intuition 5.1: Why relative error matters

Relative errors are **scale invariant**. An absolute error of 1 hour is irrelevant when estimating the age of a building, but catastrophic if you are timing your arrival to an exam. Relative error captures this idea: it measures error as a *fraction* of the true size.

Bounding the rounding error. Write x in base-β scientific notation

$$x = (0.d_1 d_2 d_3 \cdots)_\beta \, \beta^e,$$

where $e \in [e_{\min}, e_{\max}]$. If $x \notin F$, then its fraction does not terminate after m digits. Rounding to m digits changes the fraction by at most half a unit in the last place:

$$|(0.d_1 \cdots d_m)_\beta - (0.d_1 \cdots d_m(\text{exact}))_\beta| \le \frac{1}{2}\,\beta^{-m}.$$

Multiplying by β^e gives an absolute error bound

$$|\mathrm{fl}(x) - x| \le \frac{1}{2}\,\beta^{-m}\,\beta^e. \tag{5.3}$$

To convert this into a relative error bound, note that for a *normalized* non-zero number, its leading digit satisfies $d_1 \ge 1$, so the fractional part satisfies $(0.d_1 d_2 \cdots)_\beta \ge (0.1)_\beta = \beta^{-1}$. Hence

$$|x| \ge \beta^{-1}\beta^e, \tag{5.4}$$

and dividing (5.3) by $|x|$ yields

$$|\delta| \le \frac{1}{2}\,\beta^{1-m}. \tag{5.5}$$

The constant $\varepsilon_M = \frac{1}{2}\,\beta^{1-m}$ is called the **machine epsilon** (or **unit roundoff**). It depends only on the base β and the number of fraction digits m, not on x. Equation (5.5) becomes the key rounding guarantee:

$$|\delta| \le \varepsilon_M. \tag{5.6}$$

Intuition 5.2: Why "unit roundoff"?

Near 1, adjacent representable floating-point numbers are spaced about β^{1-m} apart. So ε_M is (roughly) *half* the distance from 1 to its nearest neighbour.

Example 5.1: Machine epsilon in the toy system

For our toy system with $\beta = 2$ and $m = 3$,

$$\varepsilon_M = \frac{1}{2}\,2^{1-3} = \frac{1}{8}.$$

Example 5.2: Machine epsilon in IEEE double

For IEEE double precision, $\beta = 2$ and there are $m = 53$ significant bits (including the hidden bit), so

$$\varepsilon_M = \frac{1}{2}\,2^{1-53} = 2^{-53} \approx 1.11 \times 10^{-16}.$$

As you have seen earlier, the error in computing $0.1 + 0.2$ in Python is around 10^{-17}, which is smaller than ε_M, as we know from $|\delta| \le \varepsilon_M$.

5.1 Rounding in arithmetic operations

Even if $x, y \in F$, the exact real-number result of $x \circ y$ (where $\circ \in \{+, -, \times, \div\}$) usually is not in F, so it must be rounded:

$$\mathrm{fl}(x \circ y) \in F.$$

IEEE arithmetic aims for **rounded exact operations**: compute the exact real result, then round once at the end. This is often idealized as

$$\mathrm{fl}(x \circ y) = (x \circ y)(1 + \delta), \qquad |\delta| \le \varepsilon_M. \tag{5.7}$$

Example 5.3: Multiplication in the toy system

Take our toy system again ($\beta = 2$, $m = 3$, $e_{\min} = -1$, $e_{\max} = 2$). Let $x = \frac{5}{8} = (0.101)_2\,2^0$, $y = \frac{7}{8} = (0.111)_2\,2^0$. Then

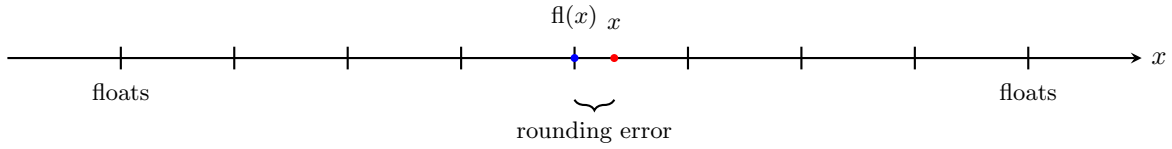
$$xy = \frac{35}{64} = (0.100011)_2\,2^0.$$

This has more than $m = 3$ fraction bits, so we must round:

$$\mathrm{fl}(xy) = (0.100)_2\,2^0 = \frac{1}{2}.$$

Intuition 5.3

Real hardware typically keeps extra "guard digits" internally during the multiplication, then rounds once to the target format (instead of rounding after every intermediate bit operation). That is a big reason why IEEE arithmetic is more accurate than naive digit-chopping.



6 Loss of significance (catastrophic cancellation)

Equation (5.7) can make floating-point arithmetic look "safe" because each operation incurs only a small relative error. But this guarantee is meaningful only when the exact result is not too small compared to the operands.

In practice we rarely start from exact real numbers. Instead we usually have floating-point inputs

$$\bar{x} = x(1 + \delta_1)$$

and

$$\bar{y} = y(1 + \delta_2)$$

with $|\delta_1|, |\delta_2| \le \varepsilon_M$. Then subtraction/addition gives

$$\bar{x} \pm \bar{y} = x(1 + \delta_1) \pm y(1 + \delta_2) = (x \pm y)\left(1 + \frac{x\delta_1 \pm y\delta_2}{x \pm y}\right). \tag{6.1}$$

If $x \pm y$ is very small (because x and y are close), the fraction $\dfrac{x\delta_1 \pm y\delta_2}{x \pm y}$ can become huge. So the *relative* error in the result can be much larger than ε_M. This phenomenon is called **loss of significance** or **catastrophic cancellation**.

Intuition 6.1

When two nearly equal numbers are subtracted, the leading digits cancel. The remaining digits may come mostly from rounding noise rather than true information. It is like measuring two long distances with a ruler that is accurate only to the nearest millimeter, then subtracting them to estimate a tiny gap: the gap inherits almost all of the measurement uncertainty.

Example 6.1

Consider

$$x^2 - 56x + 1 = 0.$$

The exact roots are $x_{1,2} = 28 \pm \sqrt{783}$. To 4 significant figures, $x_1 \approx 55.98$ and $x_2 \approx 0.01786$. But if we round $\sqrt{783}$ to 4 s.f. as 27.98, then

$$\bar{x}_2 = 28 - 27.98 = 0.02000,$$

which is not correct to 4 s.f. The subtraction destroyed the significant digits.

Fix (algebraic reformulation). Since $x^2 - 56x + 1 = (x - x_1)(x - x_2)$, the product of the roots is $x_1 x_2 = 1$. So compute the small root as

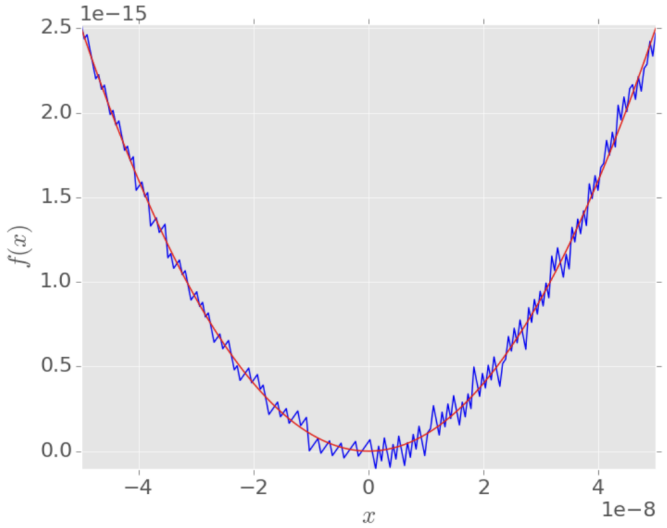
$$x_2 = \frac{1}{x_1},$$

which avoids subtracting nearly equal numbers.

Example 6.2

Let $f(x) = e^x - \cos(x) - x$. For x very close to 0 (near 10^{-8} or so), each term is close to 1 (or 0), and the exact answer is tiny. Direct evaluation can lose significance due to cancellation.

Example → Evaluate $f(x) = e^x - \cos(x) - x$ for x very near zero.
Let us plot this function in the range $-5\times10^{-8} \le x \le 5\times10^{-8}$ – even in IEEE double precision arithmetic we find significant errors, as shown by the blue curve:



A stable alternative uses the Taylor series expansions (as done in red curve)

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots, \qquad \cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots.$$

Subtracting and simplifying gives

$$f(x) \approx x^2 + \frac{x^3}{6}$$

for small $|x|$, which avoids subtracting nearly equal numbers.

Anecdote 6.1: A practical smell test

If a formula produces a small answer by subtracting two large, nearly equal quantities, that is a strong hint that it may be numerically unstable. Numerical analysts often rewrite such formulas to replace "small = big - big" by "small = small" (e.g., via series, factoring, or symmetry).

6.1 Floating point does not behave like real arithmetic

Because every operation rounds, floating-point arithmetic can violate familiar algebraic rules.

Example 6.3: Non-associativity

In 2-digit decimal arithmetic,

$$\mathrm{fl}\big((5.9 + 5.5) + 0.4\big) = \mathrm{fl}\big(\mathrm{fl}(11.4) + 0.4\big) = \mathrm{fl}(11.0 + 0.4) = 11.0,$$

but

$$\mathrm{fl}\big(5.9 + (5.5 + 0.4)\big) = \mathrm{fl}\big(5.9 + 5.9\big) = \mathrm{fl}(11.8) = 12.0.$$

Example 6.4: Averaging can leave the interval

Over the real numbers, the average of two numbers lies between them. But with rounding to 3 decimal digits,

$$\mathrm{fl}\left(\frac{5.01 + 5.02}{2}\right) = \frac{\mathrm{fl}(10.03)}{2} = \frac{10.0}{2} = 5.0,$$

which is not between 5.01 and 5.02.

7 End-of-chapter questions

1. In the toy system with $\beta = 2$, $m = 3$, $e_{\min} = -1$, $e_{\max} = 2$:
   (a) List all *positive normalized* numbers in F.
   (b) What is the spacing between consecutive representable numbers in $[1, 2)$? in $[2, 4)$?
   (c) Explain (in one paragraph) why spacing grows by a factor of β when the exponent increases by 1.
2. Convert each value into normalized base-2 floating-point form $\pm(0.1d_2d_3\cdots)_2 2^e$:
   (a) 13,
   (b) $\frac{5}{16}$,
   (c) 0.1 (explain why it does not terminate in base 2).
3. (a) What "gap" near 0 would exist if only normalized numbers were allowed?
   (b) Why do subnormals reduce sudden underflow-to-zero?
   (c) What is the main trade-off when using subnormals (precision vs range)?
4. IEEE double uses an 11-bit exponent with bias 1023.
   (a) If the stored exponent field is $e = 1023$, what is the true exponent E?
   (b) What is the true exponent when $e = 1$? When $e = 2046$?
   (c) Why are $e = 0$ and $e = 2047$ reserved (what do they represent)?
5. (a) Derive $\varepsilon_M = \frac{1}{2}\beta^{1-m}$ from the idea of rounding to nearest.
   (b) Compute ε_M for the toy system ($\beta = 2$, $m = 3$).
   (c) For IEEE double, explain why we use $m = 53$ significant bits even though only 52 fraction bits are stored.
6. In the toy system ($\beta = 2$, $m = 3$), round each value to the nearest representable number:
   (a) $(0.10111)_2 2^0$,
   (b) $(0.11101)_2 2^1$,
   (c) $(0.10001)_2 2^{-1}$.
7. Assume IEEE-style rounded exact operations: $\mathrm{fl}(x \circ y) = (x \circ y)(1 + \delta),\ |\delta| \le \varepsilon_M$. Explain carefully why this *does not* guarantee that a long computation has relative error $\le \varepsilon_M$.
8. Give two different numerical examples (your own) where subtracting nearly equal numbers causes loss of significance. For each example, propose a more stable reformulation.
9. For $ax^2 + bx + c = 0$ with $b^2 \gg 4ac$:
   (a) Explain why $\frac{-b+\sqrt{b^2-4ac}}{2a}$ can lose significance for one of the roots.
   (b) Show that the product of the roots is $\frac{c}{a}$ and use it to compute the small root stably once the large root is found.
10. Mark each statement true/false and justify briefly.
    (a) Floating-point addition is associative.
    (b) Relative error is always more informative than absolute error.
    (c) Subnormal numbers increase range near 0 but reduce precision.
    (d) Machine epsilon depends on x.