COM701 - Numerical Mathematics and Computing

Chapter 4 - Loss of Significance and Stability

Dr. Mahdi Khemakhem

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

AY - 2025/2026

Outline

Outline

Floating-Point Operations

To analyze error, we must distinguish between **exact arithmetic** and **computer arithmetic**. We use special symbols for the *floating-point operations*, which **include rounding**.

Operation	**Real Math (Exact)**	**Floating-Point (Rounded)**
Addition	$x + y$	$x \oplus y = \text{fl}(x + y)$
Subtraction	$x - y$	$x \ominus y = \text{fl}(x - y)$
Multiplication	$x \times y$	$x \otimes y = \text{fl}(x \times y)$
Division	$x \div y$	$x \oslash y = \text{fl}(x \div y)$

$\rightarrow$ **fl**(x) *means "round the exact result to the nearest floating-point number."*

The Model for Floating-Point Error

The notation **fl**(x) represents the **floating-point representation** of the real number x.

The Standard Error Model

We model the rounding error using *machine epsilon* (ϵ_{mach}).

$$\text{fl}(x) = x(1+\delta), \quad \text{where } |\delta| \leq \epsilon_{\text{mach}}$$

- δ is the **relative error** introduced by rounding.
- ϵ_{mach} is the **maximum possible relative error** for storing a number.

Error in Operations: This rounding error occurs *after every single arithmetic operation*.

- $x \oplus y = \text{fl}(x+y) = (x+y)(1+\delta_1)$, where $|\delta_1| \leq \epsilon_{\text{mach}}$
- $x \otimes y = \text{fl}(x \times y) = (x \times y)(1+\delta_2)$, where $|\delta_2| \leq \epsilon_{\text{mach}}$
- $x \ominus y = \text{fl}(x-y) = (x-y)(1+\delta_3)$, where $|\delta_3| \leq \epsilon_{\text{mach}}$
- $x \oslash y = \text{fl}(x \div y) = (x \div y)(1+\delta_4)$, where $|\delta_4| \leq \epsilon_{\text{mach}}$

$\rightarrow$ *This model allows us to* *analyze error propagation* *in numerical algorithms.*

Floating-Point Arithmetic is Not Real Arithmetic

Key Takeaways

Because of the rounding error on every step, floating-point arithmetic is **not exact**. This means **standard algebraic properties fail**.

- **Associativity Fails:** *e.g.*, $(a \oplus b) \oplus c \neq a \oplus (b \oplus c)$
  The order of operations now matters.
- **Distributivity Fails:** *e.g.*, $a \otimes (b \oplus c) \neq (a \otimes b) \oplus (a \otimes c)$
- **Absorption (or "Swallowing"):** Adding a small number to a large one may have no effect. *e.g., If* $(Large + tiny)$ *rounds back to Large.*
- **Catastrophic Cancellation:** Subtracting two nearly-equal numbers can cause a massive loss of relative precision. e.g., *if* $x \approx y$*, then* $x - y$ *may lose significant digits due to rounding errors in* x *and* y*.*

Loss of Significance

Loss of Associativity When performing arithmetic operations on floating-point numbers, the order of operations can affect the final result due to rounding errors.

Example 1: Loss of Associativity in Floating-Point Arithmetic

In floating-point arithmetic, addition is **not associative**. That means: $(a+b)+c$ **may differ from** $a+(b+c)$ because intermediate results are **rounded** after each operation.

Python demonstration (single precision float32):

```
import numpy as np

a = np.float32(1e-5)    # 0.00001
b = np.float32(1e3)     # 1000.0
c = np.float32(-1e3)    # -1000.0

expr1 = (a + b) + c
expr2 = a + (b + c)

print("(a + b) + c =", expr1)
print("a + (b + c) =", expr2)
```

Observed output:

```
(a + b) + c = 0.0
a + (b + c) = 1e-05
```

Question: Why are these results different? Let's analyze step by step.

Loss of Significance

Example 1: Loss of Associativity in Floating-Point Arithmetic (continued)

Key Fact: Single precision (`float32`) has only *23 bits for the mantissa* (fraction).
This gives approximately **7 decimal digits** of precision.
Analyzing $(a + b)$ **where** $a = 0.00001$ **and** $b = 1000.0$
The exact result is: $a + b = 0.00001 + 1000.0 = 1000.00001$
Problem: To represent both the large part (1000) and small part (0.00001), we need:

- 1000.0 in binary: $1111101000.0_2 = 1.\mathbf{1111010000}_2 \times 2^9$
- 0.00001 in binary: $\approx 0.\mathbf{0000000000001010001111010 1}\ldots_2$
- Combined: $1000.00001 = 1.\underbrace{\mathbf{11110100000000000000001}}_{\text{23 bits}}\mathbf{0100011110101}\ldots_2 \times 2^9$ requires bits beyond the 23-bit mantissa limit

Critical Issue: The mantissa has only 23 bits, but to represent the small addition 0.00001 at the magnitude of 1000, we need bits far beyond the 23rd position!

Result: The small value 0.00001 is *below the precision threshold* at magnitude ~ 1000 and gets **completely lost** during rounding.

$$\text{fl}(a + b) = \text{fl}(1000.00001) = 1000.0$$

The value a is entirely discarded!

Loss of Significance

Example 1: Loss of Associativity in Floating-Point Arithmetic (continued)

Expression 1: $(a + b) + c$

1. **First operation:** $a + b = 0.00001 + 1000.0$
   - Exact result: 1000.00001
   - After rounding (float32): $\text{fl}(a + b) = 1000.0$ **(0.00001 is completely lost!)**
2. **Second operation:** $(a + b) + c = 1000.0 + (-1000.0)$
   - Exact result: 0.0
   - After rounding: $\text{fl}(1000.0 - 1000.0) = 0.0$

Final Result: $(a + b) + c = 0.0$

The original value $a = 0.00001$ has vanished completely! This is catastrophic cancellation in action: the small value is absorbed by the large value, then the large values cancel, leaving nothing.

Loss of Significance

Example 1: Loss of Associativity in Floating-Point Arithmetic (continued)

Expression 2: $a + (b + c)$

1. **First operation:** $b + c = 1000.0 + (-1000.0)$
   - Exact result: 0.0
   - After rounding: $\text{fl}(b + c) = 0.0$ **(exact cancellation!)**
2. **Second operation:** $a + (b + c) = 0.00001 + 0.0$
   - Exact result: 0.00001
   - After rounding: $\text{fl}(0.00001 + 0.0) = 0.00001$ **(preserves original value!)**

Final Result: $a + (b + c) = 0.00001 = 1 \times 10^{-5}$

Key Difference:

- In this order, 0.00001 never needs to be represented at the magnitude of 1000
- The large values cancel exactly first, leaving only the small value
- **No precision is lost — the original value is perfectly preserved!**

Loss of Significance

Example 1: Loss of Associativity in Floating-Point Arithmetic (continued)

The Root Cause: Machine Epsilon and Magnitude

Machine epsilon for float32: $\varepsilon_{\text{mach}} \approx 1.19 \times 10^{-7}$ (relative precision)

Relative size analysis:

- At magnitude ~ 1000, the spacing between consecutive representable float32 numbers is:

$$\text{spacing} \approx 1000 \times \varepsilon_{\text{mach}} \approx 1000 \times 1.19 \times 10^{-7} \approx 1.19 \times 10^{-4}$$

- Our small value is: $0.00001 = 1 \times 10^{-5}$
- Comparing: $10^{-5} < 10^{-4}$ (spacing at magnitude 1000)
- This means 0.00001 is **smaller than the spacing** between representable numbers at magnitude 1000!
- When added to 1000, the relative size is:

$$\frac{0.00001}{1000} = 10^{-8} \ll \varepsilon_{\text{mach}}$$

- Therefore, 0.00001 **cannot be distinguished** from zero at magnitude 1000 and is completely lost!

Loss of Significance

Example 1: Loss of Associativity in Floating-Point Arithmetic (continued)

Key Insights:

1. **Order matters:** $(a + b) + c = 0.0$ loses a when computing $a + b$
2. **Exact cancellation helps:** $a + (b + c) = 0.00001$ benefits from $b + c = 0$ being exact
3. **Magnitude differences:** Adding numbers of vastly different magnitudes loses precision
4. **Loss of associativity:** $(a + b) + c \neq a + (b + c)$ violates basic arithmetic rules

Key Takeaways

Practical Lesson: When adding many numbers, *sort them by magnitude* and add from **smallest to largest** to minimize precision loss!

Loss of Significance

Loss of Significance in Subtraction

When two **nearly equal numbers** are subtracted, **most of their significant digits may cancel out**, leading to a **large relative error**.

Loss of significance in subtraction can be quantified using the following theorem:

Theorem: Loss of Significance in Subtraction

Let x and y be two positive floating-point numbers with $x > y > 0$. When computing the difference $x - y$ in floating-point arithmetic, the number of bits of significance lost is at least k if:

$$\frac{1}{2^k} \leq 1 - \frac{y}{x} < \frac{1}{2^{k-1}}$$

In other words, the **relative size** of y to x determines how many bits of precision are lost in the subtraction.

Loss of Significance

Loss of Significance in Subtraction

Example 2: Applying the theorem

In the subtraction $37.593621 - 37.584216$, *how many bits of significance are lost?*

Solution

Let $x = 37.593621$ and $y = 37.584216$. Then:

$$1 - \frac{y}{x} = 1 - \frac{37.584216}{37.593621} \approx 0.0002501754$$

Compare with powers of 2:

$$\frac{1}{2^{12}} = 0.000244 \leq 0.0002501754 < \frac{1}{2^{11}} = 0.000488$$

Therefore: **At least 11 but not more than 12 bits are lost.**

Loss of Significance

Loss of Significance in Subtraction (Amplification of relative error)

Example 3: Loss of Significance in Subtraction (Amplification of relative error)

Suppose $x = 0.3721448693$ *and* $y = 0.3720214371$. *What is the relative error in computing* $x - y$ *using* **5 significant digits**?

Solution

First, we round the inputs to 5 significant digits: $\tilde{x} = 0.37214$ and $\tilde{y} = 0.37202$
The computed difference is: $\tilde{x} - \tilde{y} = 0.37214 - 0.37202 = 0.00012$
The true difference is: $x - y = 0.3721448693 - 0.3720214371 = 0.0001234322$
The relative error in the result is: $\frac{|(\tilde{x}-\tilde{y})-(x-y)|}{|x-y|} = \frac{|0.00012-0.0001234322|}{0.0001234322} = \frac{0.0000034322}{0.0001234322} \approx 2.78 \times 10^{-2} \approx 0.0278 \approx 3\%$

Observation: The relative error of the inputs was small ($\approx 0.0013\%$), e.g, $\frac{|\tilde{x}-x|}{|x|} = \frac{|0.37214-0.3721448693|}{|0.3721448693|} = \frac{0.0000048693}{0.3721448693} = 0.00001308 \approx 10^{-5}$, but the relative error of the substraction result is large ($\approx 3\%$).
We lost almost all accuracy!.

$\rightarrow$ *This is the* *Amplification of Relative Error* *due to* **Loss of Significance**.

Loss of Significance

Example 3: Loss of Significance in Subtraction (Analysis)

Why did this happen?

The computed difference $\tilde{x} - \tilde{y} = 0.00012$ has only **2 significant digits**, while the inputs $\tilde{x}$ and $\tilde{y}$ had 5. The subtraction **cancelled the leading, correct digits (0.372)**, leaving behind a result (0.00012) that is dominated by the original rounding errors in $\tilde{x}$ and $\tilde{y}$.

$\Rightarrow$ This effect is called *Catastrophic Cancellation*.

Deeper Analysis

A 5-digit rounding system (base $\beta = 10, t = 5$) has a **unit roundoff**: $\mathbf{u} = \frac{1}{2}\beta^{1-t} = \frac{1}{2} \times 10^{-4} = 5 \times 10^{-5}$ where β is the base and t is the number of significant digits. (**u**: *represents the maximum possible relative error that can happen when you round any real number to its nearest floating-point representation.*)

This means storing x and y introduces a small relative error: $\tilde{x} = x(1+\delta_1)$, where $|\delta_1| \leq \mathbf{u}$ and $\tilde{y} = y(1+\delta_2)$, where $|\delta_2| \leq \mathbf{u}$.

Let's analyze the error in the computed difference $d_{comp} = \tilde{x} - \tilde{y}$: $d_{comp} = x(1 + \delta_1) - y(1 + \delta_2) = (x - y) + (x\delta_1 - y\delta_2)$

The absolute error is $d_{comp} - d_{true} = x\delta_1 - y\delta_2$. The **relative error** is $\frac{|d_{comp} - d_{true}|}{|d_{true}|} = \frac{|x\delta_1 - y\delta_2|}{|x-y|}$.

Using the triangle inequality and $|\delta_i| \leq \mathbf{u}$: $\frac{|x\delta_1 - y\delta_2|}{|x-y|} \leq \frac{|x||\delta_1| + |y||\delta_2|}{|x-y|} \leq \frac{|x|\mathbf{u} + |y|\mathbf{u}}{|x-y|} \leq \left(\frac{|x|+|y|}{|x-y|}\right)\mathbf{u}$

In our example, this **amplification factor** is: $\left(\frac{0.37214\ldots + 0.37202\ldots}{0.000123\ldots}\right) \approx \frac{0.744}{0.000123} \approx \mathbf{6000}$

Conclusion: The initial relative error $\mathbf{u}$ (5×10^{-5}) was amplified by ≈ 6000 times!

How to Avoid Catastrophic Cancellation?

To prevent catastrophic cancellation, we can use several techniques to reformulate computations.

Technique 1: This can often be achieved by **algebraic reformulation**.

Example 4: $f(x) = \sqrt{x+1} - \sqrt{x}$ for large $x = 1,000,000$

- Let $x = 1,000,000 \Rightarrow \sqrt{x+1} \approx 1000.000500...$ and $\sqrt{x} = 1000.000000...$
- If computed with 8-digit precision:
  - $\tilde{a} = 1000.0005$ and $\tilde{b} = 1000.0000$
  - $\tilde{a} - \tilde{b} = 0.0005$ (Only 1 significant digit!)
- The true answer is $\approx 0.00049999987...$ **The relative error is large.**

Solution: Reformulate using the conjugate.

$$f(x) = (\sqrt{x+1} - \sqrt{x}) \cdot \frac{\sqrt{x+1}+\sqrt{x}}{\sqrt{x+1}+\sqrt{x}} = \frac{(x+1)-x}{\sqrt{x+1}+\sqrt{x}} = \frac{1}{\sqrt{x+1}+\sqrt{x}}$$

- Using this new formula with $x = 1,000,000$:
  $f(x) = \frac{1}{1000.0005+1000.0000} = \frac{1}{2000.0005} \approx 0.00049999987...$
- This expression involves **addition** and is *numerically stable*.

How to Avoid Catastrophic Cancellation?

Technique 2: Use **Taylor Series approximations** for x near a point of cancellation.

Example 5: $f(x) = \frac{1-\cos(x)}{x^2}$ for $x \approx 0$

- As $x \to 0$, $\cos(x) \to 1$.
- The numerator $1 - \cos(x)$ suffers from catastrophic cancellation.
- Let $x = 10^{-4}$. In 8-digit precision:
  - $\cos(10^{-4}) \approx 0.9999999950... \implies \overline{\cos(x)} = 1.0000000$
  - $f(x) = \frac{1-1.0000000}{(10^{-4})^2} = \frac{0}{10^{-8}} = 0$ (**Completely wrong!**)

Solution: Use the Taylor expansion for $\cos(x)$ near $x = 0$. $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$

Substitute this into the formula: $f(x) = \frac{1-\left(1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}-\cdots\right)}{x^2} = \frac{\frac{x^2}{2!}-\frac{x^4}{4!}+\frac{x^6}{6!}-\cdots}{x^2} = \frac{1}{2} - \frac{x^2}{24} + \frac{x^4}{720} - \dots$

- For $x \approx 0$, the stable computation is $f(x) \approx \frac{1}{2}$.
- This avoids subtraction entirely and is *highly accurate*.

How to Avoid Catastrophic Cancellation?

Technique 3: Use **Trigonometric Identities** to reformulate expressions.

Example 6: $f(x) = \cos^2(x) - \sin^2(x)$ near $x = \frac{\pi}{4}$

- Near $x = \frac{\pi}{4}$, we have $\cos(x) \approx \sin(x) \approx \frac{1}{\sqrt{2}}$.
- Thus $\cos^2(x) \approx \sin^2(x) \approx 0.5$.
- Let $x = \frac{\pi}{4} + 10^{-5}$. In 8-digit precision:
  - $\cos^2(x) \approx 0.49999999...$
  - $\sin^2(x) \approx 0.50000001...$
  - Direct computation: $f(x) = 0.49999999 - 0.50000001 = -0.00000002$ (**Lost precision!**)

Solution: Use the double-angle identity.

Recall: $\cos(2\theta) = \cos^2(\theta) - \sin^2(\theta)$

Therefore: $f(x) = \cos^2(x) - \sin^2(x) = \cos(2x)$

- Using this formula with $x = \frac{\pi}{4} + 10^{-5}$: $f(x) = \cos(2x) = \cos\left(\frac{\pi}{2} + 2 \times 10^{-5}\right) \approx -2 \times 10^{-5}$
- This expression is *numerically stable* and avoids the problematic subtraction.

How to Avoid Catastrophic Cancellation?

Technique 4: Use **Range Reduction** for periodic functions with large arguments.

Example 7: Computing $\sin(x)$ for very large x

- To compute $\sin(x)$ for large x, we use **periodicity**: $\sin(x) = \sin(x - 2n\pi)$ where n is an integer such that $x - 2n\pi \in [0, 2\pi)$.
- Let $x = 10^8$. We need to find n such that $x' = x - 2n\pi \in [0, 2\pi)$. $\Rightarrow n = \lfloor \frac{x}{2\pi} \rfloor \approx 15915494$
- The subtraction $x - 2n\pi$ involves two very large, nearly equal numbers!

Analysis using Loss of Precision Theorem:
For the subtraction $x - 2n\pi$, we compute: $1 - \frac{2n\pi}{x} = \frac{x-2n\pi}{x} < \frac{2\pi}{x} = \frac{2\pi}{10^8} \approx 6.28 \times 10^{-8}$
Since $2^{-24} \approx 5.96 \times 10^{-8} < 6.28 \times 10^{-8} < 2^{-23}$, we lose **at least 23 bits**!

- In single precision (23 mantissa bits), the result is **completely meaningless**!
- **Conclusion:** For very large arguments, computing $\sin(x)$ requires *high-precision arithmetic* or *accurate input*.
- This is an example of an *ill-conditioned problem*: the result is extremely sensitive to small errors in x.

Solution: Always reduce the range of x before computing trigonometric functions to avoid catastrophic cancellation.

How to Avoid Catastrophic Cancellation?

Key Takeaways

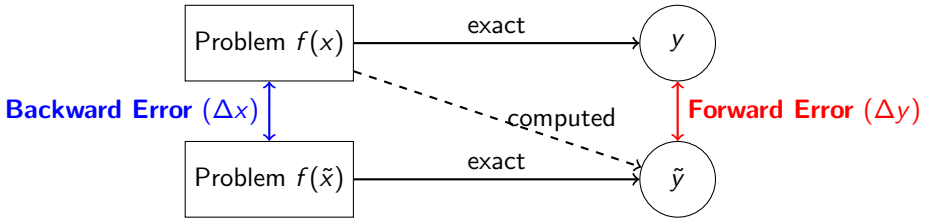
To avoid catastrophic cancellation and loss of significance:

- **Algebraic Reformulation:** Change the way the problem is expressed to avoid numerical issues.
- **Taylor Series:** Use series expansions to bypass problematic operations.
- **Trigonometric Identities:** Leverage identities to find equivalent but more stable formulations.
- **Range Reduction:** For periodic functions, reduce the range of the input to minimize loss.

Forward vs. Backward Error Analysis

When our computed answer $\tilde{y}$ is not the true answer y, we can analyze the error in two ways:

- *Forward Error* **Asks about the answer.** *"How close is my computed answer $\tilde{y}$ to the true answer y?"*
- *Backward Error* **Asks about the problem.** *"Is my computed answer $\tilde{y}$ the* **exact** *solution to a* **nearby** *problem $f(\tilde{x})$?"*



Forward vs. Backward Error Analysis

Example 8: $f(x) = \sqrt{x}$

Suppose we want to compute $y = f(2) = \sqrt{2} \approx 1.414213...$

Due to rounding, our calculator gives the computed answer $\tilde{y} = 1.414$

- **Forward Error:** How close is the answer?
  $|y - \tilde{y}| = |1.414213... - 1.414| \approx 0.000213$
- **Backward Error:** What problem did we **actually** solve?
  We find $\tilde{x}$ such that $f(\tilde{x}) = \tilde{y}$.
  $\sqrt{\tilde{x}} = 1.414 \implies \tilde{x} = (1.414)^2 = 1.999396$
  The backward error is $|\tilde{x} - x| = |1.999396 - 2| = 0.000604$.

So, our answer $\tilde{y} = 1.414$ is *exactly correct* for the (nearby) problem $f(1.999396)$.

What is a Backward Stable Algorithm?

An algorithm is *backward stable* if the backward error Δx is **small** for all inputs.
This means the algorithm *guarantees* that the computed answer $\tilde{y}$ is the **exact solution** to a **nearby problem** $\tilde{x}$.

The Big Idea

A backward stable algorithm gives (nearly) the right answer to a (nearly) right question.

Formal Definition

An algorithm is backward stable if the computed $\tilde{y}$ satisfies:

$$\tilde{y} = f(\tilde{x}) \quad \text{and} \quad \frac{\|\tilde{x} - x\|}{\|x\|} = O(\epsilon_{\text{mach}})$$

$\rightarrow$ *The error is as if the original data had a small error on the order of machine precision.*

What is a Backward Stable Algorithm?

Example 8: Was our $\sqrt{x}$ algorithm stable?

Yes. Our algorithm (rounding the output of $\sqrt{2}$) produced a backward error of ≈ 0.0006. This is very small.
The error we got ($\tilde{y} = 1.414$) is **not** the algorithm's fault. It's as if our original data was slightly off ($x = 1.999396$ instead of $x = 2$).
A backward stable algorithm's error is "as small as" the initial error from just storing the data.

Why We Want Backward Stability

Backward stability is the "gold standard" for numerical algorithms.

Key Takeaways

- **It separates concerns:** It allows us to distinguish between:
  - *Error from the algorithm* (Is it stable?).
  - *Error from the problem* (Is it sensitive? i.e., **ill-conditioned**?)
- **It's the "best" we can do:** A backward stable algorithm doesn't amplify errors. It does its job as accurately as the (im)precision of the data allows.
- **It's a design goal:** Most fundamental numerical Python libraries (e.g., NumPy, SciPy) for solving $Ax = b$ or finding eigenvalues are **designed to be backward stable**.

Why We Want Backward Stability

Example 9: An Ill-Conditioned Problem

Consider $f(x) = \tan(x)$ near $x = \pi/2 \approx 1.570796...$
Let's say our algorithm is **perfectly backward stable**. It promises to give us the exact answer for an input $\tilde{x}$ that is **very** close to x.

- **Problem 1:** $x = 1.5707$
- True Answer: $f(x) \approx 3279.8...$
- **Problem 2 (Nearby):** $\tilde{x} = 1.5708$ (a *tiny* backward error!)
- Exact Answer: $f(\tilde{x}) \approx 10381.3...$

Conclusion: The algorithm did its job! It gave a correct answer for a nearby problem. The *problem itself* is so sensitive (**ill-conditioned**) that a tiny change in the input causes a massive change in the output.

→ *Backward stability means: "Our algorithm is not to blame for this inaccuracy; the problem is."*

Outline

What is a Problem's Condition Number?

The *condition number* (κ) measures how **sensitive** a problem is to **perturbations in the input**.

Definition of Condition Number

For a problem $y = f(x)$, the **relative condition number** is:

$$\kappa = \lim_{\delta \to 0} \sup_{\|\Delta x\| \leq \delta \|x\|} \frac{\|\Delta y\|/\|y\|}{\|\Delta x\|/\|x\|} \approx \left| \frac{x f'(x)}{f(x)} \right|$$

Well-Conditioned ($\kappa \approx 1$)

- κ is small
- Small input changes $\rightarrow$ small output changes

Ill-Conditioned ($\kappa \gg 1$)

- κ is large
- Small input changes $\rightarrow$ large output changes

$\rightarrow$ *High condition numbers mean the problem is* **inherently sensitive**, *regardless of the algorithm.*

What is a Problem's Condition Number?

Example 10: Find the condition number for $f(x) = \sqrt{x}$ at $x = 100$

We have $f(x) = \sqrt{x}$ and $f'(x) = \frac{1}{2\sqrt{x}}$.
The condition number κ is:

$$\begin{aligned}\kappa = \left|\frac{xf'(x)}{f(x)}\right| &= \left|\frac{100 \cdot \frac{1}{2\sqrt{100}}}{\sqrt{100}}\right| \\ &= \left|\frac{100 \cdot \frac{1}{2 \cdot 10}}{10}\right| \\ &= \left|\frac{100 \cdot \frac{1}{20}}{10}\right| = \left|\frac{5}{10}\right| = \mathbf{0.5}\end{aligned}$$

Interpretation: Since $\kappa = 0.5 < 1$, the problem is **very well-conditioned**. A 1% relative error in x will only cause a $\approx 0.5\%$ relative error in $\sqrt{x}$.

Conditioning vs. Stability

These two concepts are the most important in numerical analysis:

- *Conditioning* is a property of the **problem**.
- *Stability* is a property of the **algorithm**.

An algorithm is **numerically stable** if it produces an accurate result for **well-conditioned problems**.

Problem	Algorithm	Result
Well-conditioned	Stable	Accurate
Well-conditioned	Unstable	Inaccurate
Ill-conditioned	Stable	Limited accuracy
Ill-conditioned	Unstable	Poor accuracy

→ *Even a perfectly stable algorithm cannot overcome an ill-conditioned problem!*

Guidelines I: Code-Level Best Practices

Code-Level Best Practices

- **Avoid catastrophic cancellation:**
  - Rewrite formulas to avoid subtracting nearly equal numbers.
  - Ex: $\sqrt{a} - \sqrt{b} \to \frac{a-b}{\sqrt{a}+\sqrt{b}}$ (if $a \approx b$)
- **Avoid swamping (absorption):**
  - Avoid adding/subtracting numbers of vastly different magnitudes.
  - If summing many numbers, sort them from smallest to largest first.
- **Avoid division by small numbers:**
  - Can lead to overflow and large intermediate error.
  - Reformulate to multiply by reciprocals if safe.

Guidelines II: Design-Level Best Practices

Design-Level Best Practices

- **Use stable algorithms:**
  - Don't invent your own; use standard libraries (e.g., NumPy, SciPy).
  - Ex: For solving $Ax = b$, use LU decomposition with partial pivoting.
- **Be aware of problem conditioning:**
  - Check condition numbers if possible.
  - An ill-conditioned problem may require higher precision.
- **Validate results:**
  - Use residual checks (e.g., $b - Ax$) to see if the answer is plausible.
  - Test your code against problems with known solutions.

Summary and Takeaways

Key Takeaways

- **Floating-point (FP)** representation has inherent limitations.
- **IEEE 754** is the standard for FP formats.
- **Machine epsilon** (ϵ_{mach}) defines the maximum relative rounding error.
- **Rounding errors** are unavoidable and accumulate.
- **Catastrophic cancellation** (subtracting nearly equal numbers) is a primary source of error.
- **Condition number** measures problem sensitivity.
- **Stability** measures algorithm quality.
- **Backward error analysis** is a tool to analyze stability.

Exercise 1: Machine Epsilon

Exercise 1: Compute Machine Epsilon

Write a Python program to compute the machine epsilon for your system. Compare it with the theoretical value.
Hint: Start with $\epsilon = 1.0$ and repeatedly divide by 2.0. The last value of ϵ **before** $1.0 + \epsilon$ is computationally equal to 1.0 is your answer.

Click here to download and test the solution

Exercise 2: Catastrophic Cancellation

Exercise 2: Catastrophic Cancellation of Quadratic Formula

Given $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
For $a = 1$, $b = 10^9$, and $c = 1$, write a Python program to:

1. Compute both roots using the standard formula.
2. Identify which root suffers from catastrophic cancellation.
3. Derive an alternative, stable formula for that root.
4. Compute the roots again using the improved formula.

Click here to download and test the solution

Exercise 3: Condition Number

Exercise 3: Condition Number for $\tan(x)$

Consider the function $f(x) = \tan(x)$ near $x = \frac{\pi}{2}$. Write a Python program to:

1. Compute the condition number at $x = 1.57$ (which is close to $\frac{\pi}{2}$).
2. Is this problem well-conditioned or ill-conditioned?
3. What are the practical implications of trying to compute $\tan(x)$ near its asymptote?

Click here to download and test the solution

End of Chapter 4