COM701 - Numerical Mathematics and Computing

Chapter 3 - Floating-Point Representation and Errors

Dr. Mahdi Khemakhem

Department of Computer Science
College of Computer Engineering and Science
Prince Sattam bin Abdulaziz University

AY - 2025/2026

Outline

Outline

Case 1: Vancouver Stock Exchange Index (1982-1984)

Impact: Market index drifted from 1000 to 520 (incorrectly).
Cause: Truncation of floating-point values after every update.
Details:

- The Vancouver Stock Exchange started its index at 1000.000.
- Each trade updated the index value, **truncated** (not rounded) to three decimal places.
- Over 22 months, the accumulated truncation error caused the index to fall to 520, even though the market value had not dropped.


INTERNATIONAL REPORT

Mexico Threatened By Oil Price Cuts

International Market Cited

Scottish Financiers Stay Wary Of Pacts

Vancouver Exchange Struggles to Repair Its Tattered Image

Consequence: Reported index dropped to 520 **instead of** $\sim$**1098**. When corrected, investors realized market value was stable.
Lesson: Even minor truncation errors can compound into large distortions.

Case 2: Patriot Anti-Missile Failure (1991, Gulf War)

Impact: 28 soldiers killed, 100+ injured.
Cause: Floating-point rounding error in time calculation.
Details:

- The U.S. Patriot missile defense system used a **floating-point number** to track the time since system startup (in tenths of a second).
- The internal clock **incremented by 0.1 seconds**, but the conversion to real time used a **24-bit fixed-point** register, introducing a **tiny rounding error** which cannot be represented exactly in binary.
- Each tick introduced a **tiny error** ($\approx 9.5 \times 10^{-8}$**s**).
- **After 100 hours, accumulated error** ≈ 0.34**s**.

Consequence: The system miscalculated the incoming Iraqi Al-husein Scud missile's position, failing to intercept it.
Lesson: Even small floating-point errors can accumulate and lead to catastrophic failures.

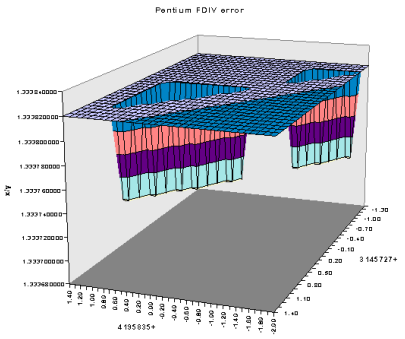
Case 3: Intel Pentium FDIV Bug (1994)

Impact: Hardware floating-point division error — cost $475M.

Cause: Missing entries in lookup table for floating-point division.

Details:

- The Intel Pentium processor had a bug in its **floating-point unit (FPU)** affecting division operations.
- Certain floating-point division operations produced incorrect results due to missing entries in a **lookup table used for the division algorithm**.
- The error was small (**about 1 over 9 billion**) but significant for scientific and financial calculations.



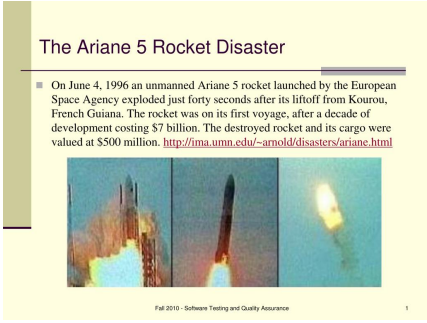
Consequence: Undermined confidence in Intel CPUs and Intel offered full replacements — estimated cost $475M.

Lesson: Even rare floating-point hardware bugs can destroy credibility.

Case 4: Ariane 5 Flight 501 Explosion (1996, ESA)

Impact: Rocket self-destructed after 37 seconds — European Space Agency lost $370M.
Cause: Float-to-integer conversion overflow.
Details:

- The Ariane 5 **reused software from Ariane 4** that converted a **64-bit floating-point number** to a **16-bit signed integer**.
- The variable represented the **horizontal velocity**, which in Ariane 5 **exceeded the maximum representable 16-bit integer**.
- This caused an **overflow error** that **triggered a diagnostic shutdown of both inertial guidance computers**.



Consequence: Rocket veered off course and self-destructed.
Lesson: Reusing code without validating assumptions can be catastrophic.

Key Takeaways from Real-World Cases

Key Takeaways

- **Small errors can accumulate:** Tiny floating-point inaccuracies can grow over time, leading to significant deviations.
- **Validation is crucial:** Always validate assumptions when reusing code or algorithms in new contexts.
- **Understand limitations:** Know the limits of floating-point representation and operations to avoid unexpected behavior.
- **Testing and verification:** Rigorous testing, especially for edge cases, is essential to catch potential floating-point issues early.
- **Impact on safety and finance:** Floating-point errors can have real-world consequences, affecting safety-critical systems and financial calculations.

Outline

Floating-Point Numbers

In scientific computing, we deal with numbers that can be **very large** or **very small**.

Examples of extreme values

- **Avogadro's number:** 6.022×10^{23} (very large)
- **Planck's constant:** 6.626×10^{-34} J·s (very small)
- **Speed of light:** 2.998×10^{8} m/s
- **Mass of electron:** 9.109×10^{-31} kg

$\rightarrow$ *A floating-point number system allows computers to represent and manipulate these numbers efficiently using a fixed amount of storage.*

- In mathematics, we work with the set of **real numbers** $\mathbb{R}$, which is *infinite and continuous*.
- However, computers have **finite memory**, so they can only represent a *finite subset* of real numbers.
- This leads to **representation errors** and **rounding errors**.

Basic Floating-Point Representation

A *floating-point number* in base β (or radix β) is **represented** in the form:

$$x = \pm q \times \beta^m$$

where:

- $\pm$ is the **sign** (positive or negative)
- q is the **mantissa** (or **significand**)
- β is the **base** (usually 2 for binary computers, or 10 for decimal)
- m is the **exponent** (an integer)

$\rightarrow$ *The mantissa q can be expressed as:*

$$q = 0.d_1 d_2 d_3 \ldots d_n \quad \text{where } 0 \leq d_i < \beta$$

where $d_1, d_2, \ldots, d_n$ are the **digits** in base β.

Basic Floating-Point Representation

Example 1: Decimal Floating-Point Number

Express the number -237.425 in floating-point form with base 10.

Solution 1

We can write: $-237.425 = -0.237425 \times 10^3$. Here:

- Sign: $-$ (negative)
- Mantissa: $q = 0.237425$
- Base: $\beta = 10$
- Exponent: $m = 3$

Verification: $-0.237425 \times 10^3 = -0.237425 \times 1000 = \boxed{-237.425}$ ✓

Solution 2

We can also write: $-237.425 = -0.0237425 \times 10^4$.
Here: Sign: $-$ (negative), Mantissa: $q = 0.237425$, Base: $\beta = 10$, Exponent: $m = 4$.

Verification: $-0.0237425 \times 10^4 = -0.0237425 \times 10000 = \boxed{-237.425}$ ✓

Basic Floating-Point Representation

Example 2: Binary Floating-Point Number

Express the number -237.425 in floating-point form with base 2.

Solution

First, convert 237.425 to binary:

- $237_{10} = 11101101_2$ (integer part) $\rightarrow$ *Divide 237 by 2 repeatedly and read the remainders from bottom to top.*
- $0.425_{10} = 0.011\mathbf{0110011001100110}\ldots_2$ (fractional part) $\rightarrow$ *Multiply 0.425 by 2 repeatedly. At each step, keep the integer part, and continue with the fractional part. The pattern '0110' repeats infinitely.*
- Use **32 bits** to exactly match the decimal: $0.425_{10} = 0.\underbrace{01101100110011001100110011001100}_{\text{32 bits of precision}}$

Combine integer and fractional parts: $-237.425 = -11101101.\underbrace{01101100110011001100110011001100}_{\text{32 bits of precision}}$

$$= -0.\underbrace{11101101011011001100110011001100}_{\text{32 bits of precision}} \times 2^8$$

Therefore: **Sign:** $-$ (negative), **Mantissa:** $q = 0.11101101011011001100110011001100_2$, **Base:** $\beta = 2$, **Exponent:** $m = 8$

Basic Floating-Point Representation

Example 2: Binary Floating-Point Number (continued)

Solution (continued)

For verification, convert the mantissa to decimal:

$$0.11101101011011001100110011001100_2 = 1\times 2^{-1}+1\times 2^{-2}+1\times 2^{-3}+0\times 2^{-4}+1\times 2^{-5}+1\times 2^{-6}+0\times 2^{-7}+1\times 2^{-8}+0\times 2^{-9}+1\times 2^{-10}+1\times 2^{-11}+0\times 2^{-12}+1\times 2^{-13}+1\times 2^{-14}+0\times 2^{-15}+0\times 2^{-16}+1\times 2^{-17}+1\times 2^{-18}+0\times 2^{-19}+0\times 2^{-20}+1\times 2^{-21}+1\times 2^{-22}+0\times 2^{-23}+0\times 2^{-24}+1\times 2^{-25}+1\times 2^{-26}+0\times 2^{-27}+0\times 2^{-28}+1\times 2^{-29}+1\times 2^{-30}+0\times 2^{-31}+0\times 2^{-32}$$
$$= 0.5+0.25+0.125+0+0.03125+0.015625+0+0.00390625+0+0.0009765625+0.00048828125+0+0.0001220703125+0.00006103515625+0+0+0.00000762939453125+0.00003814697265625+0+0+0.0000047683715820312+0.0000023841857910156+0+0+0.0000002980232387695+0.0000001490116193848+0+0+0.0000000186264514923 10+0.0000000093132257461548+0+0 = \mathbf{0.9274414060637355}$$

Verification: $-0.9274414060637355\times 256 = -237.42499995 \approx \boxed{-237.425}$ ✓

Normalized Floating-Point Representation

A floating-point number is said to be *normalized* if its **mantissa** satisfies:

$$\frac{1}{\beta} \leq |q| < 1$$

This means the first digit $d_1 \neq 0$.

Why normalize?

- **Uniqueness:** Each number has a *unique representation*.
- **Maximum precision:** Uses all available digits in the mantissa.
- **Avoids ambiguity:** Without normalization, 0.5×10^2 and 0.05×10^3 represent the same number.

Binary normalization

In binary ($\beta = 2$), normalized means $d_1 = 1$, so: $0.5 \leq |q| < 1$.

$\rightarrow$ *The mantissa always starts with* 0.1_2.

Normalized Floating-Point Representation

Example 3: Normalizing Floating-Point Numbers

Determine whether the following floating-point numbers are normalized: 0.752×10^{-3}, 0.0125×10^{5}, $0.101_2 \times 2^3$

Solution

1. 0.752×10^{-3}: **Normalized ✓**
   - Mantissa $q = 0.752$, base $\beta = 10$
   - Check: $\frac{1}{10} = 0.1 \leq 0.752 < 1$ ✓
2. 0.0125×10^{5}: **Not normalized ✗**
   - Mantissa $q = 0.0125$, base $\beta = 10$
   - Check: $0.0125 < 0.1 = \frac{1}{10}$ ✗
   - Normalized form: 0.125×10^{4}
3. $0.101_2 \times 2^3$: **Normalized ✓**
   - Mantissa $q = 0.101_2 = 0.625_{10}$, base $\beta = 2$
   - Check: $\frac{1}{2} = 0.5 \leq 0.625 < 1$ ✓

Normalized Floating-Point Representation

Example 4: Binary Floating-Point Normalization

Express the decimal number 13.25 as a normalized binary floating-point number.

Solution

First, convert 13.25 to binary:

- $13_{10} = 1101_2$ (integer part)
- $0.25_{10} = 0.01_2$ (fractional part)
- Therefore: $13.25_{10} = 1101.01_2$

Now normalize:

$$1101.01_2 = 0.110101_2 \times 2^4$$

Here:

- Sign: $+$ (positive)
- Mantissa: $q = 0.110101_2$ in binary (or 0.828125 in decimal)
- Base: $\beta = 2$
- Exponent: $m = 4$

Note: The mantissa satisfies $\frac{1}{2} = 0.5 \leq 0.828125 < 1$ ✓

Verification: $0.110101_2 \times 2^4 = 0.828125 \times 16 = 13.25$ ✓

Normalized Floating-Point Representation

Example 5: List all floating-point numbers in a simple system

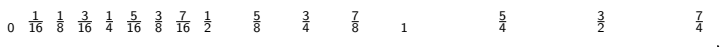
List all the floating-point numbers that can be expressed in the form: $x = \pm(0.b_1b_2b_3)_2 \times 2^{\pm k} \quad (k, b_i \in \{0, 1\})$

Solution

There are two choices for the $\pm$ sign, two for b_1, two for b_2, two for b_3, and three for the exponent (0, 1, -1). This gives $2 \times 2 \times 2 \times 2 \times 3 = 48$ combinations.
However, there is duplication. The nonnegative numbers are (24 combinations):

- $(0.000)_2 \times 2^0 = 0$, $(0.000)_2 \times 2^1 = 0$, $(0.000)_2 \times 2^{-1} = 0$
- $(0.001)_2 \times 2^0 = \frac{1}{8}$, $(0.001)_2 \times 2^1 = \frac{1}{4}$, $(0.001)_2 \times 2^{-1} = \frac{1}{16}$
- $(0.010)_2 \times 2^0 = \frac{2}{8}$, $(0.010)_2 \times 2^1 = \frac{2}{4}$, $(0.010)_2 \times 2^{-1} = \frac{2}{16}$
- $(0.011)_2 \times 2^0 = \frac{3}{8}$, $(0.011)_2 \times 2^1 = \frac{3}{4}$, $(0.011)_2 \times 2^{-1} = \frac{3}{16}$
- $(0.100)_2 \times 2^0 = \frac{4}{8}$, $(0.100)_2 \times 2^1 = 1$, $(0.100)_2 \times 2^{-1} = \frac{4}{16}$
- $(0.101)_2 \times 2^0 = \frac{5}{8}$, $(0.101)_2 \times 2^1 = \frac{5}{4}$, $(0.101)_2 \times 2^{-1} = \frac{5}{16}$
- $(0.110)_2 \times 2^0 = \frac{6}{8}$, $(0.110)_2 \times 2^1 = \frac{3}{2}$, $(0.110)_2 \times 2^{-1} = \frac{6}{16}$
- $(0.111)_2 \times 2^0 = \frac{7}{8}$, $(0.111)_2 \times 2^1 = \frac{7}{4}$, $(0.111)_2 \times 2^{-1} = \frac{7}{16}$

Altogether there are **31 distinct numbers** (15 positive + zero + 15 negative with duplicates removed).
Key observation: Numbers are **symmetrically but unevenly distributed** about zero.



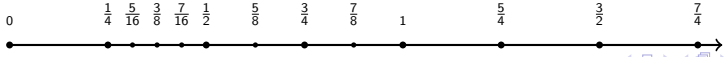
Normalized Floating-Point Representation

Overflow and Underflow

- If a computed value $x = \pm q \times 2^m$ has an exponent m **outside the representable range**, then:
  - *Overflow:* occurs when x exceeds the maximum representable number. This usually triggers a **fatal error or exception**, terminating program execution.
  - *Underflow:* occurs when x is too close to zero. Typically, x is automatically set to **zero**, and execution continues (**possibly with a warning**).
- In Example 5, the system is with **3-bit limited precision** and **exponent range of -1 to +1**:
  - Any value smaller than $\frac{1}{16}$ in magnitude *underflows to zero.*
  - Any value outside the range $[-1.75, +1.75]$ *overflows to machine infinity.*

Hole at Zero Phenomenon

- If only normalized numbers are allowed, each nonzero value has the form $x = \pm(0.1b_2b_3)_2 \times 2^{\pm k}$.
- This leads to the *"hole at zero" phenomenon*: there is a significant gap between zero and the smallest positive normalized number.
- In this system, the smallest normalized positive number is $(0.100)_2 \times 2^{-1} = \frac{1}{4}$.



Outline

Floating-Point Representation in Computers

- **Computers use binary (base 2, $\beta = 2$) floating-point representation.**
- **Finite precision:** Only a *limited number* of bits are used to represent the *mantissa* and *exponent*.
- **Rounding errors:** Not all real numbers can be represented exactly.
- **Precision vs. range trade-off:** **More bits for the mantissa** *increase precision*, while **more bits for the exponent** *increase range*.
- **Special values:** Include representations for positive and negative infinity, zero, and NaN (Not a Number).
- **Common standard:** **IEEE 754** for floating-point arithmetic.
- **There are two main formats:**
  - **Single-precision** (32 bits): used for most applications.
  - **Double-precision** (64 bits): used for more demanding applications requiring greater precision.

Note: *Understanding number representation is helpful for debugging programs.*

Single-Precision: Definition and Components

Definition: A *single-precision floating-point number* is a computer representation of a real number using a **32-bit word** (4 bytes) in memory.

This set is a *finite subset* of real numbers, consisting of:

- ± 0 (positive and negative zero)
- $\pm \infty$ (positive and negative infinity)
- **Normal floating-point numbers** (in normalized form)
- **Subnormal floating-point numbers** (very close to zero)
- **NaN** (Not a Number) values for undefined results (e.g., $0/0$, $\infty - \infty$, $\sqrt{-1}$, etc.)

Important Note: *Most real numbers (e.g., π, e, $\frac{1}{3}$, $\sqrt{2}$, etc.) cannot be represented exactly.*
$\rightarrow$They are approximated by the nearest representable value.

Single-Precision: Bit Allocation

A single-precision number represents:

$$x = \underbrace{(-1)^s}_{\pm \text{ sign}} \times \underbrace{(1.f)_2}_{q \text{ mantissa}} \times \underbrace{2^{c-127}}_{\text{exponent}\beta^m}$$

The 32 bits store these components:

Component	Bits	Description
Sign bit (s)	1	0 for $+$, 1 for $-$
Exponent (c)	8	Biased: $c = m + 127$ where $m \in [-126, 127]$
Mantissa (f)	23	Fraction after implicit leading 1
Total	**32**	

Key insights:

- **Formula uses stored values:** c is what's stored (0–255), $m = c - 127$ is actual exponent
- **Hidden bit:** Leading 1 in $(1.f)_2$ is implicit $\rightarrow$ 24-bit precision (23 stored + 1)
- **Special cases:** $c = 0$ (zero/subnormal), $c = 255$ (infinity/NaN)

Single-Precision: Biased Exponent Explained

Instead of storing signed exponents directly, we use *excess-127 (bias-127)* encoding:

$$\textit{Stored exponent } c = m + 127 \quad \Leftrightarrow \quad \textit{Actual exponent } m = c - 127$$

Why biasing? Simplifies comparison of floating-point numbers in hardware.

Actual m	**Stored c (Binary)**	**Stored c (Decimal)**
-1	01111110	$126 = 127 - 1$
0	01111111	$127 = 127 + 0$
$+1$	10000000	$128 = 127 + 1$
$+5$	10000100	$132 = 127 + 5$
$+127$	11111110	$254 = 127 + 127$
-127	**00000000**	$0 = 127 - 127$
+128	**11111111**	$255 = 127 + 128$

Usable range:

- $c = 0$ (all zeros): Reserved for ± 0 and subnormal numbers
- $c = 255$ (all ones): Reserved for $\pm\infty$ and NaN
- **Usable range:** $c \in [1, 254] \Rightarrow m \in [-126, +127]$

Single-Precision: Mathematical Representation

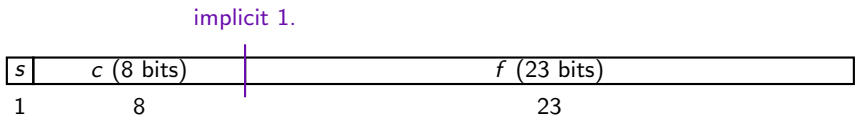
Standard Form: The value of a single-precision floating-point number is given by:

$$x = (-1)^s \times 2^{c-127} \times (1.f)_2$$

where:

- s = sign bit (0 or 1)
- c = biased exponent (8 bits)
- f = fractional part of mantissa (23 bits)

Visual Bit Layout:



Example: $5.0 = 1.01_2 \times 2^2 \Rightarrow s = 0,\ c = 2 + 127 = 129,\ f = 01000\ldots0$

$$[0\,|\,10000001\,|\,01000000000000000000000]_2$$

Single-Precision: Range and Limitations (1/2)

Representable Range: Single-precision (32-bit) floating-point numbers can represent values approximately in the range:

- **Largest representable number:**

$$(2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$$

Derivation: $c_{\max} = 254 \Rightarrow m = 127$; mantissa bits all 1's:

$$(1.\underbrace{11...1}_{23 \text{ bits}})_2 = 1 + \left(\frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^{23}}\right)$$

$$= 1 + \left(1 - \frac{1}{2^{23}}\right) = 2 - 2^{-23}$$

- **Smallest positive normalized number:**

$$1.0 \times 2^{-126} \approx 1.2 \times 10^{-38}$$

Achieved when: $c_{\min} = 1 \Rightarrow m = -126$; mantissa $= 1.0$

Any smaller normalized number would require $c = 0$ (reserved for special values)

Single-Precision: Range and Limitations (2/2)

- **Machine epsilon (relative precision):**

$$\varepsilon_{\text{mach}} = 2^{-23} \approx 1.19 \times 10^{-7}$$

Meaning: Smallest ε such that $1 + \varepsilon \neq 1$ in floating-point
Represents the gap between 1.0 and the next representable number

- **Decimal Precision:** Approximately *7 significant decimal digits*
Reason: 24 bits of mantissa (23 stored + 1 hidden) gives:

$$\log_{10}(2^{24}) = 24 \times \log_{10}(2) \approx 24 \times 0.301 \approx 7.22 \text{ digits}$$

Overflow and Underflow:

- Numbers $> 3.4 \times 10^{38}$ cause **overflow** $\rightarrow$ produce $\pm\infty$
- Numbers $< 1.19 \times 10^{-38}$ cause **underflow** $\rightarrow$ produce 0 or denormalized values

Single-Precision: Encoding Algorithm

Algorithm 1: Convert Real Number x to Single-Precision Format

Input: Real number x
Output: 32-bit representation: $[s \mid c \mid f]$

if $x = 0$ **then**
  return $[0 \mid 00000000 \mid 00000000000000000000000]$ *or* $[1 \mid 00000000 \mid 00000000000000000000000]$
else
  $s \leftarrow 0$ if $x > 0$ else 1 // Determine sign bit
  $y \leftarrow |x|$ // Work with absolute value
  Convert y from decimal to binary: $y = (b_k b_{k-1} \ldots b_1 b_0 . b_{-1} b_{-2} \ldots)_2$
  Normalize to form $1.f \times 2^m$ // Shift binary point left/right as needed
  Calculate biased exponent: $c \leftarrow m + 127$
  Convert c to 8-bit binary
  Extract mantissa fraction f: first 23 bits after binary point
  if f *has* < 23 *bits* **then**
    Pad with trailing zeros
  end
  if f *has* > 23 *bits* **then**
    Round to 23 bits (apply rounding mode)
  end
  Combine: $[s \mid c \mid f]$ as 32-bit word
  return *32-bit binary (optionally convert to hexadecimal)*
end

Single-Precision: Encoding Example

Example 6: Encode -6.5 in single precision

Find the 32-bit single-precision representation of -6.5.

Solution

Step 1: Convert to binary

- $6_{10} = 110_2$ (Divide by 2: $6 \div 2 = 3$, **R = 0**; $3 \div 2 = 1$, **R = 1**; $1 \div 2 = 0$ **R = 1**)
- $0.5_{10} = 0.1_2$ (Multiply by 2: $0.5 \times 2 = 1.0$, **I = 1**; $0.0 \times 2 = 0.0$, **I = 0**)
- Therefore: $6.5_{10} = 110.1_2$

Step 2: Normalize

$$110.1_2 = 1.101_2 \times 2^2$$

Step 3: Extract components

- **Sign bit:** $s = 1$ (negative)
- **Exponent:** $2 + 127 = 129 = 10000001_2$
- **Mantissa:** $1.101_2 \rightarrow$ fraction part $= 101\underbrace{0000\ldots0}_{\text{20 zeros}}$

Step 4: Combine $\boxed{1 \mid 10000001 \mid 10100000000000000000000}$

In hexadecimal: $\boxed{\texttt{0xC0D00000}}$ (It is converted by grouping bits in sets of 4 from left to right and converting each group to its hex equivalent. We start the hexadecimal representation with "0x" to indicate that it is in base 16.)

Single-Precision: Encoding Example

Example 7: Encode -52.234375 in single precision

Find the 32-bit single-precision representation of -52.234375.

Solution

Step 1: Convert integer and fractional parts

- Integer: $(52)_{10} = (64)_8 = (\underbrace{110}_{6}\ \underbrace{100}_{4})_2$
- Fractional: $(0.234375)_{10} = (0.17)_8 = (0.\underbrace{001}_{1}\ \underbrace{111}_{7})_2$
- Combined: $(52.234375)_{10} = (110100.001111)_2$

Step 2: One-plus normalize $(110100.001111)_2 = (1.10100001111)_2 \times 2^5$

Step 3: Determine stored values

- Sign bit: $s = 1$ (negative)
- Exponent: 5, so $c - 127 = 5 \Rightarrow c = 132 = (10000100)_2$
- Mantissa fraction: $f = (10100001111000\ldots0)_2$ (23 bits)

Step 4: Combine into 32-bit word $\boxed{[1\ 10000100\ 10100001111000000000000]_2}$

Convert to hexadecimal (group by 4 bits): $[1100\ 0010\ 0101\ 0000\ 1111\ 0000\ 0000\ 0000]_2 = \boxed{\texttt{0xC250F000}}$

Single-Precision: Decoding Machine Words Example

Example 8: Decode hexadecimal machine words

Find the decimal values represented by: (a) $[0x45DE4000]_{16}$ *and (b)* $[0xBA390000]_{16}$

Solution (a): $[0x45DE4000]_{16}$

Convert to binary: $[0100\,0101\,1101\,1110\,0100\,0000\,0000\,0000]_2$

- Sign: $s = 0$ (positive)
- Exponent: $(10001011)_2 = (213)_8 = 139 \Rightarrow$ actual exp $= 139 - 127 = 12$
- Mantissa: $(1.10111100100)_2 \times 2^{12} = (1101111001000)_2$

Convert mantissa to decimal using octal:

$$(1101111001000)_2 = (001\,101\,111\,001\,000)_2 = (15710)_8 = 1 \times 8^4 + 5 \times 8^3 + 7 \times 8^2 + 1 \times 8^1 + 0$$

Using Horner's method: $8(1 + 8(7 + 8(5 + 8(1)))) = \boxed{7112}$

Single-Precision: Decoding Machine Words Example

Example 8: Decode hexadecimal machine words (continued)

Solution (b): $[0xBA390000]_{16}$

Convert to binary: $[1011\ 1010\ 0011\ 1001\ 0000\ 0000\ 0000\ 0000]_2$

- Sign: $s = 1$ (negative)
- Exponent: $(01110100)_2 = (164)_8 = 116 \Rightarrow$ actual exp $= 116 - 127 = -11$
- Mantissa: $(1.01110010)_2 \times 2^{-11}$

Shift binary point 11 places left:

$$(1.01110010)_2 \times 2^{-11} = (0.00000000001011100100)_2$$

Convert to octal:

$$(0.00000000001011100100)_2 = (0.000\ 000\ 000\ 010\ 111\ 001\ 00\cancel{0})_2 = (0.000271)_8 = 2 \times 8^{-4} + 7 \times 8^{-5} + 1 \times 8^{-6}$$

Using Horner's method: $8^{-6}(1 + 8(7 + 8(2))) = \frac{185}{262144}$

Therefore: $\boxed{-\frac{185}{262144} \approx -7.0571899 \times 10^{-4}}$ (Here we keep only 7 significant digits to reflect single-precision accuracy.)

Double-Precision: Bit Allocation

A double-precision number represents:

$$x = \underbrace{(-1)^s}_{\pm \text{ sign}} \times \underbrace{(1.f)_2}_{q \text{ mantissa}} \times \underbrace{2^{c-1023}}_{\text{exponent}\beta^m}$$

The 64 bits store these components:

Component	Bits	Description
Sign bit (s)	1	0 for $+$, 1 for $-$
Exponent (c)	11	Biased: $c = m + 1023$ where $m \in [-1022, 1023]$
Mantissa (f)	52	Fraction after implicit leading 1
Total	**64**	

Key insights:

- **Formula uses stored values:** c is what's stored (0–2047), $m = c - 1023$ is actual exponent
- **Hidden bit:** Leading 1 in $(1.f)_2$ is implicit $\rightarrow$ 53-bit precision (52 stored + 1)

Double-Precision: Mathematical Representation

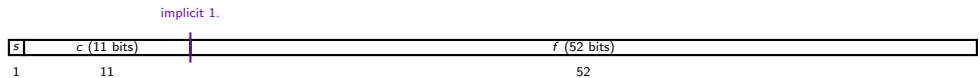
Standard Form: The value of a double-precision floating-point number is given by:

$$x = (-1)^s \times 2^{c-1023} \times (1.f)_2$$

where:

- s = sign bit (0 or 1)
- c = biased exponent (11 bits)
- f = fractional part of mantissa (52 bits)

Visual Bit Layout:



Example: $5.0 = 1.01_2 \times 2^2 \Rightarrow s = 0,\ c = 2 + 1023 = 1025,\ f = 01000\ldots0$

$$[0\,|\,10000000001\,|\,0100000000000000000000000000000000000000000000000000]_2$$

Double-Precision: Range and Limitations (1/2)

Representable Range: Double-precision (64-bit) floating-point numbers can represent values approximately in the range:

- **Largest representable number:**

$$(2 - 2^{-52}) \times 2^{1023} \approx 1.8 \times 10^{308}$$

Derivation: $c_{\max} = 2046 \Rightarrow m = 1023$; mantissa bits all 1's:

$$(1.\underbrace{11\ldots1}_{52 \text{ bits}})_2 = 1 + \left(\frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^{52}}\right)$$

$$= 1 + \left(1 - \frac{1}{2^{52}}\right) = 2 - 2^{-52}$$

- **Smallest positive normalized number:**

$$1.0 \times 2^{-1022} \approx 2.2 \times 10^{-308}$$

Achieved when: $c_{\min} = 1 \Rightarrow m = -1022$; mantissa $= 1.0$

Any smaller normalized number would require $c = 0$ (reserved for special values)

Double-Precision: Range and Limitations (2/2)

- **Machine epsilon (relative precision):**

$$\varepsilon_{\text{mach}} = 2^{-52} \approx 2.22 \times 10^{-16}$$

Meaning: Smallest ε such that $1 + \varepsilon \neq 1$ in floating-point
Represents the gap between 1.0 and the next representable number

- **Decimal Precision:** Approximately *16 significant decimal digits*
Reason: 53 bits of mantissa (52 stored + 1 hidden) gives:

$$\log_{10}(2^{53}) = 53 \times \log_{10}(2) \approx 53 \times 0.301 \approx 15.95 \text{ digits}$$

Overflow and Underflow:

- Numbers $> 1.8 \times 10^{308}$ cause **overflow** $\rightarrow$ produce $\pm\infty$
- Numbers $< 2.2 \times 10^{-308}$ cause **underflow** $\rightarrow$ produce 0 or denormalized values

Note: *Double precision provides* **much wider range** *($\sim 10^{308}$ vs 10^{38}) and* **better accuracy** *(16 vs 7 digits) at the cost of* **twice the memory**.

Double-Precision: Encoding Example

Example 9: Encode -52.234375 in double precision

Find the 64-bit double-precision representation of -52.234375.

Solution

Step 1: Convert integer and fractional parts

- Integer: $(52)_{10} = (64)_8 = (\underbrace{110}_{6}\ \underbrace{100}_{4})_2$
- Fractional: $(0.234375)_{10} = (0.17)_8 = (0.\underbrace{001}_{1}\ \underbrace{111}_{7})_2$
- Combined: $(52.234375)_{10} = (110100.001111)_2$

Step 2: One-plus normalize $(110100.001111)_2 = (1.10100001111)_2 \times 2^5$

Step 3: Determine stored values

- Sign bit: $s = 1$ (negative)
- Exponent: 5, so $c - 1023 = 5 \Rightarrow c = 1028 = (10000000100)_2$
- Mantissa fraction: $f = (1010000111100000\ldots0)_2$ (52 bits)

Step 4: Combine into 64-bit word $[1\ 10000000100\ 1010000111100000000000000000000000000000000000000000000000]_2$

Convert to hexadecimal (group by 4 bits): $[1100\ 0000\ 0100\ 1010\ 0001\ 1110\ 0000\ldots0000]_2 =$ `0xC04A1E0000000000`

Single vs. Double Precision: Side-by-Side

Property	Single (32-bit)	Double (64-bit)
Sign bits	1	1
Exponent bits	8	11
Mantissa bits (stored)	23	52
Total mantissa precision	24 (23+1 hidden)	53 (52+1 hidden)
Exponent bias	127	1023
Exponent range	$[-126, +127]$	$[-1022, +1023]$
Decimal precision	≈ 7 digits	≈ 16 digits
Smallest positive	$\approx 1.2 \times 10^{-38}$	$\approx 2.2 \times 10^{-308}$
Largest positive	$\approx 3.4 \times 10^{38}$	$\approx 1.8 \times 10^{308}$
Machine epsilon	$\approx 1.2 \times 10^{-7}$	$\approx 2.22 \times 10^{-16}$

Single vs. Double Precision: Recommendation

Key Takeaways

When to use which?

- **Single precision:** Sufficient for graphics, games, and simple simulations where memory and speed are critical.
- **Double precision:** Needed for scientific computing, financial calculations, and applications requiring high accuracy.

Caution: Mixing precisions can lead to unexpected results due to rounding errors.

Tip: $\Rightarrow$ For high-accuracy computations, prefer **double precision**!

Special Values in Floating-Point Representation

Floating-point formats reserve specific bit patterns for **exceptional values**.

Value	Sign	Exponent	Mantissa	Meaning
$+0$	0	all 0s	all 0s	Positive zero
-0	1	all 0s	all 0s	Negative zero
$+\infty$	0	all 1s	all 0s	Positive infinity
$-\infty$	1	all 1s	all 0s	Negative infinity
NaN	any	all 1s	non-zero	Not a Number
Denormal	any	all 0s	non-zero	Subnormal number

Key Points:

- ± 0**:** Distinct representations, but $+0 = -0$ numerically
- $\pm\infty$**:** Result of overflow or division by zero (e.g., $1/0$, 10^{400})
- **NaN:** Undefined operations ($0/0$, $\infty - \infty$, $\sqrt{-1}$)
- **Denormals:** Fill gap near zero for *gradual underflow*

Conclusion: *These values allow graceful handling of edge cases without program crashes.*

Machine Epsilon: Definition and Importance

Notation: $\text{fl}(x) =$ *floating-point representation* of real number x (the value stored in computer memory)

Machine Epsilon (ε_{mach}): The **smallest positive number** such that:

$$\text{fl}(1 + \varepsilon_{\text{mach}}) > 1$$

i.e., the smallest ε where $1 + \varepsilon$ is distinguishable from 1 in floating-point arithmetic

What does this mean?

- Represents the **relative precision limit** of the floating-point system
- Equals the **gap between 1.0 and the next representable number**
- Any number smaller than $\varepsilon_{\text{mach}}$ relative to 1.0 will be *rounded away*
- Directly related to mantissa bits: $\varepsilon_{\text{mach}} = \beta^{-t}$ (where β is base, t is mantissa bits number)

For IEEE 754 Binary Formats:

Format	Mantissa bits (t)	$\varepsilon_{\text{mach}} = 2^{-t}$
Single (32-bit)	23	$2^{-23} \approx 1.19 \times 10^{-7}$
Double (64-bit)	52	$2^{-52} \approx 2.22 \times 10^{-16}$

Significant Digits: Converting Binary Precision to Decimal

Key Question: How many **decimal digits** can we represent accurately with t binary mantissa bits?

The Formula:

$$\text{Decimal digits} = t \log_{10}(\beta) \quad \text{where } \beta \text{ is the base}$$

Derivation:

- t bits in base β can represent β^t distinct values
- d decimal digits can represent 10^d distinct values
- For equivalent precision: $10^d = \beta^t$
- Taking $\log_{10}$ of both sides:

$$d = \log_{10}(\beta^t) = t \log_{10}(\beta)$$

- For **binary** ($\beta = 2$): $d = t \times \log_{10}(2) = t \times 0.30103$

Practical Examples:

- **Single precision:** $t = 24$ bits $\Rightarrow d = 24 \times 0.301 = 7.22 \approx \boxed{7}$ decimal digits
- **Double precision:** $t = 53$ bits $\Rightarrow d = 53 \times 0.301 = 15.95 \approx \boxed{16}$ decimal digits

Machine Epsilon and Significant Digits: Summary

Complete Precision Summary:

Format	Mantissa	$\varepsilon_{\text{mach}}$	Calculation	Decimal digits
Single	24 bits	$2^{-23} \approx 1.19 \times 10^{-7}$	24×0.301	≈ 7
Double	53 bits	$2^{-52} \approx 2.22 \times 10^{-16}$	53×0.301	≈ 16

What These Numbers Mean in Practice:

- **Single precision:** Can accurately represent 7 significant decimal digits
  - Example: 3.141592*7* $\approx \pi$ (7th digit may be wrong)
  - Numbers like 1.23456789 will be rounded to 1.234568
- **Double precision:** Can accurately represent 16 significant decimal digits
  - Example: 3.14159265358979*3* $\approx \pi$ (16th digit may be wrong)
  - Much more accurate for scientific computing

Rounding: IEEE Rounding Modes

When a real number cannot be represented exactly, it must be *rounded* to the nearest representable value.

The IEEE 754 standard defines four rounding modes:

1. **Round to nearest (ties to even) - Default:**
   - Rounds to the *nearest* representable value
   - **Tie-breaking rule:** If exactly halfway between two values, round to the one with **even least significant bit**
   - **Why "ties to even"?** Avoids systematic bias (no tendency to round up or down)
   - **Examples:**
     - $2.5 \to 2$ (even), $3.5 \to 4$ (even), $4.5 \to 4$ (even)
     - $1.25 \to 1.2$ (even in last digit), $1.35 \to 1.4$ (even in last digit)
2. **Round toward zero (truncation):** Always rounds the magnitude down (e.g., $-1.8 \to -1$, $+1.8 \to +1$, $1.035 \to 1.03$, $12.999 \to 12.99$)
3. **Round toward $+\infty$ (ceiling):** Always rounds up (e.g., $-1.8 \to -1$, $+1.2 \to +2$)
4. **Round toward $-\infty$ (floor):** Always rounds down (e.g., $-1.2 \to -2$, $+1.8 \to +1$)

Note: $\to$ *The choice of rounding mode affects the* **accuracy**, **bias**, *and* **stability** *of numerical computations.*

Rounding: Rounding Error Example

Example 10: Rounding error with finite precision

Represent $\frac{1}{3}$ in decimal floating-point with 5 significant digits.

Solution

Step 1: Exact value $\frac{1}{3} = 0.333333333333\ldots$ (infinite repeating decimal)

The digit 3 repeats forever — this cannot be stored exactly in finite memory!

Step 2: Rounded representation (5 significant digits) $\text{fl}\left(\frac{1}{3}\right) = 0.33333$

We must truncate or round after 5 digits. Using round-to-nearest, we keep 5 threes.

Step 3: Absolute error (difference between exact and stored)

$$E_{\text{abs}} = |0.333333\ldots - 0.33333| = 0.000003333\ldots = 3.33\ldots \times 10^{-6}$$

The error is approximately 3.33×10^{-6} *in absolute terms.*

Step 4: Relative error (error as a fraction of the true value)

$$E_{\text{rel}} = \frac{E_{\text{abs}}}{|x|} = \frac{3.33\ldots \times 10^{-6}}{1/3} = 3.33\ldots \times 10^{-6} \times 3 = 10^{-5} = 0.001\%$$

Relative error is only 0.001% — quite small, but still present!

Key Insight: Even a *simple rational number* like $\frac{1}{3}$ has **unavoidable rounding error**.

This error propagates through calculations — imagine millions of operations!

Key Takeaways (1/2): IEEE 754 Standard

Key Takeaways

IEEE 754 Structure: $x = (-1)^s \times (1.f)_2 \times 2^{c-\text{bias}}$

- **Single (32-bit):** 1 sign + 8 exponent + 23 mantissa bits
- **Double (64-bit):** 1 sign + 11 exponent + 52 mantissa bits
- **Hidden bit:** Actual precision is 24 bits (single) or 53 bits (double)

Precision:

Format	$\varepsilon_{\text{mach}}$	Decimal digits	Range
Single	$2^{-23} \approx 1.19 \times 10^{-7}$	≈ 7	$\pm 10^{-38}$ to $\pm 10^{38}$
Double	$2^{-52} \approx 2.22 \times 10^{-16}$	≈ 16	$\pm 10^{-308}$ to $\pm 10^{308}$

Machine Epsilon: Smallest number where $\text{fl}(1 + \varepsilon_{\text{mach}}) > 1$
Significant Digits: $d = t \log_{10}(2)$ where t is mantissa bits

Key Takeaways (2/2): Special Values and Practical Considerations

Key Takeaways

IEEE 754 Special Values:

- **Exact and boundary values:** ± 0 (signed zeros), $\pm\infty$ (overflow/division by zero)
- **Error handling:** NaN (invalid operations like $0/0$), Denormals (gradual underflow)

Rounding: Round-to-nearest (default), toward-zero, toward-$\pm\infty$

Practical Guidelines:

- Finite precision causes *rounding errors* and **catastrophic cancellation** in subtraction
- Use **single** (speed/memory) or **double** (accuracy) based on requirements
- *See Chapter 4 for detailed Loss of Significance and Stability!*
- *Next: Practical Python examples showing precision impact!*

Outline

Practical Precision: Python Examples

Demonstrating the Impact of Single vs. Double Precision

Why Precision Matters: Real-World Impact

In numerical computing, the choice between *single precision (32-bit)* and *double precision (64-bit)* can dramatically affect:

- **Accuracy of results** - Rounding errors accumulate differently
- **Stability of algorithms** - Some methods fail with insufficient precision
- **Convergence of iterative methods** - Lower precision may prevent convergence
- **Detection of special conditions** - Subtle differences may be lost

Key Takeaways

In this section, we explore *Python examples* that demonstrate **different results** between single and double precision, helping you understand **when precision matters** in practice.

These examples will show you why scientific computing typically requires double precision!

Python Setup: Controlling Precision

Python's NumPy library allows us to explicitly control precision using data types:

Single Precision (float32):

```python
import numpy as np

# Single precision
x = np.float32(1.0)
print(f"Type: {x.dtype}")
# Output: float32
```

Double Precision (float64):

```python
import numpy as np

# Double precision
x = np.float64(1.0)
print(f"Type: {x.dtype}")
# Output: float64
```

Key Points:

- `np.float32` $\rightarrow$ 32-bit single precision ($\varepsilon_{\text{mach}} \approx 1.19 \times 10^{-7}$)
- `np.float64` $\rightarrow$ 64-bit double precision ($\varepsilon_{\text{mach}} \approx 2.22 \times 10^{-16}$)
- Default Python `float` is double precision (`float64`)

Example 1: Catastrophic Cancellation (Code & Output)

Computing $f(x) = (1 + x) - 1$ for very small $x = 10^{-8}$

Problem: Theoretically, $f(x) = (1 + x) - 1 = x$, so we should get back 10^{-8}.

```python
import numpy as np

x = 1e-8  # Small value

# Single precision
x_single = np.float32(x)
result_single = (np.float32(1.0) + x_single) - np.float32(1.0)

# Double precision
x_double = np.float64(x)
result_double = (np.float64(1.0) + x_double) - np.float64(1.0)

print(f"Expected result:      {x:.15e}")
print(f"Single precision:     {result_single:.15e}")
print(f"Double precision:     {result_double:.15e}")
```

Actual Output:

```
Expected result:      1.000000000000000e-08
Single precision:     0.000000000000000e+00  # 100% error - Complete loss!
Double precision:     9.999999939225290e-09  # ~0.06% error
```

Question: Why does single precision give exactly zero? Why does double precision have error?

Example 1: Detailed Explanation

Let's understand what happens step by step:

Single Precision Analysis (float32):

- Machine epsilon: $\varepsilon_{\text{mach}} = 2^{-23} \approx 1.19 \times 10^{-7}$
- When adding 10^{-8} to 1.0, the relative size is: $\frac{10^{-8}}{1.0} = 10^{-8} < \varepsilon_{\text{mach}}$
- This means 10^{-8} is *below the precision threshold* near 1.0
- Result: $\text{fl}(1.0 + 10^{-8}) = 1.0$ (exactly!) — the 10^{-8} is **completely discarded**
- Therefore: $(1.0 + 10^{-8}) - 1.0 = 1.0 - 1.0 = 0.0$
- **All information lost!** Relative error: $\frac{|0 - 10^{-8}|}{10^{-8}} = 100\%$

Double Precision Analysis (float64):

- Machine epsilon: $\varepsilon_{\text{mach}} = 2^{-52} \approx 2.22 \times 10^{-16}$
- Relative size: $\frac{10^{-8}}{1.0} = 10^{-8} \gg \varepsilon_{\text{mach}}$ (**well above threshold**)
- Can represent $1.0 + 10^{-8}$ with good accuracy
- Small rounding error: gets $9.999999939 \times 10^{-9}$ instead of 1.0×10^{-8}
- Result: $(1.0 + 10^{-8}) - 1.0 \approx 9.999999939 \times 10^{-9}$
- Relative error: $\frac{|9.999999939 \times 10^{-9} - 10^{-8}|}{10^{-8}} \approx 0.06\%$ (**acceptable**)

Example 1: Why This Matters — Catastrophic Cancellation

This phenomenon is called catastrophic cancellation:

What is catastrophic cancellation?

- When subtracting two nearly equal numbers, *leading significant digits cancel*
- The remaining digits were affected by rounding errors in previous operations
- Result: **Massive amplification of relative error**

In our example:

$$1.0 + 10^{-8} \approx 1.00000001 \quad \text{(in decimal)}$$
$$\text{Subtract } 1.0: \quad 1.00000001 - 1.00000000 = 0.00000001$$

The 8 leading zeros after subtraction means we've **lost 8 decimal digits of precision**.

Key Lesson:

- **Single precision (7 digits):** Cannot handle this — all 7 digits used for "1", nothing left for 10^{-8}
- **Double precision (16 digits):** Can handle this — enough digits to represent both 1 and 10^{-8}

Rule of Thumb: *Avoid subtracting nearly equal numbers, or use higher precision!*

Example 2: Summation of Many Small Numbers (Code & Output)

Task: Sum 10 million copies of 10^{-8}. *Expected result:* 0.1

```python
import numpy as np

n = 10_000_000
value = 1e-8

# Single precision
sum_single = np.float32(0.0)
for i in range(n):
    sum_single += np.float32(value)

# Double precision
sum_double = np.float64(0.0)
for i in range(n):
    sum_double += np.float64(value)

print(f"Expected result:      {n * value}")
print(f"Single precision:     {sum_single}")
print(f"Double precision:     {sum_double}")
print(f"Single error:         {abs(sum_single - 0.1):.2e}")
print(f"Double error:         {abs(sum_double - 0.1):.2e}")
```

Actual Output:

```
Expected result:      0.1
Single precision:     0.09210723638534546
Double precision:     0.09999999998802997
Single error:         7.89e-03
Double error:         1.20e-11
```

Example 2: Detailed Explanation

What's happening?

The Problem: Accumulating small numbers through repeated addition

- Each addition: $\text{sum}_{\text{new}} = \text{sum}_{\text{old}} + 10^{-8}$
- Performing this *10 million times*
- Expected final result: $10^7 \times 10^{-8} = 0.1$

Single Precision Analysis:

- Machine epsilon: $\varepsilon_{\text{mach}} = 2^{-23} \approx 1.19 \times 10^{-7}$
- When sum reaches ≈ 0.01, the relative size of 10^{-8} becomes:

$$\frac{10^{-8}}{0.01} = 10^{-6} < 10 \times \varepsilon_{\text{mach}}$$

- At this point, adding 10^{-8} has **barely any effect** — most additions are lost!
- Result: 0.0921 instead of 0.1 $\Rightarrow$ *7.9% error*

Double Precision Analysis:

- Machine epsilon: $\varepsilon_{\text{mach}} = 2^{-52} \approx 2.22 \times 10^{-16}$
- Even when sum reaches 0.1: $\frac{10^{-8}}{0.1} = 10^{-7} \gg 10^{9} \times \varepsilon_{\text{mach}}$
- Each addition is **clearly distinguishable** — minimal loss
- Result: 0.0999999999880 $\Rightarrow$ only 1.2×10^{-11} error

Example 2: Why This Matters

Key Insight: Error Accumulation in Iterative Processes

The Fundamental Problem:

- **Single precision:** After 1 million iterations, sum ≈ 0.01
- From that point, adding 10^{-8} becomes increasingly ineffective
- *Remaining 9 million additions contribute almost nothing!*
- Final error: **650,000× worse** than double precision

Real-World Implications:

1. **Financial calculations:** Summing millions of small transactions
   - Single precision: Could lose \$79,000 on every \$1,000,000!
2. **Scientific simulations:** Long-running simulations with many time steps
   - Accumulated errors can make results completely meaningless
3. **Machine learning:** Gradient descent with millions of updates
   - Poor precision $\Rightarrow$ failure to converge or wrong minima

Rule of Thumb: When accumulating values over *many iterations* ($> 10,000$),
always use double precision or employ **compensated summation algorithms** (e.g., Kahan summation).

Example 3: Quadratic Equation (Code & Output)

Solve $x^2 - 10000.0001x + 1 = 0$ using the standard quadratic formula

```python
import numpy as np

a, b, c = 1.0, -10000.0001, 1.0

# Single precision
a_s, b_s, c_s = np.float32(a), np.float32(b), np.float32(c)
discriminant_s = b_s**2 - 4*a_s*c_s
x1_s = (-b_s + np.sqrt(discriminant_s)) / (2*a_s)
x2_s = (-b_s - np.sqrt(discriminant_s)) / (2*a_s)

# Double precision
a_d, b_d, c_d = np.float64(a), np.float64(b), np.float64(c)
discriminant_d = b_d**2 - 4*a_d*c_d
x1_d = (-b_d + np.sqrt(discriminant_d)) / (2*a_d)
x2_d = (-b_d - np.sqrt(discriminant_d)) / (2*a_d)

print(f"Single precision: x1={x1_s:.10f}, x2={x2_s:.10f}")
print(f"Double precision: x1={x1_d:.10f}, x2={x2_d:.10f}")
print(f"Verification (double): x1*x2 = {x1_d*x2_d:.10f} (should be 1.0)")
print(f"Verification (single): x1*x2 = {x1_s*x2_s:.10f} (should be 1.0)")
```

Actual Output:

```
Single precision: x1=10000.0000000000, x2=0.0000000000
Double precision: x1=10000.0000000000, x2=0.0001000000
Verification (double): x1*x2 = 1.0000000020 (should be 1.0)
Verification (single): x1*x2 = 0.0000000000 (should be 1.0)
```

Example 3: Detailed Explanation

What went wrong in single precision?

The Quadratic Formula: $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$ with $a = 1, b = -10000.0001, c = 1$

Step 1: Calculate discriminant

- $b^2 = 100000002.0001$ (very large)
- $4ac = 4$ (very small)
- $b^2 - 4ac = 100000002.0001 - 4 = 99999998.0001$

Step 2: Single precision problem — only ≈ 7 decimal digits

- $\sqrt{99999998.0001} \approx 9999.9999$ in exact math
- But single precision: $\sqrt{99999998.0001} \approx 10000.0$ **(loses precision!)**
- Why? Single precision can only represent ≈ 7 significant digits

Step 3: Catastrophic cancellation for x_2

$$x_2 = \frac{-(-10000.0001) - \sqrt{99999998.0001}}{2} = \frac{10000.0001 - 10000.0}{2}$$

- Single: $10000.0001 - 10000.0 = 0.0001 \rightarrow$ **rounds to 0.0!**
- Result: $x_2 = 0$ *(completely wrong!)*
- Double: $10000.0001000 - 9999.9999000 = 0.0002000 \rightarrow x_2 = 0.0001$ **(correct)**

Example 3: Why This Matters

Key Insight: Catastrophic Cancellation in Classic Algorithms

The Root Cause:

- When two nearly equal numbers are subtracted: $(10000.0001 - 10000.0)$
- All *leading significant digits cancel out*
- Only trailing digits remain, but single precision doesn't have enough precision there
- Result: **Complete loss of accuracy** — not just error, but **total failure**!

The Mathematical Fix: Use the alternative formula for smaller root

$$x_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}} = \frac{2}{10000.0001 + 10000.0} = \frac{2}{20000.0001} \approx 0.0001$$

This formula **avoids subtraction** *of nearly equal numbers!*

Real-World Implications:

1. **Engineering calculations:** Root finding, optimization, control systems
   - Wrong roots $\Rightarrow$ system instability or failure
2. **Physics simulations:** Solving differential equations, collision detection
   - Incorrect solutions $\Rightarrow$ completely wrong predictions
3. **Graphics/games:** Ray-tracing, intersection calculations
   - Single precision often used for speed — but can cause visual artifacts

Example 4: Numerical Derivative (Code & Output)

Compute $f'(x) = \frac{f(x+h)-f(x)}{h}$ *for* $f(x) = e^x$ *at* $x = 1.0$ *with* $h = 10^{-5}$

```python
import numpy as np
def f(x):
    return np.exp(x)
x = 1.0
h = 1e-5
exact_derivative = np.exp(1.0)  # True derivative at x=1

# Single precision
x_s, h_s = np.float32(x), np.float32(h)
deriv_single = (f(x_s + h_s) - f(x_s)) / h_s

# Double precision
x_d, h_d = np.float64(x), np.float64(h)
deriv_double = (f(x_d + h_d) - f(x_d)) / h_d

print(f"Exact derivative:     {exact_derivative:.10f}")
print(f"Single precision:     {deriv_single:.10f}")
print(f"Double precision:     {deriv_double:.10f}")
print(f"Single error:         {abs(deriv_single - exact_derivative):.2e}")
print(f"Double error:         {abs(deriv_double - exact_derivative):.2e}")
```

Actual Output:

```
Exact derivative:     2.7182818285
Single precision:     2.7418136597
Double precision:     2.7182954200
Single error:         2.35e-02
Double error:         1.36e-05
```

Example 4: Detailed Explanation

Understanding Numerical Differentiation Errors

The Forward Difference Formula:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad \text{for small } h$$

For $f(x) = e^x$ at $x = 1$: exact derivative is $f'(1) = e^1 \approx 2.7182818285$

Single Precision Analysis: ($h = 10^{-5}$)

- $f(1 + 10^{-5}) - f(1) = e^{1.00001} - e^1 \approx 0.000027183$
- Dividing by $h = 10^{-5}$ gives: $0.000027183/10^{-5} = 2.7183$
- But single precision loses digits in both:
  - Computing $f(1.00001)$: limited to 7 digits
  - Subtracting nearly equal numbers: loses leading digits
  - Dividing by small h: amplifies remaining errors
- Result: **2.7418** instead of 2.7183 $\Rightarrow$ *0.86% error*

Double Precision Analysis:

- Same calculation, but with 16 decimal digits throughout
- Subtraction: $2.7182954 - 2.7182818 = 0.0000136$ preserved accurately
- Result: 2.7182954 $\Rightarrow$ only **0.0005% error** (1,700$\times$ better!)

Example 4: Why This Matters

Key Insight: Numerical Derivatives Amplify Rounding Errors (The Fundamental Trade-off)

- **Smaller h:** Better approximation to true derivative (truncation error $\downarrow$)
- **Smaller h:** Worse rounding error because dividing by smaller number (rounding error $\uparrow$)
- **Optimal h:** Balances truncation vs. rounding error
- With single precision: optimal $h \approx \sqrt{\varepsilon_{\text{mach}}} \approx 10^{-4}$ (not 10^{-5}!)
- With double precision: optimal $h \approx 10^{-8}$ (much more flexibility)

Real-World Implications:

1. **Optimization algorithms:** Gradient descent, Newton's method
   - Poor gradients $\Rightarrow$ wrong search direction, slow/failed convergence
2. **Physics simulations:** Computing forces, velocities, accelerations
   - Errors in derivatives $\Rightarrow$ accumulated errors in trajectories
3. **Machine learning:** Backpropagation, gradient-based training
   - Inaccurate gradients $\Rightarrow$ poor model performance
4. **Automatic differentiation:** Modern ML frameworks avoid this problem
   - Use symbolic/algorithmic differentiation instead of finite differences

Rule of Thumb: Numerical differentiation is *highly sensitive* to precision.

For critical applications, use **double precision** or **automatic differentiation** (AD).

Example 5: Newton-Raphson Method (Code)

Find $\sqrt{2}$ using Newton-Raphson: $x_{n+1} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right)$

```python
import numpy as np
def newton_sqrt2_single(x0, max_iter=20):
    x = np.float32(x0)
    for i in range(max_iter):
        x_new = np.float32(0.5) * (x + np.float32(2.0) / x)
        if abs(x_new - x) < np.float32(1e-10):
            return x_new, i+1
        x = x_new
    return x, max_iter
def newton_sqrt2_double(x0, max_iter=20):
    x = np.float64(x0)
    for i in range(max_iter):
        x_new = 0.5 * (x + 2.0 / x)
        if abs(x_new - x) < 1e-10:
            return x_new, i+1
        x = x_new
    return x, max_iter
x0 = 1.5
result_single, iters_single = newton_sqrt2_single(x0)
result_double, iters_double = newton_sqrt2_double(x0)
exact = np.sqrt(2.0)
print(f"Exact sqrt(2):        {exact:.15f}")
print(f"Single precision:     {result_single:.15f} (converged in {iters_single} iters)")
print(f"Double precision:     {result_double:.15f} (converged in {iters_double} iters)")
print(f"Single error:         {abs(result_single - exact):.2e}")
print(f"Double error:         {abs(result_double - exact):.2e}")
```

Example 5: Newton-Raphson Method (Output)

Actual Output:

```
Exact sqrt(2):        1.414213562373095
Single precision:     1.414213538169861 (converged in 4 iters)
Double precision:     1.414213562373095 (converged in 4 iters)
Single error:         2.42e-08
Double error:         2.22e-16
```

Example 5: Detailed Explanation

Understanding Iterative Method Convergence Limits

Newton-Raphson for $\sqrt{2}$: Solving $f(x) = x^2 - 2 = 0$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - 2}{2x_n} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right)$$

Starting from $x_0 = 1.5$, this converges *quadratically* (doubles accuracy each iteration)

Single Precision Analysis:

- Iteration 1: $x_1 = \frac{1}{2}(1.5 + 2/1.5) = 1.41666\ldots$
- Iteration 2: $x_2 = 1.41421\ldots$
- Iteration 3: $x_3 = 1.414213\ldots$
- Iteration 4: $x_4 = 1.41421354$ (converged!)
- **But:** Final answer is 1.414213538 vs exact 1.414213562
- Error: $2.42 \times 10^{-8} \approx \varepsilon_{\text{mach}}/5$ for single precision
- *Cannot improve further!* Rounding errors dominate

Double Precision Analysis:

- Same iteration count (4 iterations)
- Final answer: 1.414213562373095 (exact to machine precision!)
- Error: $2.22 \times 10^{-16} = \varepsilon_{\text{mach}}$ for double precision
- Achieved **maximum possible accuracy** for this precision

Example 5: Why This Matters (1/2)

Key Insight: Machine Epsilon is the Ultimate Convergence Barrier

The Fundamental Limitation:

- **Single precision:** Best possible accuracy $\sim \varepsilon_{\text{mach}} \approx 10^{-7}$
  - Result: 2.42×10^{-8} error (about $\varepsilon_{\text{mach}}/5$)
  - *Cannot distinguish improvements smaller than ε_{mach}*
- **Double precision:** Best possible accuracy $\sim \varepsilon_{\text{mach}} \approx 10^{-16}$
  - Result: 2.22×10^{-16} error (exactly $\varepsilon_{\text{mach}}$!)
  - Achieved theoretical maximum accuracy
- **Same iterations, vastly different final accuracy!**

Key Point: *Convergence speed $\neq$ final accuracy*.
Newton-Raphson converged in 4 iterations for both precisions, but **machine epsilon determines final precision**.

Example 5: Why This Matters (2/2)

Real-World Implications:

1. **Root finding:** Solving equations, optimization
   - Single: accurate to 7 decimal places maximum
   - Critical for engineering calculations where high precision matters
2. **Linear algebra:** Solving large systems, eigenvalue problems
   - Iterative solvers (CG, GMRES) hit precision wall
   - Single precision: may not satisfy tight convergence criteria
3. **Scientific computing:** Climate models, molecular dynamics
   - Long simulations: accumulated errors compound over time
   - Single precision errors can invalidate results after many steps

Rule of Thumb: Iterative methods converge at same rate, but **machine epsilon determines final precision**. For high-accuracy requirements, **double precision is essential**.

Summary: When Precision Matters

From our Python examples, we learned:

1. **Catastrophic Cancellation:** Subtracting nearly equal numbers $\Rightarrow$ **Use double precision**
2. **Accumulation:** Summing many small numbers $\Rightarrow$ **Errors compound quickly in single precision**
3. **Ill-conditioned Problems:** Quadratic equations with $b^2 \gg 4ac$ $\Rightarrow$ **Single precision fails**
4. **Numerical Differentiation:** Finite differences amplify errors $\Rightarrow$ **Need higher precision**
5. **Iterative Methods:** Convergence limited by $\varepsilon_{\text{mach}}$ $\Rightarrow$ **Double precision for accuracy**

General Guidelines:

- **Scientific computing:** Always use **double precision (float64)**
- **Graphics/games:** Single precision often sufficient
- **Machine learning:** Often uses `float32` or even `float16` (speed $>$ accuracy)
- **Financial calculations:** Use `decimal` module (arbitrary precision)

When in doubt, choose double precision!

End of Chapter 3