COM701 - Numerical Mathematics and Computing

Chapter 1 - Numerical Precision, Error, and Efficient Computation

Dr. Mahdi Khemakhem

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

AY - 2025/2026

Outline

Outline

Significant Digits of Precision (1/7)

Significant digits are the digits in a number that carry meaningful information about its precision. They start from the first **nonzero digit** on the left and include all digits up to and including the last **reliably known digit**, even if that includes trailing zeros that are known to be exact.

Counting Significant Digits

- 4500 → *ambiguou* (**2, 3, or 4 digits**) depending on context → *Trailing zeros (00) may or may not be significant*
  - *Estimated population:* "The village has 4500 residents." → **2 digits**
  - *Rounded measurement:* "Weight = 4500 kg (nearest 10 kg)." → **3 digits**
  - *Exact volume:* "Dispensed = 4500 mL." → **4 digits**

  → Use scientific notation: 4.500×10^3 for 4 significant digits
- `4500.` → **4 significant digits** → *Trailing zeros are significant due to decimal point*
- `4.500` → **4 significant digits** (trailing zero is significant) → *Trailing zero to the right of the decimal is always significant*
- `0.00456` → **3 significant digits** (4, 5, 6) → *Leading zeros (0.00) are not significant*
- `0.0700` → **3 significant digits** (7, 0, 0) → *Leading zeros are not significant, trailing zeros are*

Significant Digits of Precision (2/7)

Example 1: Diagonal Length of a Triangle

In a machine shop, a technician cuts a 2-meter by 3-meter rectangular metal sheet into two equal right triangles.

What is the diagonal length of each triangle?

Solution

Using the **Pythagorean Theorem**, we compute the diagonal $d = \sqrt{2^2 + 3^2} = \sqrt{13} = 3.605551275$

To verify the result: $3.605551275 \times 3.605551275 = 13$.

This suggests high precision, but is the number truly that accurate? Let's investigate further.

Can we trust this level of accuracy?

If the given dimensions are only accurate to *one millimeter*, then the actual values could range between:

$$2.001^2 + 3.001^2 = \sqrt{13.01002} \approx 3.6069$$

and

$$1.999^2 + 2.999^2 = \sqrt{12.990002} \approx 3.6042$$

Thus, the diagonal d may lie within:

$$3.6042 \leq d \leq 3.6069$$

Significant Digits of Precision (3/7)

Example 1: Diagonal Length of a Triangle (continued)

Can the pieces be trimmed so that the diagonal becomes exactly 3.6 meters?

Solution

To make the diagonal exactly 3.6 meters, we trim both dimensions equally. Let x be the amount removed from both the 2 m and 3 m sides:

$$(3-x)^2 + (2-x)^2 = 3.6^2$$

This simplifies to:

$$x^2 - 5x + 0.02 = 0$$

Using the quadratic formula with $a = 1, b = -5, c = 0.02$:

$$x = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a} = 0.00400321 \approx 0.00400$$

So, by trimming 4 mm from each side, the new triangle (1.996 m by 2.996 m) has a diagonal approximately equal to 3.6 m.

Significant Digits of Precision (4/7)

To illustrate the **impact of significant digits** in computations, let's solve a simple system of equations for the variable y using different levels of precision.

Example 2: System of Equations

$$\begin{cases} 0.1036x + 0.2122y = 0.7381 \\ 0.2081x + 0.4247y = 0.9327 \end{cases}$$

Solve for y:

- First, using *3 significant digits*
- Then, using *4 significant digits*
- Finally, using *10 significant digits*

Significant Digits of Precision (5/7)

Example 2: System of Equations

Solution (3 digits)

Round all coefficients to 3 significant digits:

$$\begin{cases} 0.104x + 0.212y = 0.738 \\ 0.208x + 0.425y = 0.933 \end{cases}$$

Take a multiple α of the first equation and subtract it from the second equation to eliminate the x-term in the second equation.

$$\alpha = \frac{0.208}{0.104} \approx 2.00$$

$$0.425 - 2.00 \times 0.212 = 0.425 - 0.424 = 0.001$$

$$0.933 - \mathbf{2.00} \times \mathbf{0.738} = 0.933 - \mathbf{1.48} = -0.547$$

$$y = \frac{-0.547}{0.001} \approx -547$$

Significant Digits of Precision (6/7)

Example 2: System of Equations

Solution (4 and 10 digits)

Using 4 significant digits:

$$\alpha = \frac{0.2081}{0.1036} \approx 2.009$$

$$0.4247 - 2.009 \times 0.2122 \approx 0.4247 - 0.4263 = -0.001600$$

$$0.9327 - 2.009 \times 0.7381 \approx 0.9327 - 1.483 = -0.5503$$

$$y = \frac{-0.5503}{-0.001600} \approx 343.9$$

With 10 digits: $y \approx 356.2907199$

This shows how a simple rounding step can flip results entirely—changing not just the value, but even the sign or the order of magnitude!

Significant Digits of Precision (7/7)

Key Takeaways

- The previous example shows that data believed to be accurate should be carried with *full precision* throughout calculations, avoiding *premature rounding*.
- Most computers use a *double-length accumulator* to improve internal precision—but even this cannot fully prevent accuracy loss.
- *Loss of accuracy* may occur due to:
  - **Roundoff errors**
  - **Subtracting nearly equal numbers**

Outline

Accuracy and Precision (1/3)

- **Accurate to *n* decimal places:** This means you can rely on *n* digits to the right of the decimal point. Example: 12.345 m is accurate to *3 decimal places*.
- **Accurate to *n* significant digits:** This means you can rely on a total of *n* digits, starting from the first nonzero digit. Example: 0.076 m has only *2 significant digits*.
- Using a millimeter ruler: measurements are accurate to 0.001 m (three decimal places). A reading like 12.3456789 m is meaningless since the ruler provides only three decimals.
  $\Rightarrow$ *It should be reported as* 12.345 *m or* 12.346 *m.*

Accuracy and Precision (2/3)

Addition and Subtraction

When using a calculator or computer, one may get a false sense of precision that is not supported by the data.

Example 3: Addition and Subtraction

$1.2 + 3.45 = 4.65$ appears precise, but in reality it has only *two significant digits*, since the second digit in 1.2 may come from rounding (e.g., $1.249 \mapsto 1.2$ or $1.16 \mapsto 1.2$).
Possible values:

$$1.249 + 3.454 = 4.703$$

$$1.160 + 3.449 = 4.609$$

Thus, the sum is reliable only to **two decimal places**.

Key Takeaways

In *addition* and *subtraction*, the result is accurate only up to the smallest number of **decimal places** among the inputs.

Accuracy and Precision (3/3)

Multiplication and Division

Example 4: Multiplication and Division

$$1.2 \times 3.45 = 4.14$$

appears precise, but in reality it has only *two significant digits*, since the second digit in 1.2 may come from rounding (e.g., $1.249 \mapsto 1.2$ or $1.16 \mapsto 1.2$).
Possible values:

$$1.249 \times 3.454 = 4.312$$

$$1.160 \times 3.449 = 4.002$$

Thus, the product is reliable only to **two significant digits**.

Key Takeaways

In *multiplication* and *division*, the result is accurate only up to the smallest number of **significant digits** among the inputs.

Outline

Rounding vs. Chopping (1/3)

- *Rounding* - reduces the number of significant digits while keeping the value close to the original.
- *Chopping* - simply discards all digits beyond the chosen position without adjustment.
- Both are used to control precision in numerical computations.

Rounding vs. Chopping (2/3)

Over a large set of data, the *round-to-even* rule tends to reduce the total rounding error with (on average) an equal portion of numbers rounding up as well as rounding down.

- Also called *statistician's rounding* or *bankers' rounding*.
- When the discarded part is **exactly** 0.5, round so that the last kept digit is *even*.
- Balances rounding errors: about half up, half down.
- This is the rule commonly used in **floating-point arithmetic** on computers.

Example 5: Rounding

$$2.5 \to 2 \quad (evendigitkept)$$
$$3.5 \to 4 \quad (evendigitkept)$$

Rounding vs. Chopping (3/3)

Example 6: Rounding vs. Chopping

Rounding to 2 digits:	**Chopping to 2 digits:**
$0.217 \approx 0.22$	$0.217 \approx 0.21$
$0.365 \approx 0.36$	$0.365 \approx 0.36$
$0.475 \approx 0.48$	$0.475 \approx 0.47$
$0.592 \approx 0.59$	$0.592 \approx 0.59$

Key Takeaways

- **Chopping** always truncates $\rightarrow$ introduces a bias (error always downward).
- **Rounding** gives smaller average error across large datasets.
- **Round-to-even** minimizes systematic bias in repeated calculations.
- Most computer systems default to **round-to-even**.

Outline

Absolute and Relative Errors (1/4)

When approximating a number x by $\hat{x}$, we define:

$$\textit{Absolute Error} = |x - \hat{x}|$$

This is the magnitude of the difference between the **exact value** x and its **approximation** $\hat{x}$.

$$\textit{Relative Error} = \frac{|x - \hat{x}|}{|x|}, \quad x \neq 0$$

This measures the size of the error relative to the **exact value**.

- For **absolute error**, x and $\hat{x}$ are interchangeable.
- For **relative error**, x must represent the *exact value*.
- Note: The relative error is undefined when $x = 0$.

Absolute and Relative Errors (2/4)

In practice, the **relative error** is usually *more informative* than the **absolute error**.

Example 7: Absolute and Relative Errors

Suppose we have two approximations:

- $x_1 = 1.333$, $\hat{x}_1 = 1.334$
- $x_2 = 0.001$, $\hat{x}_2 = 0.002$

In both cases, the **absolute error** is:

$$|x_i - \hat{x}_i| = 10^{-3}$$

However, the **relative errors** are:

$$\text{For } x_1: \quad \frac{10^{-3}}{1.333} \approx \frac{3}{4} \times 10^{-3}, \quad \text{For } x_2: \quad \frac{10^{-3}}{0.001} = 1$$

- $\hat{x}_1$ is a *good approximation* to x_1 (small relative error).
- $\hat{x}_2$ is a *poor approximation* to x_2 (relative error of 1).

Absolute and Relative Errors (3/4)

Example 7: Absolute and Relative Errors

Consider the values:

- $x = 0.00347$ rounded to $\hat{x} = 0.0035$
- $y = 30.158$ rounded to $\hat{y} = 30.16$

For each case, determine:

- Number of *significant digits*
- *Absolute error* and *relative error*

Solution

Case 1: $x = 0.00347$, $\hat{x} = 0.0035 = 0.35 \times 10^{-2}$

- Significant digits: **2**
- Absolute error: $|\hat{x} - x| = |0.0035 - 0.00347| = 0.00003 = 0.3 \times 10^{-4}$
- Relative error: $\frac{0.00003}{0.00347} \approx 0.00865 = 0.865 \times 10^{-2}$

Absolute and Relative Errors (4/4)

Example 7: Absolute and Relative Errors

Solution

Case 2: $y = 30.158$, $\hat{y} = 30.16 = 0.3016 \times 10^2$

- Significant digits: **4**
- Absolute error: $|\hat{y} - y| = |30.16 - 30.158| = 0.002 = 0.2 \times 10^{-2}$
- Relative error: $\frac{0.002}{30.158} \approx 0.000066 = 0.66 \times 10^{-4}$

Key Takeaways

- Although both values have **absolute errors of the same order of magnitude**, their **relative errors** differ significantly.
- The smaller relative error for y reflects its higher number of *significant digits*.
- *Relative error* is a **more meaningful measure** of numerical accuracy than absolute error.

Outline

Nested Multiplication (1/9)

Polynomial Evaluation

We start with remarks on evaluation a polynomial efficiently and on rounding and chopping real numbers.

- **Goal**: evaluate efficiently

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{1}$$

$$= \sum_{i=0}^{n} a_i x^i = \sum_{i=0}^{n} \left(a_i \prod_{j=1}^{i} x \right) \tag{2}$$

- **Naive method**: compute powers $x^2, x^3, \ldots, x^n$ explicitly (*costly*).
- **Better method**: use *Nested Multiplication* (*Horner's Algorithm*).

Nested Multiplication (2/9)

Horner's Algorithm

$$p(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n)\cdots)) \tag{3}$$

- To evaluate $p(x)$, starts from the innermost parentheses and works outward.
- Avoids *repeated exponentiation* (repeated powers).
- Assume that numerical values have been assigned to the integer variable n, the real variable x, as well as the coefficients $a_0, a_1, \ldots, a_n$, which are stored in a real linear array.

Nested Multiplication (3/9)

Horner's Algorithm

```python
def horner(a, x):
    """
    Inputs:
    a: list of coefficients [a0, a1, ..., an]
    x: value at which to evaluate polynomial
    Outputs:
    p: polynomial p(x)= a0 + a1*x + ... + an*x^n evaluated at x
    """
    n = len(a) - 1  # degree of polynomial
    p = a[n]    # start with highest-degree coefficient
    for i in range(n-1, -1, -1):    # from n-1 down to 0
        p = a[i] + x * p
    return p
# Example usage
coeffs = [-2, -5, 7, -4, 1] # x^4 - 4*x^3 + 7*x^2 - 5*x + -2
x_val = 3
print(horner(coeffs, x_val))
```

Click here to download and test the code

Nested Multiplication (4/9)

Efficiency of Horner's Algorithm

- Each loop: exactly *1 addition + 1 multiplication*.
- Total: *n additions + n multiplications*.
- This is the *minimum possible* for polynomial evaluation!
- Naive method requires many more operations (e.g. repeated powers).

Example 8: Horner's Method

$$p(x) = 5 + 3x - 7x^2 + 2x^3$$

Using Horner's method, we can rewrite it as

$$p(x) = 5 + x(3 + x(-7 + x(2)))$$

Nested Multiplication (5/9)

Horner's algorithm can also be used for *polynomial deflation*[1] — the process of removing a **linear factor** from a polynomial. If r is a **root**[2] of a polynomial $p(x)$, then $(x - r)$ is a factor of p, and the polynomial can be written as:

$$p(x) = (x - r) \times q(x) + p(r) \tag{4}$$

where $q(x)$ is the **quotient polynomial** of degree one less than $p(x)$, and:

$$q(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1} \tag{5}$$

- The remainder of the division, b_{-1}, is equal to $p(r)$.
- If r is an **exact root**[3], then $p(r) = 0$, and $p(x)$ is exactly divisible by $(x - r)$.

[1] Polynomial deflation is the process of dividing a polynomial by one of its known factors, typically of the form $(x - r)$, where r is a known root (zero) of the polynomial.

[2] A *root* (or zero) of a polynomial $p(x)$ is a value r such that $p(r) = 0$.

[3] An *exact root* is a true mathematical root r with $p(r) = 0$, as opposed to an *approximate root*, which is a numerically computed value $\tilde{r}$ that only makes $p(\tilde{r})$ close to zero.

Nested Multiplication (6/9)

When carried out manually (by hand), Horner's method is often arranged as:

$r)$	a_n	a_{n-1}	a_{n-2}	$\cdots$	a_1	a_0	$+$
	$\downarrow$	$r \times b_{n-1}$	$r \times b_{n-2}$	$\cdots$	$r \times b_1$	$r \times b_0$	
	b_{n-1}	b_{n-2}	b_{n-3}	$\cdots$	b_0	b_{-1}	

Note: This table shows how each coefficient b_i is generated using the recurrence in Horner's method.

Example 9: Evaluating a Polynomial

Use Horner's algorithm to evaluate $p(3)$, where p is the polynomial $p(x) = x^4 - 4x^3 + 7x^2 - 5x - 2$
We arrange the calculation as suggested above:

$3)$	1	-4	7	-5	-2
	$\downarrow$	3	-3	12	21
	1	-1	4	7	$\mathbf{19}$

Thus, we obtain $p(3) = 19$, and we can write $p(x) = (x - 3)(x^3 - x^2 + 4x + 7) + 19$

Nested Multiplication (7/9)

The code for *Horner's algorithm* for *polynomial deflation* is as follows:

```python
def horner_deflation(a, r):
    """
    Inputs:
    a: list of coefficients [a0, a1, ..., an] of polynomial p(x)
    r: root of p(x), i.e., p(r)=0
    Outputs:
    b: list of coefficients [b0, b1, ..., b_{n-1}] of deflated polynomial q(x)
           where p(x) = (x - r) * q(x) + remainder
    remainder: value of p(r), should be close to 0 if r is a root
    """
    n = len(a) - 1  # degree of polynomial
    b = [0] * n     # initialize list for coefficients of deflated polynomial
    b[n-1] = a[n]  # leading coefficient remains the same
    for i in range(n-2, -1, -1):  # from n-2 down to 0
        b[i] = a[i+1] + r * b[i+1]
    remainder = a[0] + r * b[0]
    return b, remainder
```

Click here to download and test the code

Nested Multiplication (8/9)

Example 10:

Use Horner's algorithm to evaluate $p(2)$, where p is the polynomial $p(x) = x^4 - 4x^3 + 7x^2 - 5x - 2$. Also, perform polynomial deflation using the root $r = 2$.

Solution

We use the same arrangement of computations as explained previously:

2)	1	-4	7	-5	-2
	$\downarrow$	2	-4	6	2
	1	-2	3	1	**0**

Thus, we have $p(2) = 0$, and we can write $p(x) = x^4 - 4x^3 + 7x^2 - 5x - 2 = (x-2)(x^3 - 2x^2 + 3x + 1)$

Nested Multiplication (9/9)

Key Takeaways

- Nested multiplication = Horner's Algorithm.
- Uses only n additions and n multiplications.
- Eliminates exponentiation $\rightarrow$ faster, more stable.
- Useful for both evaluation and polynomial deflation.

Outline

Pairs of Easy/Hard Problems (1/4)

In **scientific computing**, we often encounter *pairs of problems* where:

- One direction is *easy* or *well-Conditioned* to compute.
- The inverse direction is *hard* or *ill-conditioned*.

This is a key concept across multiple areas of computational mathematics — including *cryptology*, *polynomials*, *linear systems*, *differential equations*, and *eigenvalue problems*[4].

Example 11: Cryptology

- **Easy Problem:** Multiply two large integers.
- **Hard Problem:** Factor a large number into primes.

Note: Public-key cryptography relies on this asymmetry:

Easy to compute: $N = p \times q$ *Hard to reverse:* $N \Rightarrow p, q$

[4] It arises whenever we want to find special scalars and vectors associated with a linear transformation.

Pairs of Easy/Hard Problems (2/4)

Example 12: Polynomials

- **Easy Problem:** Given the roots $r_1, r_2, \ldots, r_n$, construct the polynomial:

$$p(x) = (x - r_1)(x - r_2) \cdots (x - r_n)$$

- **Hard Problem:** Given the power form of $p(x)$, compute its roots.

Note: The reverse problem is often **ill-conditioned**, especially for high-degree polynomials.

Example 13: Linear Systems

- **Easy Problem:** Given matrix A and vector x, compute $b = Ax$.
- **Hard Problem:** Given A and b, solve for x in $Ax = b$.

Note: Matrix inversion is computationally intensive and sensitive to small changes in A.

Example 14: Boundary Value Problems

- **Easy Problem:** Given a function f, compute its derivative Df, and evaluate $f(0)$, $f(1)$.
- **Hard Problem:** Given Df, $f(0)$, and $f(1)$, find f.

Note: This is a classic example of an **inverse problem** in differential equations.

Pairs of Easy/Hard Problems (3/4)

Example 15: Eigenvalue Problems

- **Easy Problem (Reconstruction):** Suppose we already know the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ of an $n \times n$ matrix A, and the corresponding eigenvectors $v_1, v_2, \ldots, v_n$.

$$D = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}, V = \begin{bmatrix} | & | & & | \\ v_1 & v_2 & \cdots & v_n \\ | & | & & | \end{bmatrix}$$

  Then the matrix A can be reconstructed as $A\,V = V\,D \quad \Rightarrow \quad A = VDV^{-1}$.

- **Hard Problem (Computation):** Given a matrix A, determine all eigenvalues λ_i and eigenvectors v_i.

Note: Finding eigenvalues from A is computationally challenging and very sensitive to small changes (perturbations) in A.

Pairs of Easy/Hard Problems (4/4)

Key Takeaways

- Many problems in scientific computing exhibit this *asymmetry*.
- Forward computations are usually stable and efficient.
- Inverse problems tend to be sensitive, ill-conditioned, or computationally expensive.
- Recognizing this helps in choosing the right numerical methods and setting realistic expectations.

Outline

Experiment 1

Experiments 1: Finding Relative Error

Find the relative error involved in rounding 4.9997 to 5.000?

Solution

$$\text{Relative error} = \frac{|\text{approx} - \text{exact}|}{|\text{exact}|} = \frac{|5.000 - 4.9997|}{4.9997} = \frac{0.0003}{4.9997} \approx 6.0 \times 10^{-5}$$

Python check:

```
exact = 4.9997
approx = 5.0
rel_error = abs(approx - exact) / abs(exact)
print(rel_error)
```

Click here to download and test the code

Experiment 2

Experiment 2: Finding x from Relative Error

A real number x is represented approximately by 0.6032, and we are told that the relative error is at most 0.1%. Find the range in which x lies.

Solution

$$\text{Relative error} = \frac{|x - 0.6032|}{|x|} \leq 0.001 \Rightarrow -0.001 \leq \frac{x - 0.6032}{x} \leq 0.001 \Rightarrow$$

$$-0.001 \leq 1 - \frac{0.6032}{x} \leq 0.001 \Rightarrow -1.001 \leq -\frac{0.6032}{x} \leq -0.999 \Rightarrow 0.999x \leq 0.6032 \leq 1.001x \Rightarrow$$

$$0.6032/1.001 \leq x \leq 0.6032/0.999 \Rightarrow 0.6026 \leq x \leq 0.6038$$

Python check:

```
approx = 0.6032
rel_err = 0.001
x_min = approx / (1 + rel_err)
x_max = approx / (1 - rel_err)
print(x_min, x_max)
```

Click here to download and test the code

Experiment 3

Experiment 3: Investigating Rational Approximations of π

It is commonly believed that $\frac{22}{7}$ is a good approximation of π. In this experiment, we will investigate the quality of this approximation and explore other rational approximations of π.

- Task 1: Show that $\frac{355}{113}$ is a significantly better approximation than $\frac{22}{7}$ by computing their absolute and relative errors with respect to the true value of π.
- Task 2: Find other rational approximations $\frac{n}{m}$ such that the absolute error $|\pi - \frac{n}{m}|$ is less than 10^{-9}.

Hint: You may find it helpful to use continued fractions to find good rational approximations of π.
Notice: You may solve this assignment by hand or using Python — however, using Python is highly recommended. If you choose Python, include your code, output, and a brief analysis.

Experiment 3

Experiment 3: Investigating Rational Approximations of π

Solution of Task 1

```python
import math
pi_true = math.pi # True value of pi
approx_1 = 22 / 7 # First approximation
approx_2 = 355 / 113 # Second approximation
def show_errors(label, value):
    abs_error = abs(value - pi_true) # Absolute error
    rel_error = abs_error / pi_true # Relative error
    print(f"{label:<8} | Value: {value:.12f} | Abs Error: {abs_error
        :.2e} | Rel Error: {rel_error:.2e}") # Print errors
show_errors("22/7", approx_1) # Show errors for first approximation
show_errors("355/113", approx_2) # Show errors for second
    approximation
```

Click here to download and test the code of Task 1

Experiment 3

Experiment 3: Investigating Rational Approximations of π

Solution of Task 2

```python
import math
from fractions import Fraction
pi_true = math.pi # True value of pi
threshold = 1e-9 # Error threshold
max_denominator = 100000 # Maximum denominator
count = 0 # Count of good approximations
for m in range(1, max_denominator): # Iterate over denominators
    n = round(pi_true * m) # Approximate numerator
    frac = Fraction(n, m) # Create fraction
    error = abs(float(frac) - pi_true) # Calculate error
    if error < threshold: # Check if error is within threshold
        print(f"n={n}, m={m}, n/m={float(frac):.12f}, Abs Error={
            error:.2e}")
        count += 1
        if count >= 5:  # Show only the first 5 results
            break
```

Click here to download and test the code of Task 2

Experiment 4

Experiment 4: Array Summation Patterns and Efficiency Analysis

Objective: Analyze different summation patterns for a doubly subscripted array and evaluate their computational efficiency. Given a doubly subscripted array $(a_{ij})_{n\times n}$, the sum of all elements can be computed in various orders. While mathematically equivalent due to the commutative property of addition, different summation patterns may have varying computational characteristics in practice.

Mathematical summation patterns to implement:

$$\textbf{a.}\ \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} \qquad \textbf{b.}\ \sum_{j=1}^{n}\sum_{i=1}^{n} a_{ij} \qquad \textbf{c.}\ \sum_{i=1}^{n}\left(\sum_{j=1}^{i} a_{ij} + \sum_{j=1}^{i-1} a_{ji}\right)$$
$$\textbf{d.}\ \sum_{k=0}^{n-1}\sum_{|i-j|=k} a_{ij} \qquad \textbf{e.}\ \sum_{k=2}^{2n}\sum_{i+j=k}^{n} a_{ij}$$

In **computer science**, the order of array traversal can affect **cache performance**, **memory access patterns**, and overall *efficiency*. Row-major order (common in languages like C and Python) typically favors row-wise traversal for better cache utilization. **Note:** For large matrices, the choice of summation pattern can significantly impact performance due to memory hierarchy considerations and potential cache misses.

$\Rightarrow$ *The following pseudocode segments implement these summation patterns for a $n \times n$ matrix.*

Experiment 4

Experiment 4: Array Summation Patterns and Efficiency Analysis

Pseudocode Solutions

a. Row-major order summation

```
sum = 0
for i from 1 to n:
    for j from 1 to n:
        sum += a[i][j]
```

b. Column-major order summation

```
sum = 0
for j from 1 to n:
    for i from 1 to n:
        sum += a[i][j]
```

Experiment 4

Experiment 4: Array Summation Patterns and Efficiency Analysis

Pseudocode Solutions (continued)

c. Diagonal-aware summation

```
sum = 0
for i from 1 to n:
    for j from 1 to i:          # Lower triangular including diagonal
        sum += a[i][j]
    for j from 1 to i-1:        # Upper triangular excluding diagonal
        sum += a[j][i]
```

d. Summation by distance from main diagonal

```
sum = 0
for k from 0 to n-1:
    for i from 1 to n:
        for j from 1 to n:
            if |i-j| == k:
                sum += a[i][j]
```

Experiment 4

Experiment 4: Array Summation Patterns and Efficiency Analysis

Pseudocode Solutions (continued)

e. Summation by constant i+j

```
sum = 0
for k from 2 to 2n:
    for i from 1 to n:
        for j from 1 to n:
            if i+j == k:
                sum += a[i][j]
```

Best Choice? To evaluate the performance of these patterns, we can implement them in code and measure their execution times on large matrices.

Click here to download and test the code that compares the efficiency of these patterns

Experiment 5

Experiment 5: Investigating Numerical Derivatives

Objective: Approximate the derivative of a function using finite differences on a computer. From calculus, the derivative of a function $f(x)$ is defined as:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

However, in **practice**, we often work with **discrete data points** rather than *continuous functions*. Therefore, we approximate the derivative using **finite differences**. A computer can *imitate* the limit by using a *sequence of small numbers* h, for example:

$$h = \frac{1}{4}, \frac{1}{4^2}, \frac{1}{4^3}, \ldots, \frac{1}{4^n}, \ldots$$

which *approach zero rapidly*. Other possible sequences include $h = 1/n, 1/n^2$, or $1/10^n$.

Note: On a 32-bit computer, the sequence $h = 1/4^n$ reaches machine-precision levels around $n \approx 27$.
$\Rightarrow$ *The following is a Python code to compute $f'(x)$ at the point $x = 0.5$, with $f(x) = \sin x$.*

Experiment 5

Experiment 5: Investigating Numerical Derivatives

Solution

```python
import math
x = 0.5 # Point where derivative is evaluated
n = 100 # Number of steps
h = 1.0 # Initial step size
emax = 0.0 # Maximum error
imax = 0 # Iteration at which maximum error occurs
for i in range(1, n+1): # Loop over decreasing h
    h *= 0.25   # Reduce step size
    y = (math.sin(x + h) - math.sin(x)) / h # Finite difference approximation
    error = abs(math.cos(x) - y) # True derivative is cos(x)
    if error > emax: # Update maximum error
        emax = error
        imax = i #
print("imax =", imax)
print("Maximum error =", emax)
```

Click here to download and test the code

End of Chapter 1