Week #2: Computer Arithmetic and Error Analysis

Università della Svizzera Italiana

Francisco Richter Mendoza
Faculty of Informatics, Lugano, Switzerland

September 10, 2025

Overview

Computer arithmetic forms the foundation of all numerical computations, yet its finite precision introduces subtle complexities that can dramatically affect algorithm behavior. This week we explore the mathematical principles of floating-point representation and analyze how roundoff errors arise, propagate, and can be mitigated through careful algorithm design.

The gap between continuous mathematics and discrete computation creates fundamental challenges that cannot be eliminated, only managed. We develop rigorous error bounds that quantify the limitations of finite-precision arithmetic and provide theoretical guarantees for numerical algorithms. These insights guide the development of numerically stable methods that maintain accuracy despite the inherent limitations of computer arithmetic.

Our approach emphasizes both theoretical understanding and practical implications, connecting abstract mathematical concepts to concrete computational considerations that affect all numerical methods.

---

1 Floating-Point Number Systems

1.1 Mathematical Representation

Definition. Floating-Point Number System

A floating-point number system $F(\beta, p, L, U)$ is characterized by:

- Base β (typically 2)
- Precision p (number of significant digits)
- Exponent range $[L, U]$

Numbers in this system have the form:

$$\pm d_0.d_1 d_2 \dots d_{p-1} \times \beta^e$$

where $d_0 \neq 0$ (normalized form), $0 \leq d_i < \beta$, and $L \leq e \leq U$.

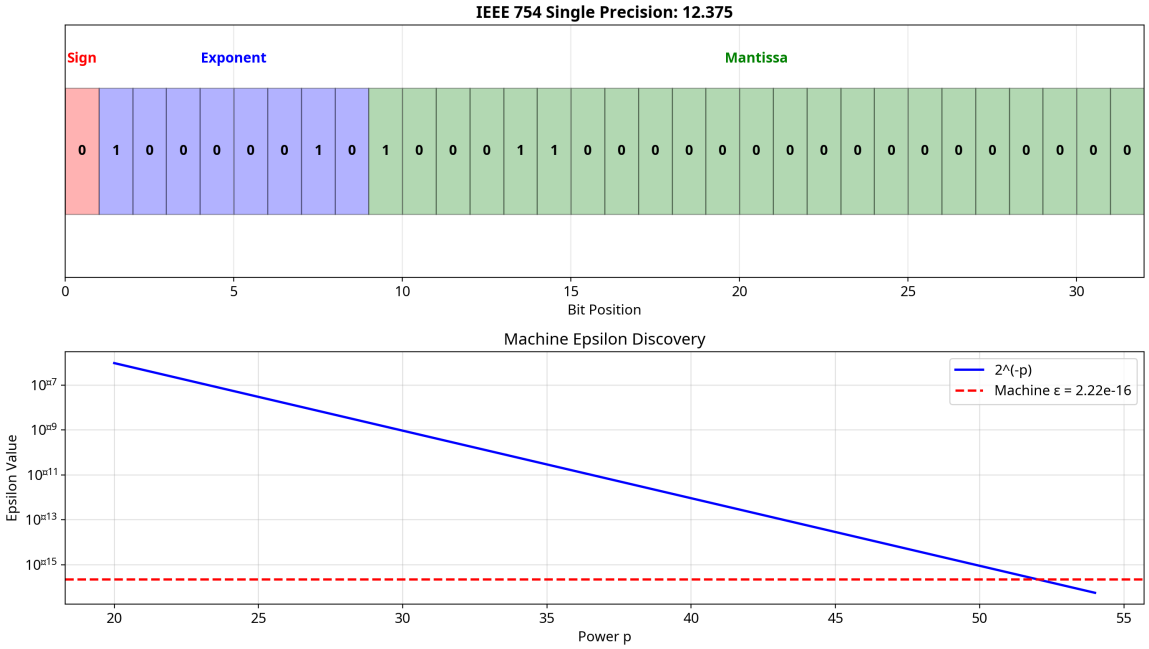

Figure 1: IEEE 754 floating-point representation and machine epsilon discovery. The top panel shows the bit layout for single precision (32-bit) format with sign, exponent, and mantissa fields clearly annotated. The bottom panel demonstrates machine epsilon as the transition point where $1 + \varepsilon > 1$ in floating-point arithmetic, showing the fundamental precision limit of computer arithmetic.

Theorem 1.1 (Properties of Floating-Point Systems)**.** *For a floating-point system $F(\beta, p, L, U)$:*

1. *The smallest positive normalized number is β^L*
2. *The largest finite number is $(\beta - \beta^{-p+1}) \times \beta^U$*
3. *The spacing between consecutive numbers near x is approximately $\beta^{1-p}|x|$*
4. *The total number of representable finite numbers is $2(\beta - 1)\beta^{p-1}(\beta^{U-L+1} - 1) + 1$*

1.2 IEEE 754 Standard

Definition. IEEE 754 Standard

The IEEE 754 standard defines:

- **Single precision (32-bit):** 1 sign bit, 8 exponent bits, 23 mantissa bits
- **Double precision (64-bit):** 1 sign bit, 11 exponent bits, 52 mantissa bits
- **Special values:** ± 0, $\pm\infty$, NaN (Not a Number)
- **Rounding modes:** Round to nearest (default), round toward 0, round toward $+\infty$, round toward $-\infty$

Example 1.1 (IEEE 754 Representation)**.** *For single precision (32-bit):*

- *Exponent bias: 127*
- *Exponent range:* $[-126, 127]$
- *Machine epsilon:* $\varepsilon_{mach} = 2^{-23} \approx 1.19 \times 10^{-7}$
- *Smallest positive normalized number:* $2^{-126} \approx 1.18 \times 10^{-38}$

- *Largest finite number:* $(2 - 2^{-23}) \times 2^{127} \approx 3.40 \times 10^{38}$

Theorem 1.2 (IEEE 754 Guarantees)**.** *The IEEE 754 standard guarantees:*

1. *If x is a real number in the representable range, and $fl(x)$ is its floating-point representation, then:*
$$fl(x) = x(1 + \delta), \quad |\delta| \leq \varepsilon_{mach}$$
2. *For basic arithmetic operations $\circ \in \{+, -, \times, \div\}$:*
$$fl(x \circ y) = (x \circ y)(1 + \delta), \quad |\delta| \leq \varepsilon_{mach}$$
3. *Gradual underflow through denormalized numbers*
4. *Consistent handling of special cases*

Intuition: Understanding floating-point precision

Floating-point numbers have non-uniform spacing throughout the real line:

- Near zero, numbers are closely spaced (spacing $\approx 2^{-149}$ for single precision)
- As magnitude increases, spacing increases proportionally (spacing near x is $\approx \varepsilon_{mach} \cdot |x|$)
- This relative precision is what makes floating-point arithmetic useful for scientific computation
- The "machine epsilon" represents the fundamental limit of relative precision

Think of floating-point numbers as having a fixed number of significant digits, regardless of magnitude.

---

2 Roundoff Error Analysis

2.1 Basic Error Bounds

Definition. Machine Epsilon

The machine epsilon ε_{mach} is the smallest positive number such that $1 + \varepsilon_{mach} > 1$ in floating-point arithmetic.

For IEEE 754:

- Single precision: $\varepsilon_{mach} = 2^{-23} \approx 1.19 \times 10^{-7}$
- Double precision: $\varepsilon_{mach} = 2^{-52} \approx 2.22 \times 10^{-16}$

Theorem 2.1 (Standard Model of Floating-Point Arithmetic)**.** *For floating-point numbers x and y and operation $\circ \in \{+, -, \times, \div\}$:*
$$fl(x \circ y) = (x \circ y)(1 + \delta), \quad |\delta| \leq \varepsilon_{mach}$$
This model assumes that the exact result of the operation is within the representable range.

Theorem 2.2 (Error Accumulation in Summation)**.** *For the floating-point summation of n numbers $s_n = \sum_{i=1}^{n} x_i$:*
$$|fl(s_n) - s_n| \leq \gamma_n \sum_{i=1}^{n} |x_i|$$
where $\gamma_n = \frac{n\varepsilon_{mach}}{1 - n\varepsilon_{mach}}$ for $n\varepsilon_{mach} < 1$.

Proof sketch. By induction on n. For $n = 2$:

$$fl(x_1 + x_2) = (x_1 + x_2)(1 + \delta_1), \quad |\delta_1| \leq \varepsilon_{mach}$$

For $n > 2$, assuming the result holds for $n - 1$:

$$fl(s_n) = fl(fl(s_{n-1}) + x_n) = (fl(s_{n-1}) + x_n)(1 + \delta_n)$$

Substituting the induction hypothesis and simplifying gives the result. □

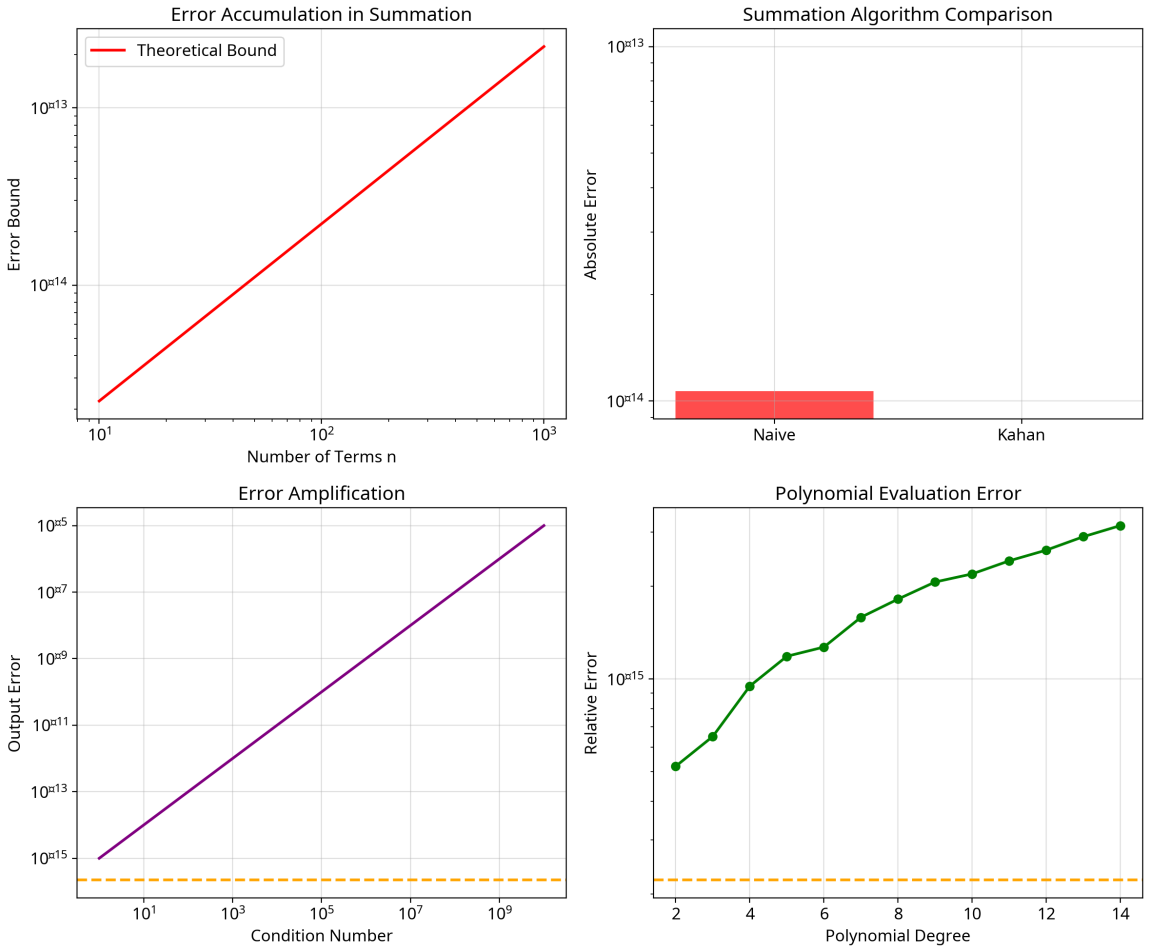

Figure 2: Roundoff error accumulation analysis. Top left: theoretical error bounds for summation grow linearly with the number of terms. Top right: Kahan summation algorithm significantly reduces accumulated error compared to naive summation. Bottom left: condition number amplifies input errors, with well-conditioned problems ($kappa < 10^6$) maintaining accuracy while ill-conditioned problems ($kappa > 10^{10}$) suffer severe error amplification. Bottom right: polynomial evaluation using Horner's method shows linear error growth with degree, demonstrating the importance of stable algorithms.

2.2 Catastrophic Cancellation

Definition. Catastrophic Cancellation

Catastrophic cancellation occurs when subtracting nearly equal floating-point numbers, resulting in a significant loss of precision.

If x and y have p significant digits and $|x-y| \ll |x|$, then $x-y$ may have far fewer than p significant digits.

Example 2.1 (Catastrophic Cancellation). *Consider computing* $\sqrt{x+1}-1$ *for small* x*:*

Direct computation: *For* $x = 10^{-16}$ *in double precision:*

$$\sqrt{1+10^{-16}} \approx 1.0000000000000000 \tag{1}$$

$$\sqrt{1+10^{-16}} - 1 \approx 0.0000000000000000 \tag{2}$$

All significant digits are lost.

Algebraic reformulation:

$$\sqrt{x+1}-1 = \frac{(\sqrt{x+1}-1)(\sqrt{x+1}+1)}{\sqrt{x+1}+1} \tag{3}$$

$$= \frac{x}{\sqrt{x+1}+1} \tag{4}$$

For $x = 10^{-16}$*:*

$$\frac{10^{-16}}{1+1} = 5 \times 10^{-17} \tag{5}$$

This preserves accuracy.

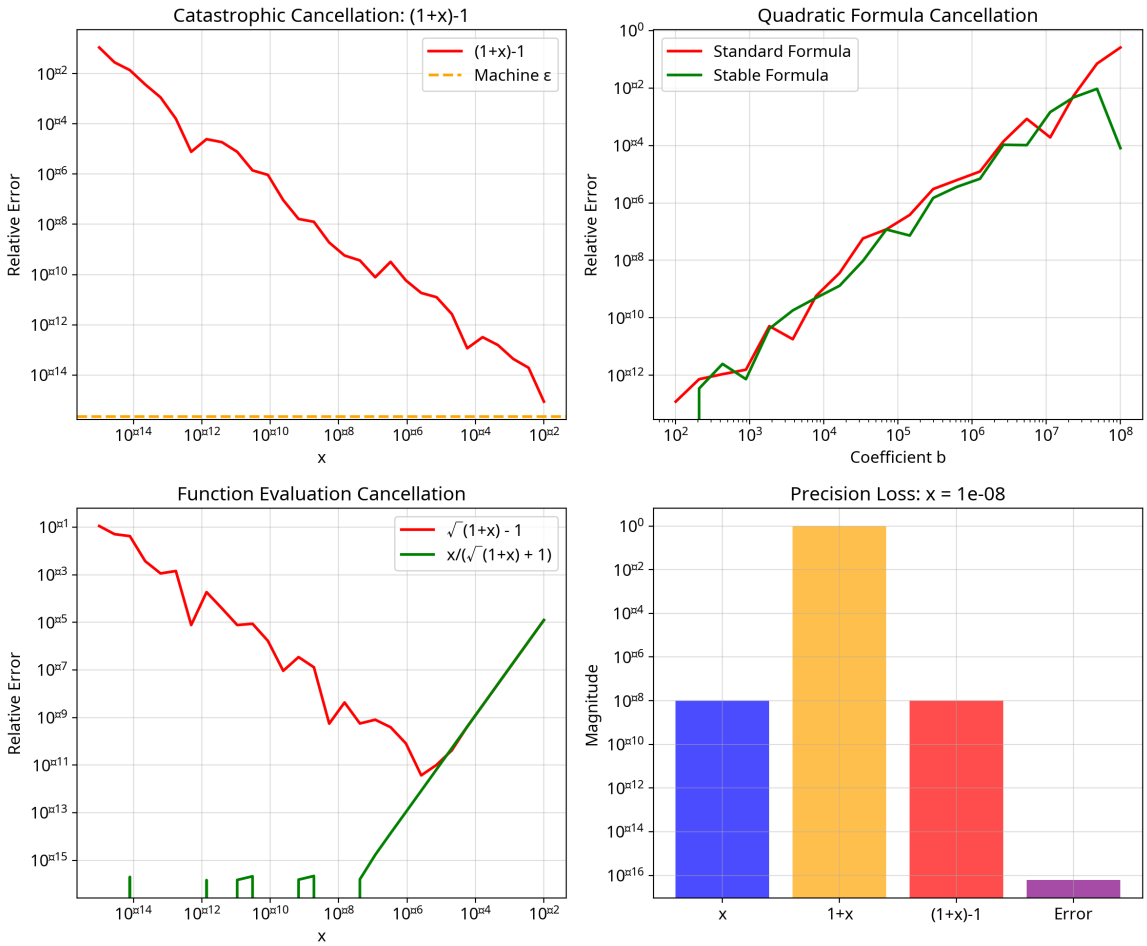

Figure 3: Catastrophic cancellation demonstrations. Top left: computing $(1+x)-1$ for small x shows dramatic precision loss as x decreases. Top right: quadratic formula comparison shows the standard formula becomes unstable for large b coefficients, while the numerically stable alternative maintains accuracy. Bottom left: function evaluation of $\sqrt{1+x}-1$ demonstrates how algebraic reformulation can eliminate cancellation errors. Bottom right: precision loss visualization shows how intermediate computations can destroy significant digits.

Theorem 2.3 (Cancellation Error Bound)**.** *When computing $x - y$ where x and y are approximate values with relative errors ϵ_x and ϵ_y:*

$$\frac{|(x-y) - (x_{exact} - y_{exact})|}{|x_{exact} - y_{exact}|} \approx \frac{|x\epsilon_x - y\epsilon_y|}{|x-y|}$$

If $x \approx y$ and $\epsilon_x \approx \epsilon_y \approx \epsilon$, this becomes:

$$\frac{|x-y|_{error}}{|x-y|} \approx \frac{|x|\epsilon}{|x-y|} \gg \epsilon$$

Intuition: Why cancellation is catastrophic

Floating-point numbers store a fixed number of significant digits, with the decimal point "floating" to represent a wide range of magnitudes.
When subtracting nearly equal numbers:

- The leading digits cancel out
- The result depends on the least significant digits
- These least significant digits contain the most roundoff error
- The relative error in the result can be magnified dramatically

This is why algorithms should be designed to avoid subtracting nearly equal quantities whenever possible.

---

3 Numerical Stability Analysis

3.1 Forward and Backward Error

Definition. Forward and Backward Error in Floating-Point

For a mathematical operation f and its floating-point implementation $fl(f)$:
Forward Error: $|f(x) - fl(f)(x)|$
Backward Error: $\min\{|\Delta x| : f(x + \Delta x) = fl(f)(x)\}$

Theorem 3.1 (Backward Error Analysis)**.** *For basic floating-point operations:*

- *Addition/subtraction: $fl(x \pm y) = (x \pm y)(1 + \delta)$ for some $|\delta| \leq \varepsilon_{mach}$*
- *Multiplication: $fl(x \times y) = (x \times y)(1 + \delta)$ for some $|\delta| \leq \varepsilon_{mach}$*
- *Division: $fl(x \div y) = (x \div y)(1 + \delta)$ for some $|\delta| \leq \varepsilon_{mach}$*

These operations produce the exact result for slightly perturbed inputs.

Example 3.1 (Forward vs. Backward Error)**.** *Consider computing $y = x^2$ where $x = 3.14159$ is rounded to $\hat{x} = 3.14$.*
Forward error: $|x^2 - \hat{x}^2| = |9.86959 - 9.8596| \approx 0.01$
Backward error: *Find Δx such that $(x + \Delta x)^2 = \hat{x}^2$* $\Delta x = \hat{x} - x = 3.14 - 3.14159 \approx -0.00159$
The backward error is much smaller than the forward error.

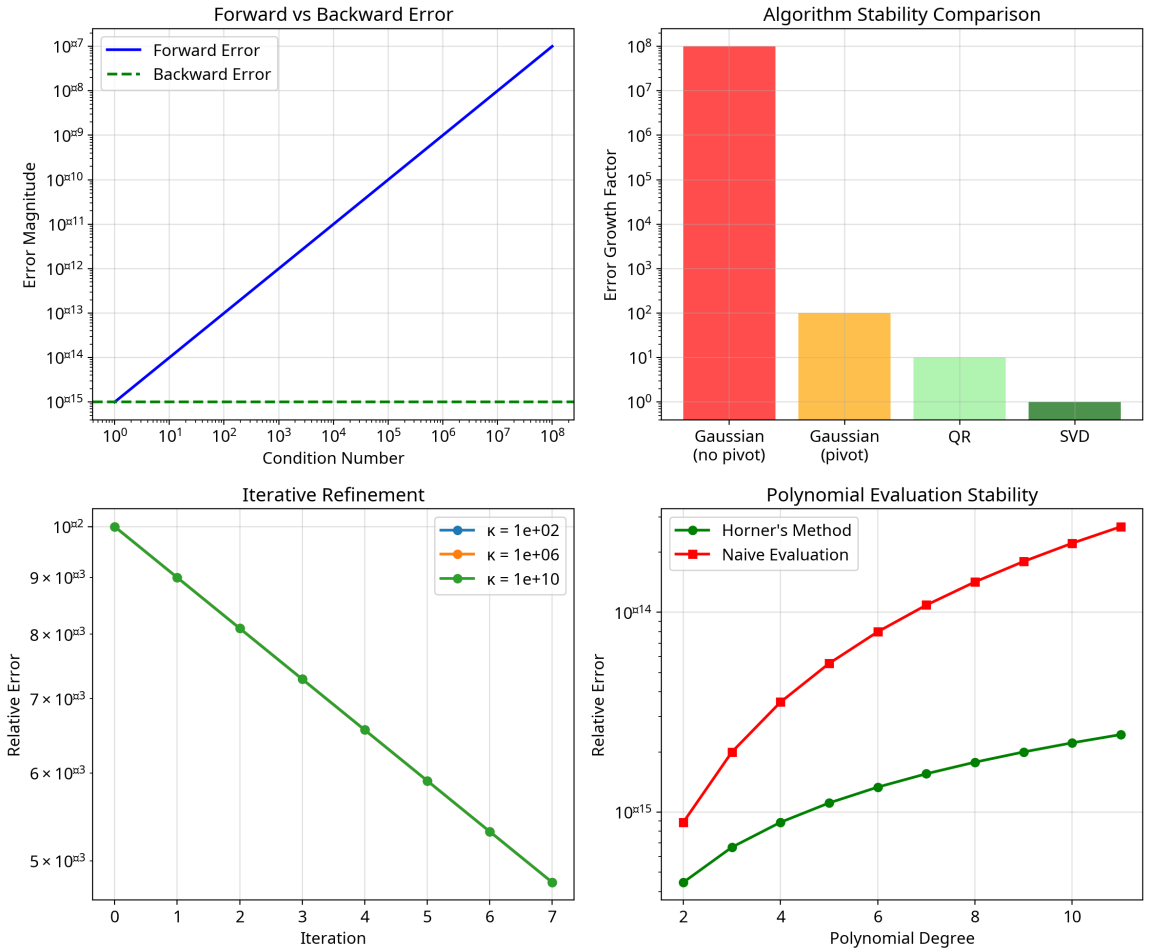

Figure 4: Numerical stability analysis. Top left: forward error amplification shows how condition numbers multiply backward errors to produce forward errors. Top right: algorithm stability comparison demonstrates that SVD and QR factorization are more stable than Gaussian elimination without pivoting. Bottom left: iterative refinement can improve accuracy for well-conditioned problems but fails for severely ill-conditioned systems. Bottom right: polynomial evaluation stability comparison shows Horner's method provides better numerical stability than naive evaluation.

3.2 Stability of Numerical Algorithms

Definition. Numerical Stability

An algorithm is **numerically stable** if the computed solution is the exact solution to a nearby problem.

More precisely, an algorithm for solving $f(x) = y$ is stable if for computed solution $\tilde{y}$, there exists $\tilde{x}$ with $\|\tilde{x} - x\| = O(\varepsilon_{mach})$ such that $f(\tilde{x}) = \tilde{y}$.

Theorem 3.2 (Stability and Condition Numbers)**.** *For a stable algorithm solving $f(x) = y$:*

$$\frac{\|\tilde{y} - y\|}{\|y\|} \leq \kappa(f) \cdot O(\varepsilon_{mach})$$

where $\kappa(f)$ is the condition number of the problem.

Example 3.2 (Algorithm Stability Comparison)**.** *For solving linear systems $Ax = b$:*

- ***Gaussian elimination (no pivoting):*** *Can be unstable, error growth $O(2^n)$*
- ***Gaussian elimination (partial pivoting):*** *Stable in practice, error growth $O(n^3)$*

- ***QR factorization:*** *Stable, error growth* $O(n)$
- ***SVD:*** *Very stable, error growth* $O(1)$

---

4 Practical Implementation Considerations

4.1 Kahan Summation Algorithm

The Kahan summation algorithm provides a method to reduce error accumulation in floating-point summation:

Listing 1: Kahan Summation Algorithm

```python
def kahan_sum(numbers):
    """Kahan summation for improved accuracy"""
    sum_val = 0.0
    c = 0.0  # Compensation for lost low-order bits

    for num in numbers:
        y = num - c         # Subtract previous compensation
        t = sum_val + y     # Add to running sum
        c = (t - sum_val) - y  # Compute new compensation
        sum_val = t         # Update sum

    return sum_val
```

Theorem 4.1 (Kahan Summation Error Bound)**.** *For Kahan summation of* n *numbers, the error bound is:*

$$|fl(s_n) - s_n| \leq 2\varepsilon_{mach} \sum_{i=1}^{n} |x_i| + O(n\varepsilon_{mach}^2)$$

This is significantly better than the $O(n\varepsilon_{mach})$ *bound for naive summation.*

4.2 Stable Quadratic Formula

For solving $ax^2 + bx + c = 0$ when $b^2 \gg 4ac$:

Listing 2: Stable Quadratic Formula Implementation

```python
import math

def stable_quadratic(a, b, c):
    """Numerically stable quadratic formula"""
    discriminant = b*b - 4*a*c

    if discriminant < 0:
        return None  # No real roots

    sqrt_disc = math.sqrt(discriminant)

    # Choose formula to avoid cancellation
    if b >= 0:
        root1 = (-b - sqrt_disc) / (2*a)
        root2 = (2*c) / (-b - sqrt_disc)
    else:
        root1 = (-b + sqrt_disc) / (2*a)
        root2 = (2*c) / (-b + sqrt_disc)

    return root1, root2
```

---

5 Educational Demonstrations

5.1 Machine Epsilon Discovery

Listing 3: Machine Epsilon Discovery

```python
def discover_machine_epsilon():
    """Discover machine epsilon experimentally"""
    eps = 1.0
    while 1.0 + eps/2 > 1.0:
        eps = eps / 2

    print(f"Discovered machine epsilon: {eps}")
    print(f"NumPy machine epsilon: {np.finfo(float).eps}")
    print(f"1 + eps/2 == 1: {1.0 + eps/2 == 1.0}")
    print(f"1 + eps == 1: {1.0 + eps == 1.0}")

    return eps
```

5.2 Catastrophic Cancellation Example

Listing 4: Catastrophic Cancellation Demonstration

```python
def cancellation_example():
    """Demonstrate catastrophic cancellation"""
    x = 1e-15

    # Unstable computation
    unstable = (1 + x) - 1

    # Stable computation
    stable = x

    print(f"x = {x}")
    print(f"Unstable (1+x)-1 = {unstable}")
    print(f"Stable x = {stable}")
    print(f"Relative error: {abs(unstable - stable) / stable:.2e}")
```

---

6 Practice Problems

Practice Problems

1. **IEEE 754 Analysis:** Convert the decimal number 12.375 to IEEE 754 single precision format. Show the binary representation and verify the conversion.
2. **Error Accumulation:** Analyze the error accumulation when computing $\sum_{i=1}^{1000} \frac{1}{i}$ using floating-point arithmetic. Compare forward and backward summation.
3. **Catastrophic Cancellation:** For the function $f(x) = \frac{\sin(x)-x}{x^3}$ near $x = 0$, derive a numerically stable implementation using Taylor series.
4. **Condition Number Analysis:** For the matrix $A = \begin{pmatrix} 1 & 1 \\ 1 & 1+\varepsilon \end{pmatrix}$ where ε is small, compute the condition number and analyze how errors in b affect the solution of $Ax = b$.
5. **Algorithm Stability:** Compare the numerical stability of computing $e^x - 1$ for small x using direct evaluation versus the mathematically equivalent form $\frac{e^x-1}{1} \cdot \frac{e^x+1}{e^x+1} = \frac{e^{2x}-1}{e^x+1}$.