CS601: Software Development for Scientific Computing

Autumn 2021

Week1: Overview

Who this course is for?

- Anybody who wishes to develop "computational thinking"

  - A skill necessary for everyone, not just computer programmers

  - More on this later…

Course Takeaways

- ## Non-CS majors:
  - – Write code and
  - – Develop software (not just write standalone code)
    - • Numerical software

- ## CS-Majors:
  - – Face mathematical equations and implement them with confidence

What is this course about?

Software Development

+

Scientific Computing

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

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

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*

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

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

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:

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

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

```c
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?

  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);

}
```

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

The von Neumann Architecture
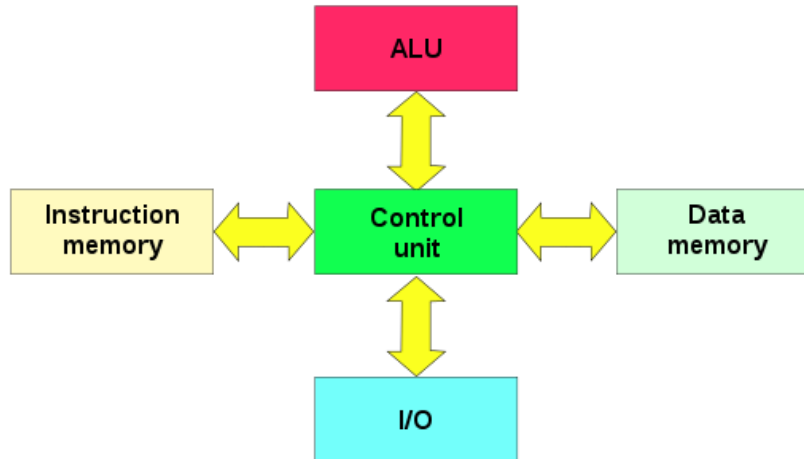
- Proposed by Jon Von Neumann in 1945
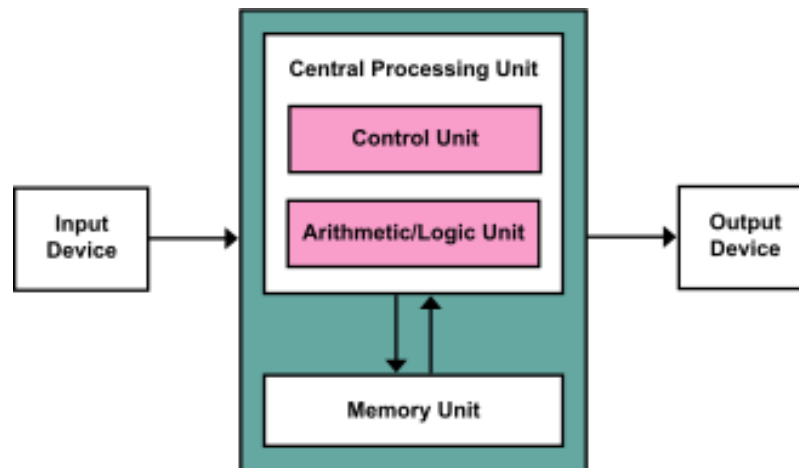
source: wikipedia

- The memory unit stores both instruction and data
  - consequence: cannot fetch instruction and data simultaneously - *von Neumann bottleneck*
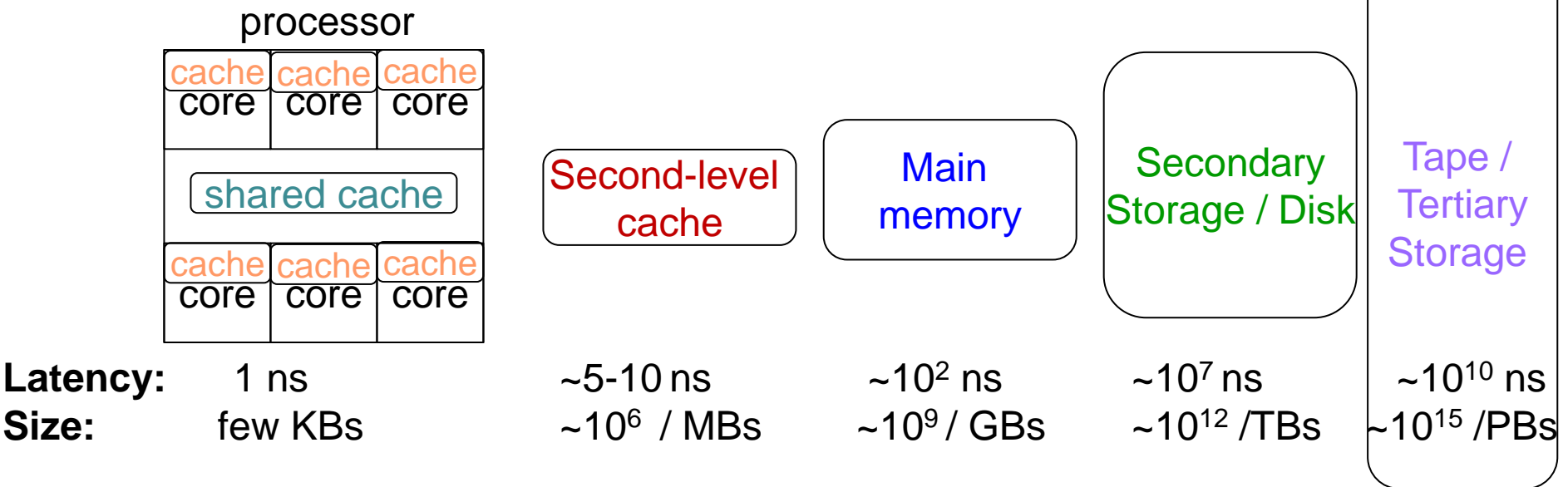
Harvard Architecture

- Origin: Harvard Mark-I machines
- Separate memory for instruction and data

  – advantage: speed of execution
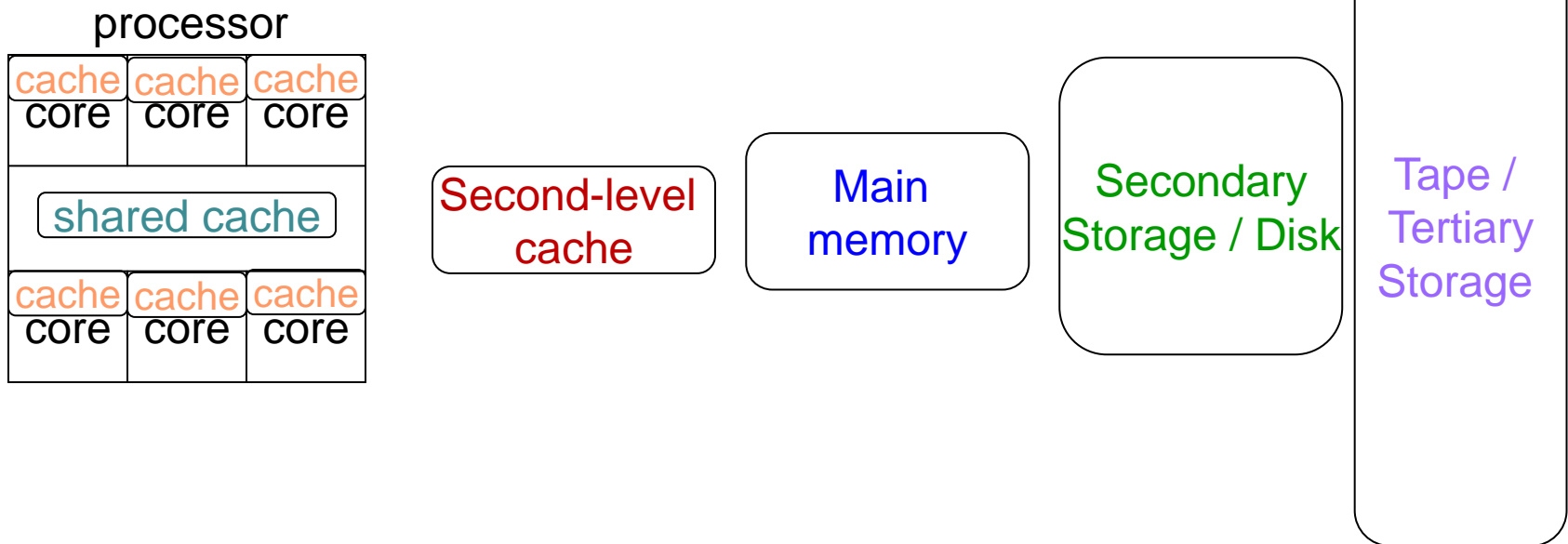 – disadvantage: complexity

Memory Hierarchy

- Most computers today have layers of cache in between processor and memory

processor

cache	cache	cache
core	core	core

shared cache

cache	cache	cache
core	core	core

Second-level cache

Main memory

Secondary Storage / Disk

Tape / Tertiary Storage

Latency: 1 ns   ~5-10 ns   ~10^2 ns   ~10^7 ns   ~10^{10} ns

Size: few KBs   ~10^6 / MBs   ~10^9 / GBs   ~10^{12} /TBs   ~10^{15} /PBs

- Closer to cores exist separate D and I caches

- Where are *registers?*

24

Memory Hierarchy

- Consequences on programming?
  - Data access pattern influences the performance
  - Be aware of the *principle of locality*

processor

cache	cache	cache
core	core	core

shared cache

cache	cache	cache
core	core	core

Second-level cache

Main memory

Secondary Storage / Disk
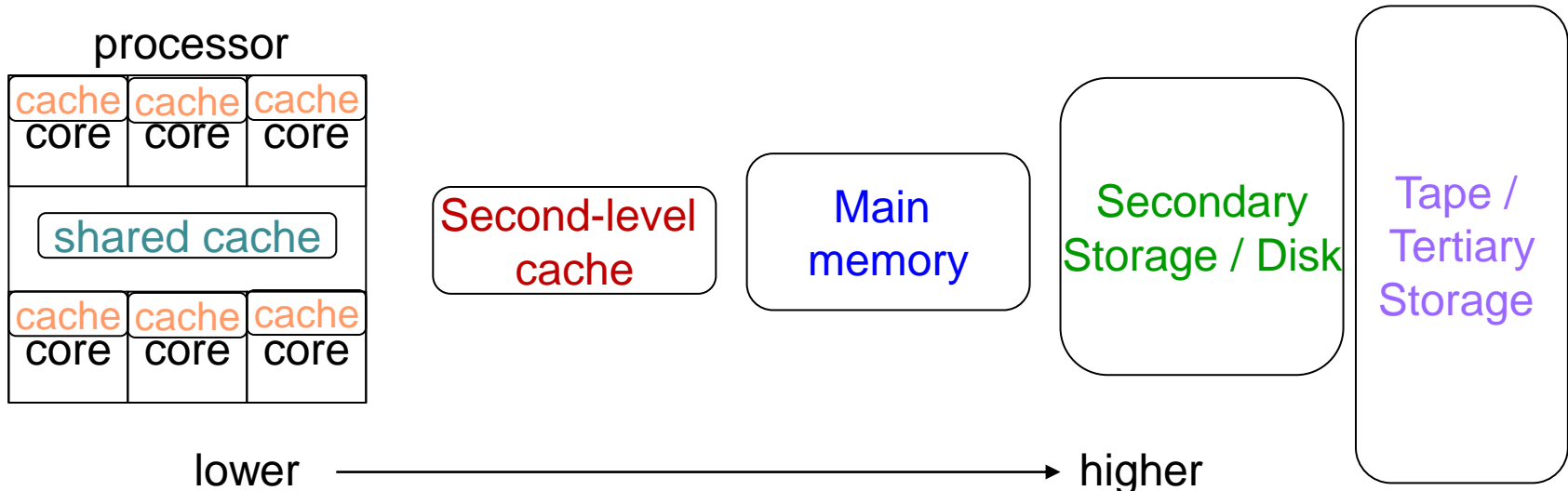
Tape / Tertiary Storage

Principle of Locality

1. If a data item is accessed, it will tend to be accessed soon *(temporal locality)*
   - So, keep a copy in cache
   - E.g. loops

2. If a data item is accessed, items in nearby addresses in memory tend to be accessed soon *(spatial locality)*
   - Guess the next data item (based on access history) and fetch it
   - E.g. array access, code without any branching

Memory Hierarchy - Terminology

- <u>Hit:</u> data found in a lower-level memory module
  - Hit rate: fraction of memory accesses found in lower-level

- <u>Miss:</u> data to be fetched from the next-level (higher) memory module
  - Miss rate: 1 – Hit rate
  - Miss penalty: time to replace the data item at the lower-level

processor

cache	cache	cache
core	core	core

shared cache

cache	cache	cache
core	core	core

Second-level cache

Main memory

Secondary Storage / Disk

Tape / Tertiary Storage

lower ———————————————→ higher

Scientific Software - Examples

Biology

  - Shotgun algorithm expedites sequencing of human genome
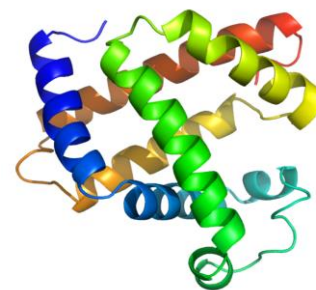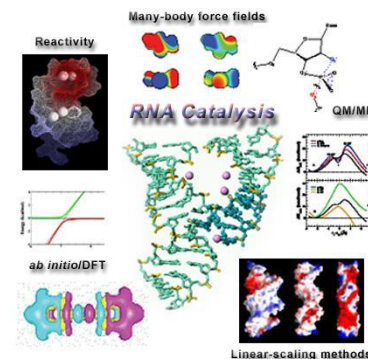

Credit: Wikipedia

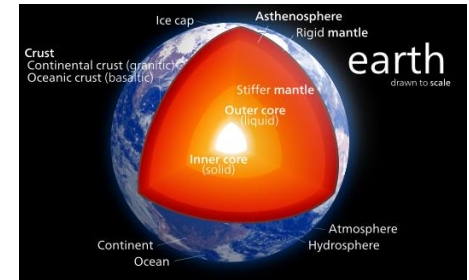 - Analyzing fMRI data with machine learning

Credit: Wikipedia

Chemistry

  - optimization and search algorithms to identify best chemicals for improving reaction conditions to improve yields

28

Credit: University of Minnesota

Scientific Software - Examples

Geology
- Modeling the Earth's surface to the core

Credit: Wikipedia

Astronomy
- kd-trees help analyze very large multi-dimensional data sets

Credit: Kaggle.com

Engineering
- Boeing 777 tested via computer simulation (not via wind tunnel)

Scientific Software - Examples

Economics
- ad-placement

Entertainment
- Toy Story, Shrek rendered using data center nodes

Toward Scientific Software

Physical process

↓

Mathematical model

↓

Algorithm

↓

Software program

↓

Simulation results

Toward Scientific Software

- Necessary Skills:
  - Understanding the mathematical problem
  - Understanding numerics
  - Designing algorithms and data structures
  - Selecting 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 $x = x_n$, the value of y is $f(x_n)$. The coordinates of the point are $(x_n, f(x_n))$ = known point.

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

credit: wikipedia

34
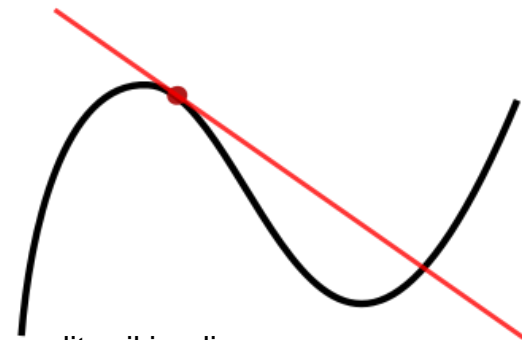
Mathematical Problem

- 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 :

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

35

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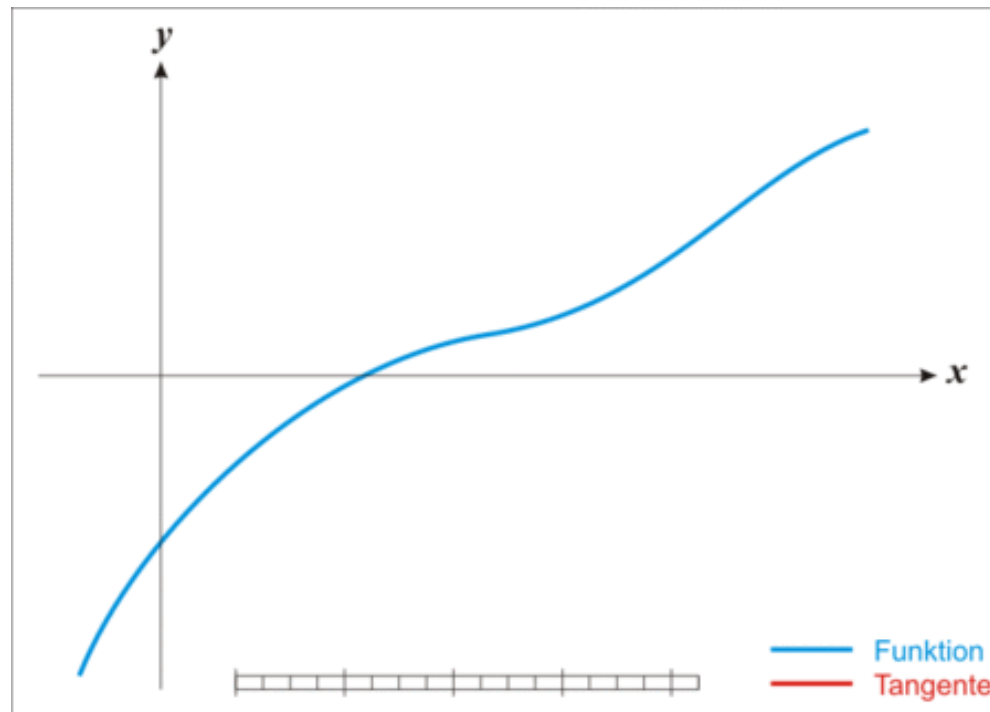
$$y - f(x_n) = f'(x_n)(x - x_n)$$

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

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

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

$$x_2 = x_1 - f(x_1) \, / \, f'(x_1)$$
$$x_3 = x_2 - f(x_2) \, / \, f'(x_2)$$
$$x_4 = x_3 - f(x_3) \, / \, f'(x_3)$$
$$\ldots$$

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

$$x_2 = x_1 - f(x_1) \mathbin{/} f'(x_1)$$
$$x_3 = x_2 - f(x_2) \mathbin{/} f'(x_2)$$
$$x_4 = x_3 - f(x_3) \mathbin{/} f'(x_3)$$

…

- 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