MT308 Numerical Analysis

Girne American University
Faculty of Engineering
Summer 2025
Dr. Kamil Dimililer

1

Computational Cost and Complexity

- **Computational cost** refers to the amount of resources required to execute a computational task
  - Time (CPU cycles, execution time)
  - Memory (RAM used during execution)
  - Disk I/O (reading/writing to storage)
  - Energy consumption
- **Computational complexity** describes how the computational cost scales with input size n.
  - Time complexity (The number of operations grows (e.g., O(n), $O(n^2)$, O(log n))
  - Space complexity (The memory use grows with input size)

2

Example: Evaluating a Polynomial

- Evaluate,

$$P(x) = 2x^4 + 3x^3 - 3x^2 + 5x - 1$$

at x=1/2.

1. Direct Method: (10 multiplication +4 summation)

$$P\left(\frac{1}{2}\right) = 2 * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} + 3 * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} - 3 * \frac{1}{2} * \frac{1}{2} + 5 * \frac{1}{2} - 1 = \frac{5}{4}$$

2. Find powers and sum each term:
   (7 multiplications and 4 additions)

$$P\left(\frac{1}{2}\right) = 2 * \left(\frac{1}{2}\right)^4 + 3 * \left(\frac{1}{2}\right)^3 - 3 * \left(\frac{1}{2}\right)^2 + 5 * \frac{1}{2} - 1 = \frac{5}{4}$$

$$\frac{1}{2} * \frac{1}{2} = \left(\frac{1}{2}\right)^2$$
$$\left(\frac{1}{2}\right)^2 * \frac{1}{2} = \left(\frac{1}{2}\right)^3$$
$$\left(\frac{1}{2}\right)^3 * \frac{1}{2} = \left(\frac{1}{2}\right)^4$$

3

Example: Evaluating a Polynomial

- Evaluate,

$$P(x) = 2x^4 + 3x^3 - 3x^2 + 5x - 1$$

at x=1/2.

3. (Nested Multiplication, Horner's Method) Rewrite the polynomial so that it can be evaluated from the inside out. (4 multiplications and 4 additions)

$$\begin{aligned} P(x) &= -1 + x(5 - 3x + 3x^2 + 2x^3) \\ &= -1 + x(5 + x(-3 + 3x + 2x^2)) \\ &= -1 + x(5 + x(-3 + x(3 + 2x))) \\ &= -1 + x * (5 + x * (-3 + x * (3 + x * 2))) \end{aligned}$$

4

Evaluating a Polynomial

	Direct Computation	Powers/Sums	Horner's Method
#sum	4	4	4
#multiplication	10	7	4

Cost efficient

Note: Computational cost of multiplication operation is 3 to 10 times higher than the summation operation.

5

Time Complexity

- Computational complexity is a way to measure how much time or memory (space) an algorithm needs to run, depending on the size of the input.
- Big O notation gives an upper bound on the growth rate of an algorithm's resource usage.

O(1) - Constant Time: The algorithm takes the same amount of time regardless of the input size (e.g., accessing an element in an array by its index).
O(log n) - Logarithmic Time: The algorithm's runtime increases logarithmically with the input size (e.g., binary search).
O(n) - Linear Time: The algorithm's runtime increases linearly with the input size (e.g., iterating through an array once).
O(n log n) - Linearithmic Time: The algorithm's runtime increases slightly faster than linear (e.g., efficient sorting algorithms like merge sort).
$O(n^2)$ - Quadratic Time: The algorithm's runtime increases quadratically with the input size (e.g., nested loops iterating through all pairs of elements in an array).
$O(2^n)$ - Exponential Time: The algorithm's runtime grows exponentially with the input size (e.g., some recursive algorithms

6

Example: Find the maximum of an array

To find the largest number,

1- Start with the first number.

2- Compare it to each of the others one by one.

3- Keep track of the biggest so far.

This algorithm checks each number once — so if there are n numbers, it makes n comparisons.

Time Complexity: O(n)

7

Error in numerical operations

In numerical analysis, errors represent the difference between the true (exact) value and the approximated (computed) value. There are various types of errors:

a. Round-off Error: Caused by the finite precision of computer arithmetic (floating-point representation). Occurs because computers can't represent most real numbers exactly. Example: Representing π as 3.14159 instead of its infinite decimal.

b. Truncation Error: Occurs when an infinite process is approximated by a finite one. Common in methods like Taylor series or numerical integration. Example: Using a finite number of terms in a series expansion.

c. Absolute Error: The difference between the true value x and the approximation x' : Absolute Error=$|x-x'|$

d. Relative Error: Measures error relative to the size of the exact value: Relative Error=$\frac{|x-x'|}{|x|}$

8

Decimal vs Binary Numbers

- **Decimal Numbers (Base-10):**
- The number system we use in everyday life.
- Uses **10 digits**: **0 to 9**.
- Each digit's place value is a power of **10**.

Example:

$345 = 3\times10^2 + 4\times10^1 + 5\times10^0$

- **Binary Numbers (Base-2):**
- Used by computers and digital electronics.
- Bit = Binary Digit
- Uses only **2 digits: 0 and 1**.
- Each digit's place value is a power of **2.**

Example:

$1011 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 =$ 11 in decimal

9

Representing numbers

Since computer hardware is built from logic gates that detect two voltage levels (high=1 and low= 0), all of the numbers are stored/processed by using their binary forms (0s and 1s). Integers can be stored as unsigned or signed formats.

Unsigned integers

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Unsigned Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Signed integers, The leftmost bit is the sign bit

2's Complement representation is used for negative values)

Decimal	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
Unsigned Binary	1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111

10

Binary to Decimal Conversion

Method:

1. Separate the integer and fractional parts.
2. Convert the integer part using powers of 2 (left to right).
3. Convert the fractional part using negative powers of 2 (right to left).

Example 1: Convert 101.0110_2 to Decimal

- Integer part:
  $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 0 + 1 = 5$
- Fractional part:
  $0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} = 0 + 0.25 + 0.125 + 0 = 0.375$
- **Result:** 5.375_{10}

11

Decimal to Binary Conversion

Method:

1. Multiply the fractional part by 2.
2. Record the integer part (0 or 1).
3. Use the new fractional part and repeat.
4. Stop when fraction becomes 0 or repeat pattern is detected.

Example 2: Convert 0.5_{10} to Binary

- Step 1: 0.5 × 2 = **1.0** → Integer part = 1
- **Result:** 0.1_2

Example 3: Convert 0.1_{10} to Binary

- 0.1 × 2 = 0.2 → 0
- 0.2 × 2 = 0.4 → 0
- 0.4 × 2 = 0.8 → 0
- 0.8 × 2 = 1.6 → 1
- 0.6 × 2 = 1.2 → 1
- 0.2 × 2 = 0.4 → 0 (pattern repeats)
- **Result:** $0.0001100110011\ldots_2$ **(repeating)**

The Example 3 shows that some numbers cannot be stored exactly in limited number of bits.

12

Write a code: (i) define and Initialize a real variable a= 0.0; then (ii) Add 0.1 to the variable 100 000 times. The answer is 10000.00.

```c
#include <stdio.h>

int main() {
    int n;
    float a=0.0;
    for(n=0;n<100000;n++) a+=0.1;
    printf("%f",a);
    return 0;
}
```

Output

```
9998.556641
```

```c
#include <stdio.h>

int main() {
    int n;
    double a=0.0;
    for(n=0;n<100000;n++) a+=0.1;
    printf("%f",a);
    return 0;
}
```

Output

```
10000.000000
```

The only difference is how the variable a is declared. The **float** data type uses 32 bits to represent real numbers. But data type **double** uses 64 bits. This example shows there may be an error even when we define/store a real number in a computer/digital system. When we use more bits, the error decreases

13

Floating-Point Representation

Computers use a normalised floating-point binary representation for real numbers:

$$x \approx \pm q \times 2^n$$

The number q is the **mantissa** and it is a finite binary expression satisfying the inequality $1/2 \leq q < 1$, and the integer n is called the **exponent**. Exponent stored as the 2's complement format.

On the other hand IEEE-754 standard uses the following format

$$x = (-1)^s \times (1.f)_2 \times 2^e \quad \text{where} \quad 1 \leq (1.f) < 2$$

In IEEE-754 the **biased exponent** is preferred so that all stored exponent values are non-negative.

Remember $0.5_{DEC}=0.1000000_{BIN}$ and $0.1_{DEC}=0.00011000110001100011_{BIN}$

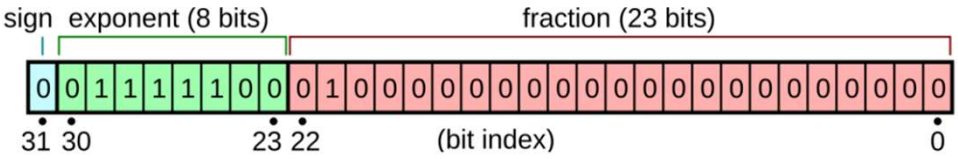
14

Single-precision floating-point format

The IEEE 754 standard specifies a *binary32* as having:

- Sign bit: 1 bit
- Exponent width: 8 bits
- Significand precision: 24 bits (23 explicitly stored)

32 bit types	
C/C++	float
Matlab	single
Julia	Float32



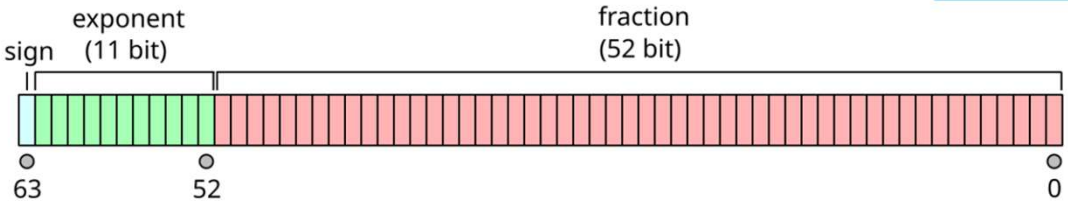
15

Double-precision floating-point format

The IEEE 754 standard specifies a **binary64** as having:

- Sign bit: 1 bit
- Exponent: 11 bits
- Significand precision: 53 bits (52 explicitly stored)

64 bit types	
C/C++	double
Python	double
Matlab	double
Julia	Float64



16

Find $x = \sqrt{S}$ using Heron's method

The Heron's method, also known as the Babylonian method, is an algorithm for approximating square roots.

It's an **iterative process** where you start with an initial guess and repeatedly refine it until you reach the desired level of accuracy. **Iteration** is the process of **repeatedly applying** a **formula** or **set of steps** to refine an approximate solution to a problem until a predefined **stopping condition** is satisfied.

Fundamental assumption which based on the method is,

$\lim_{n\to\infty} x_n = \sqrt{S}$ where $x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$

17

An algorithm to find $x = \sqrt{S}$

- **Make an initial guess**. Guess any positive number x_0.
  Note: The better the initial guess, the faster the algorithm converges.
- **Improve the guess**. Apply the formula $x_1 = (x_0 + S / x_0) / 2$. The number x_1 is a better approximation to sqrt(S).
- **Iterate until convergence**. Apply the formula,

$$x_{n+1} = \frac{\left(x_n + \frac{S}{x_n}\right)}{2}$$

until the process converges. Convergence is achieved when the **error** is less then a *predefined value*. For example, **| xn+1- xn|<1E-3** or **| (xn+1- xn)/ xn| <0.5%** . The choice between **absolute error** and **relative error** should be made based on the values and the problem.

18

To calculate $\sqrt{S}$, where $S = 125348$, to six significant figures, use the rough estimation method above to get

$$
\begin{aligned}
x_0 &= 6 \cdot 10^2 && = 600.000 \\
x_1 &= \tfrac{1}{2}\left(x_0 + \tfrac{S}{x_0}\right) && = \tfrac{1}{2}\left(600.000 + \tfrac{125348}{600.000}\right) && = 404.457 \\
x_2 &= \tfrac{1}{2}\left(x_1 + \tfrac{S}{x_1}\right) && = \tfrac{1}{2}\left(404.457 + \tfrac{125348}{404.457}\right) && = 357.187 \\
x_3 &= \tfrac{1}{2}\left(x_2 + \tfrac{S}{x_2}\right) && = \tfrac{1}{2}\left(357.187 + \tfrac{125348}{357.187}\right) && = 354.059 \\
x_4 &= \tfrac{1}{2}\left(x_3 + \tfrac{S}{x_3}\right) && = \tfrac{1}{2}\left(354.059 + \tfrac{125348}{354.059}\right) && = 354.045 \\
x_5 &= \tfrac{1}{2}\left(x_4 + \tfrac{S}{x_4}\right) && = \tfrac{1}{2}\left(354.045 + \tfrac{125348}{354.045}\right) && = 354.045
\end{aligned}
$$

47.3

3.12

0.014

2.7E-07

0 ← Stop here

The x_n values does not change anymore.

Therefore, $\sqrt{125348} \approx 354.045$.

19

Example: Use Heron's method to find $\sqrt{9}$ using the following initial guess values: (a) $x_0 = 1$; (b) $x_0 = 4$; (c) $x_0 = 7$

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$$

	x_0	x_1	x_2	x_3	x_4	x_5
(a)	1	5	3.4	3.023529	3.000092	3
(b)	4	3.125	3.0025	3.000001	3	3
©	7	4.142857	3.157635	3.003935	3.000003	3

20

Example: Use Heron's method to find $\sqrt{0.09}$ using the following initial guess values: (a) $x_0 = 1$; (b) $x_0 = 0.1$; (c) $x_0 = 0.04$

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$$

	x_0	x_1	x_2	x_3	x_4	x_5
(a)	1	0.545	0.355069	0.30427	0.30003	0.3
(b)	0.1	0.5	0.34	0.302353	0.300009	0.3
©	0.04	1.145	0.611801	0.379454	0.308318	0.300112

21

A function: Heron

```
##Matlab
function x1 = heron (S,x0)
 x1=(x0+S/x0)/2;
 while abs(x0-x1)>1.0E-4
  x0=x1
  x1=(x0+S/x0)/2
 end
end
```

Stopping condition

This code works in both MATLAB and GNU Octave

```
##GNU Octave
function x1 = heron (S,x0)
 x1=(x0+S/x0)/2;
 while abs(x0-x1)>1.0E-4
  x0=x1
  x1=(x0+S/x0)/2
 endwhile
endfunction
```

This code works in GNU Octave

22

Solving nonlinear equations

Let f(x) to be a nonlinear function of the variable x. The equation **f(x)=0** can be solved by different methods.

Examples:

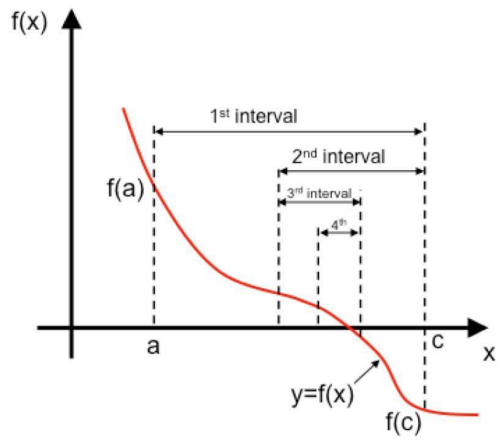
$$x^2 - 3x + 4 = 0$$
$$x + 5 = 0$$
$$\cos(x) - \tan(x) + 0.5 = 0$$
$$\ln(x - 1) + \exp(3x) + 2 = 0$$

In some cases, we may find analytical solutions. But in many practical cases either the analytical solutions are not available, or it is very difficult to find them. Therefore, **numerical methods** are required to solve such problems.

23

Bisection Method

- The **Bisection Method** is a numerical **root-finding technique** used to solve equations. It works by **dividing an interval into smaller intervals** and narrowing down the range of possible solutions **iteratively**.
- The process of repeatedly dividing a set in half to find a solution to a problem is known as **binary search**.



24