CS601: Software Development for Scientific Computing

Autumn 2022

Week1: Overview

Let us listen to Jack Dongarra

- https://youtu.be/Oe9LRKoE6L0

Course Takeaways (intended)

- Non-CS majors:
  1. Write code and
  2. Develop software (not just write standalone code)
     - Numerical software

- CS-Majors:

  In addition to the above two:

  3. Learn to face mathematical equations and implement them with confidence

What is this course about?

Software Development

+

Scientific Computing

This course NOT about..

- Software Engineering
  - Systematic study of Techniques, Methodology, and Tools to build correct software within time and price budget (topics covered in CS305)
    - People, Software life cycle and management etc.
- Scientific Computing
  - Rigorous exploration of numerical methods, their analysis, and theories
  - Programming models (topics covered in CS410)

Who this course is for?

- You are interested in scientific computing
- You are interested in high-performance computing
- You want to build / add to a large software system

Software Development

- *Software development is the process of conceiving, specifying, designing, programming, documenting, testing, and bug fixing involved in creating and maintaining applications, frameworks, or other software components.*

Software development is a process of writing and maintaining the source code, but in a broader sense, it includes all that is involved between the conception of the desired software through to the final manifestation of the software, …

- Wikipedia on "Software Development"

Scientific Computing

- Also called computational science
  - *Development of models to understand systems (biological, physical, chemical, engineering, humanities)*

Collection of tools, techniques, and theories required to solve on a computer mathematical models of problems in science and engineering

Why C++ ?

- C/C++/Fortran codes form the majority in scientific computing codes
- Catch a lot of errors early (e.g. at *compile-time* rather than at *run-time)*
- Has features for *object-oriented* software development
- Known to result in codes with better *performance*

Let us dive into an example….

Example - Factorial

- `n! = n x (n-1) x (n-2) x . . . x 3 x 2 x 1`

  `(n–1)! = (n-1) x (n-2) x . . . x 3 x 2 x 1`

  therefore,

  Definition1: `n! = n x (n-1)!`

  is this definition complete?

  - plug 0 to n and the equation breaks.

  Definition2:

$$
n! = \begin{cases} n \times (n-1)! & \text{when } n>=1 \\ 1 & \text{when } n=0 \end{cases}
$$

Exercise 1

- Does this code implement the definition of factorial correctly?

```
int fact(int n){
    if(n==0)
       return 1;

    return n*fact(n-1);

}
```

Example - Factorial

Definition2:

$$
n! = \begin{cases} n \times (n-1)! & \text{when } n>=1 \\ 1 & \text{when } n=0 \end{cases}
$$

is this definition complete?

- `n!` is not defined for negative `n`

Solution - Factorial

```
int fact(int n){
    if(n<0)
        return ERROR;
    if(n==0)
        return 1;

    return n*fact(n-1);

}
```

Exercise 2

- In how many `flops` does the code execute?

  assume 1 `flop = 1` step executing ***any*** arithmetic operation

```
int fact(int n){
    if(n<0)
        return ERROR;
    if(n==0)
        return 1;

    return n*fact(n-1);

}
```

Exercise 3

- Does the code yield correct results for any n?

```
int fact(int n){
    if(n<0)
        return ERROR;
    if(n==0)
        return 1;

    return n*fact(n-1);

}
```

Who this course is for?

- Anybody who wishes to develop "computational thinking"
  - A skill necessary for everyone, not just computer programmers
  - An approach to problem solving, designing systems, and understanding human behavior that draws on concepts fundamental to computer science.

Computational Thinking - Examples

- How difficult is the problem to solve? And what is the best way to solve?
- Modularizing something in anticipation of multiple users
- Prefetching and caching in anticipation of future use
- Thinking recursively
- Reformulating a seemingly difficult problem into one which we know how to solve by reduction, embedding, transformation, simulation
  - Are approximate solutions accepted?
  - False positives and False negatives allowed? etc.
- Using abstraction and decomposition in tackling large problem
- ...

Computational Thinking – 2 As

- Abstractions
  - Our "mental" tools
  - Includes: <u>choosing right abstractions</u>, operating at multiple <u>layers</u> of abstractions, and defining <u>relationships</u> among layers
- Automation
  - Our "metal" tools that <u>amplify</u> the power of "mental" tools
  - Is mechanizing our abstractions, layers, and relationships
    - Need precise and exact notations / models for the "computer" below ("computer" can be human or machine)

Computing - 2 As Combined

- Computing is the automation of our abstractions
- Provides us the ability to scale
  - Make infeasible problems feasible
    - E.g. SHA-1 not safe anymore
  - Improve the answer's precision
    - E.g. capture the image of a black-hole

Summary: choose the right abstraction and computer

Recap

- Need to be precise
  - recall: `n! = 1` for `n=0,` not defined for negative `n`
- Choosing right abstractions
  - recall: use of recursion, correct data type
- Ability to define the complexity
  - recall: `flop` calculation
- Next?

Recap

- Need to be precise
  - recall: `n! = 1` for `n=0`, not defined for negative `n`
- Choosing right abstractions
  - recall: use of recursion, correct data type
- Ability to define the complexity
  - recall: `flop` calculation
- Choose the right “computer” for mechanizing the abstractions chosen

Scientific Software - Characteristics

- The answer is not a typical yes/no, red/blue/green
- The answer varies continuously. Think of computing the value of pi = 3.141592…
- Uses approximations. Think of *discretization*
- Employs efficient *kernels*
  – Kernels are core operations that are executed very frequently
- Should be able to adapt to change.
  – Writing everything from scratch is not an option
- Deals with large-scale problems
  – Lot of input/output data or both
  – Computationally hard

General Approach to Solving a Computational Problem

1. **Problem statement:** more precise this is, the easier it is to design and implement
2. **Solution Algorithm:** exactly how is the problem going to be solved
3. **Implementation:** breaking the algorithm into manageable pieces and putting it all together to solve the problem using a language of choice.
4. **Verification:** checking that the implementation solves the original problem.
    1. Often most difficult step, because you don’t know the correct answer.

Toward Scientific Software

- Necessary Skills:
  - Understanding the mathematical problem
  - Understanding numerics
  - Designing algorithms and data structures
  - Selecting language and using libraries and tools
  - Verify the correctness of the results
  - Quick learning of new programming languages
    - E.g. Regent

Exercise

Compute <u>root(s)</u> of:

$$x = \cos x; \; x \in \mathbb{R}$$

`roots`, also called `zeros`, is the value of the argument/input to the function when the function output vanishes i.e. becomes zero

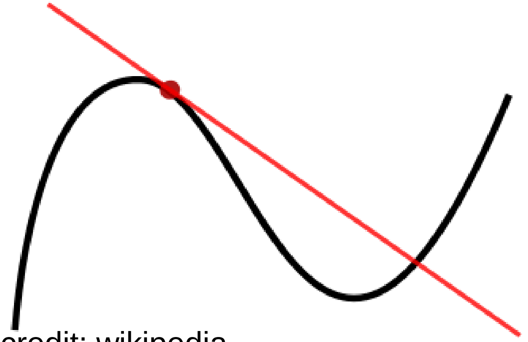
Mathematical Problem

- let $y = f(x)$

  $f(x) = \cos(x) - x$

- At $\mathrm{x} = \mathrm{x_n}$, the value of y is $f(x_n)$. The coordinates of the point are ($\mathrm{x_n}$, $f(x_n)$) = known point.

- From calculus: ***<u>derivative</u>*** of a function of single variable at a chosen input value, when it exists, is the ***<u>slope of the tangent</u>*** to the graph at that input value.
  - $f'(x_n)$ is the slope of the line that is tangent to $f(x)$ at $\mathrm{x_n}$



credit: wikipedia

Mathematical Problem

- From high-school math: point-slope formula for equation of a line

$$y - y_1 = m(x - x_1),$$

given the slope m and any known point (x_1, y_1)

- Substituting with:
  - $(x_n, f(x_n))$ = known point
  - $f'(x_n)$ = slope

Equation of the tangent line to graph of $f(x)$ at x_n :

$$y - f(x_n) = f'(x_n)(x - x_n)$$

Mathematical Problem

- Interested in finding roots i.e. value of x at $y=0$ i.e. at point $(x_{np1}, 0)$.
- Substituting in the equation of the tangent line,

$$y - f(x_n) = f'(x_n)(x - x_n)$$

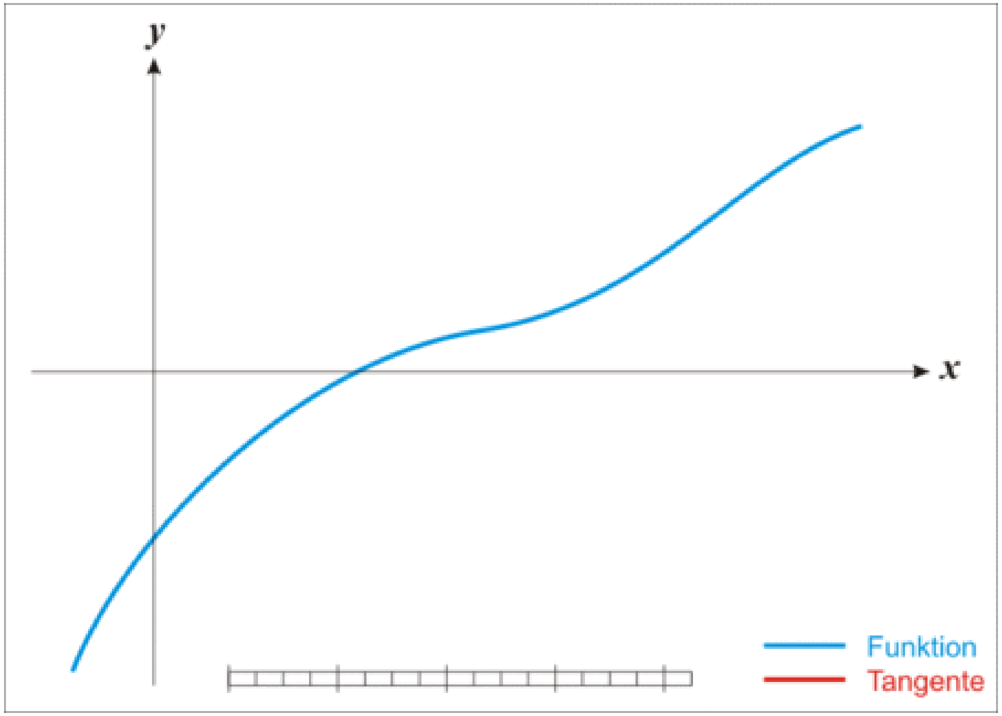
$$= \quad -f(x_n) = f'(x_n)(x_{np1} - x_n)$$

$$= \quad \boldsymbol{x_{np1} = x_n - f(x_n) / f'(x_n)}$$

Mathematical Problem

- Visualizing

  (source: https://en.wikipedia.org/wiki/Newton's_method) :



The function f is shown in blue and the tangent line is in red. We see that x_{n+1} is a better approximation than x_n for the root x of the function f.

Mathematical Problem

$\boldsymbol{x_2 = x_1 - \; f(x_1)}$ / $\boldsymbol{f'(x_1)}$

$\boldsymbol{x_3 = x_2 - \; f(x_2)}$ / $\boldsymbol{f'(x_2)}$

$\boldsymbol{x_4 = x_3 - \; f(x_3)}$ / $\boldsymbol{f'(x_3)}$

$\boldsymbol{...}$

Numerical Analysis

Talk to domain experts

- Choosing the initial value of x
- Does the method converge ?
- What is an acceptable approximation?
- etc.

Designing Algorithms and Data Structures

- Start with x_1

$$\boldsymbol{x_2 = x_1 - f(x_1) \,/\, f'(x_1)}$$

$$\boldsymbol{x_3 = x_2 - f(x_2) \,/\, f'(x_2)}$$

$$\boldsymbol{x_4 = x_3 - f(x_3) \,/\, f'(x_3)}$$

…

- Repeat for up to `maxIterations`
- Check for $x_{n+1} - x_n$ to be "sufficiently small"
- Choose appropriate data types for x

Selecting libraries and tools

- E.g. use the math library in C++ (cmath)

Verify the correctness of results

- Compare with ‘gold’ code / benchmark
- Compare with empirical data

Real Numbers $\mathbb{R}$

- Most scientific software deal with Real numbers. Our toy code dealt with Reals
  - Numerical software is scientific software dealing with Real numbers
- Real numbers include rational numbers (integers and fractions), irrational numbers (pi etc.)
- Used to represent values of continuous quantity such as time, mass, velocity, height, density etc.
  - Infinitely many values possible
  - But computers have limited memory. So, have to use approximations.

Representing Real Numbers

- Real numbers are stored as *floating point numbers*
  (floating point system is a scheme to represent real numbers)
- E.g. floating point numbers:
  - $\pi = 3.14159$,
  - $6.03*10^{23}$
  - $1.60217733*10^{-19}$

exponent

General format: $\pm \mathbf{x} \times \mathbf{b}^{\mathbf{e}}$

mantissa
(number ranges from:
1 to b   OR   1/b to 1)

base
(e.g. base 10, 8, 2, 16)

3-digit Calculator

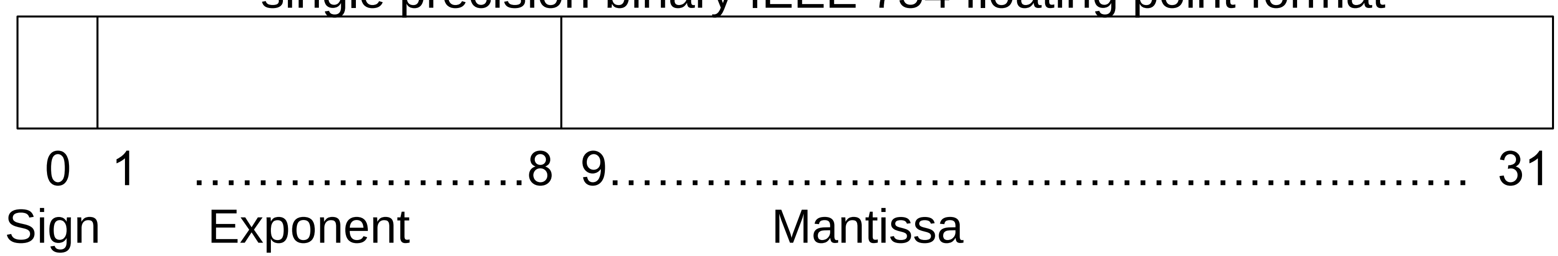
- Suppose base, b=10 and
- $x = \pm d_0 . d_1 d_2 \times 10^e$ where $\begin{cases} 1 \leq d_0 \leq 9, \\ 0 \leq d_1 \leq 9, \\ 0 \leq d_2 \leq 9 \\ -9 \leq e \leq 9 \end{cases}$
- precision = length of mantissa
  - What is the precision here?
- Exercise: What is the smallest positive number?
- Exercise: What is the largest positive number?
- Exercise: When is this representation not enough?
- Exercise: How many numbers can be represented in this format?

IEEE 754 Floating Point System

- Prescribes single, double, and extended precision formats

Precision	u	Total bits used (sign, exponent, mantissa)
Single	6×10^{-8}	32 (1, 8, 23)
Double	2×10^{-16}	64 (1, 11, 52)
Extended	5×10^{-20}	80 (1, 15, 64)

single precision binary IEEE 754 floating point format



IEEE 754 Floating Point Arithmetic

double precision binary IEEE 754 floating point format

0	1…………………..11	1………………………………………….52
Sign	Exponent	Mantissa

- if exponent bits e_1-e_{11} are not all 1s or 0s, then the *normalized* number

$$n = \pm(1.m_1 m_2..m_{52})_2 \times 2^{(e_1 e_2..e_{11})_2 - 1023}$$

- **Machine epsilon** is the gap between 1 and the next largest floating point number. $2^{-52} \approx 10^{-16}$ for double.
- Exercise: What is minimum positive *normalized* double number?
- Exercise: What is maximum positive *normalized* double number?

IEEE 754 Floating Point Arithmetic

double precision binary IEEE 754 floating point format

0 1…………………..11 1……………………………………………52

Sign Exponent Mantissa

- if exponent bits e_1-e_{11} are all 0s, then:

  the *subnormal* number

$$n = \pm(\mathbf{0}.m_1 m_2 .. m_{52})_2 \times 2^{(e_1 e_2 .. e_{11})_2 - 102\mathbf{2}}$$

- if exponent bits e_1-e_{11} are all 1s, then:

  we can get –inf, NaN, or +inf based on value of $m_1 m_2 .. m_{52}$
  - If any m is non-zero, the number is NaN (not a number)

IEEE 754 Floating Point Arithmetic

- Order is important
  - Floating point arithmetic is *not associative*
    - (x+y)+z not the same as x+(y+z)

- Explicit coding of textbook formula may not be the best option to solve
  - $x^2 - 2px - q = 0$ p and q are positive: p=12345678, q=1
  - **Exercise:** find the minimum of the roots.

- Subtracting approximations of two nearby numbers results in a bad approximation of the actual difference – **catastrophic cancellation**

Floating Point System - Terminology

- **Precision (p)** - Length of mantissa
  - E.g. p=3 in 1.00×10^{-1}
- **Machine epsilon (ϵ_{mach})** – smallest a-1, where a is the smallest representable number greater than 1
  - E.g. ϵ_{mach} =1.001 – 1.000 = 0.001.
- **Unit roundoff (u)** – smallest positive number where the *computed* value of 1+u is different from 1
  - E.g. suppose p=4 and we wish to compute 1.0000+ 0.0001=1.0001
  - But we can't store the exact result (since p=4). We end up storing 1.000.
  - So, computed result of 1+u is same as 1
  - Suppose we tried adding 0.0005 instead. 1.0000+0.0005=1.0005 Now, round this: 1.001

$\Rightarrow$ **u =0.0005**

$\Rightarrow$ **usually u = ½ * ϵ_{mach}**