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

- 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: choosing right abstractions, operating at multiple layers of abstractions, and defining relationships among layers
- Automation
  - Our "metal" tools that amplify 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

- $n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$

  $(n-1)! = (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$

  therefore,

  Definition1: $n! = n \times (n-1)!$

  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?

  `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

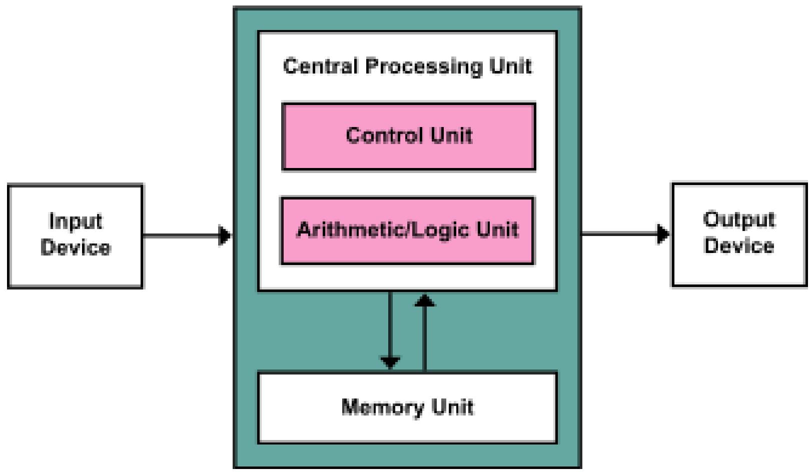
- Need to be precise
  - recall: `n! = 1` for `n=0,` not defined for negative `n`
- Choosing right abstractions
  - recall: use of recursion, correct data type
- Ability to define the complexity
  - recall: `flop` calculation
- Next?

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

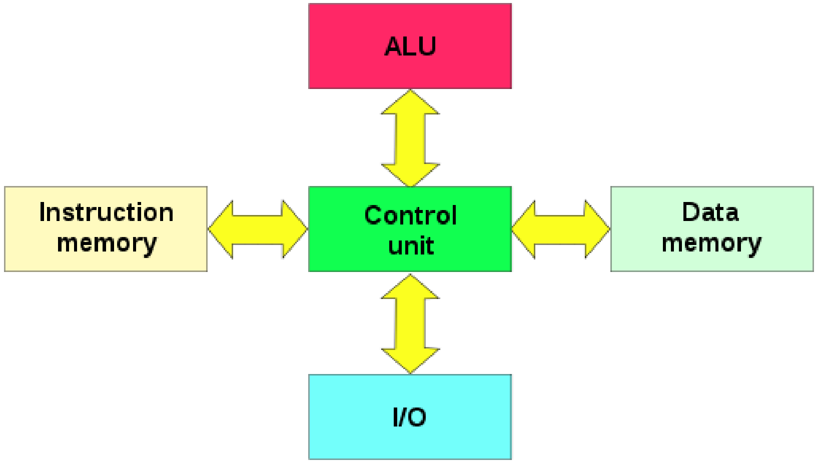


source: wikipedia

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

Harvard Architecture

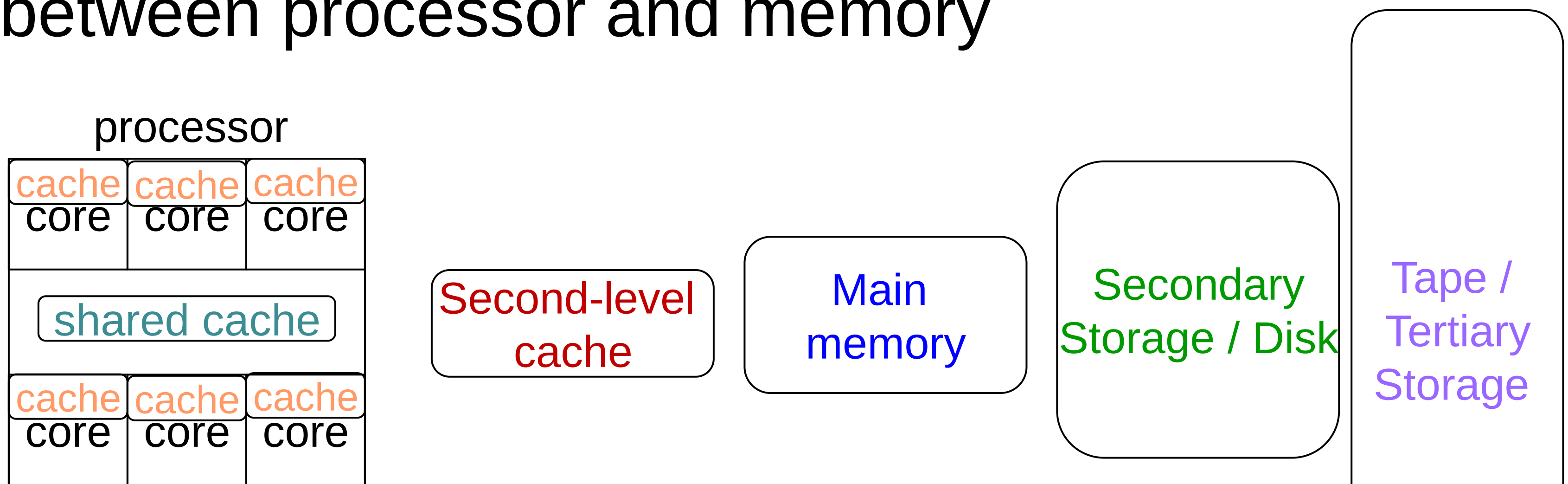
- Origin: Harvard Mark-I machines
- Separate memory for instruction and data



  - advantage: speed of execution
  - disadvantage: complexity

Memory Hierarchy

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

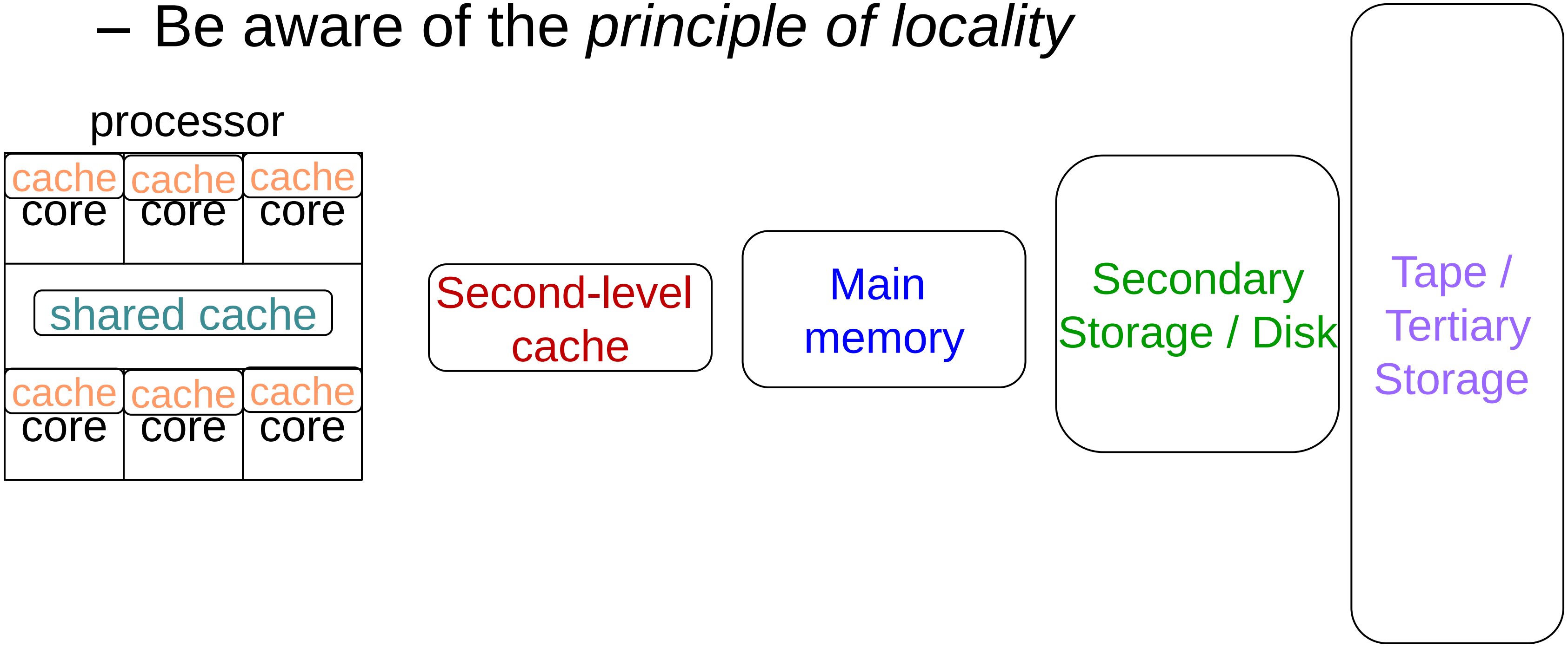


Latency:	1 ns	~5-10 ns	~10^2 ns	~10^7 ns	~10^{10} ns
Size:	few KBs	~10^6 / MBs	~10^9 / GBs	~10^{12} /TBs	~10^{15} /PBs

  - Closer to cores exist separate D and I caches
- Where are *registers?*

Memory Hierarchy

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

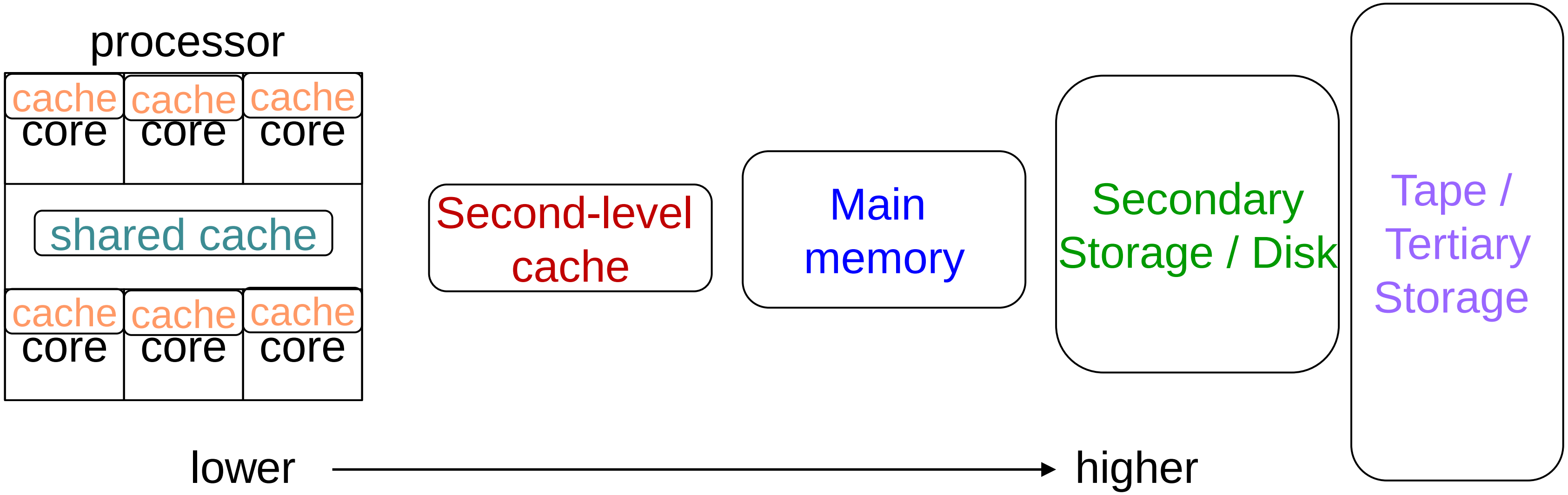


Principle of Locality

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

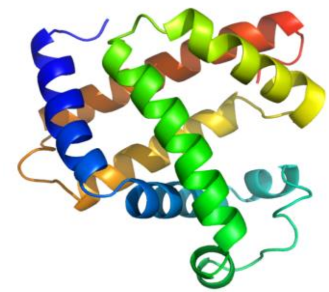
- <u>Hit:</u> data found in a lower-level memory module
  - Hit rate: fraction of memory accesses found in lower-level
- <u>Miss:</u> data to be fetched from the next-level (higher) memory module
  - Miss rate: 1 – Hit rate
  - Miss penalty: time to replace the data item at the lower-level



Scientific Software - Examples

Biology

- Shotgun algorithm expedites sequencing of human genome



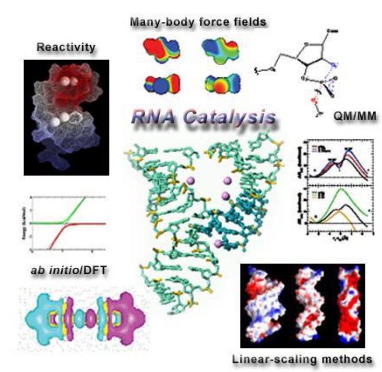
Credit: Wikipedia

- Analyzing fMRI data with machine learning

Credit: Wikipedia

Chemistry

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

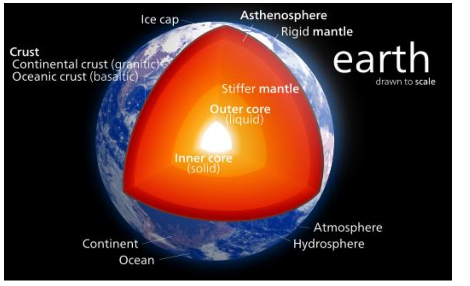


Credit: University of Minnesota

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 language and using libraries and tools
  - Verify the correctness of the results
  - Quick learning of new programming languages
    - E.g. Regent

Exercise

Compute root(s) 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

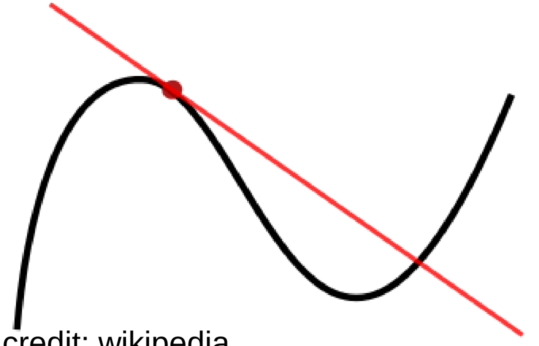
Mathematical Problem

- 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: ***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 $\mathrm{x_n}$



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 :

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

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,

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

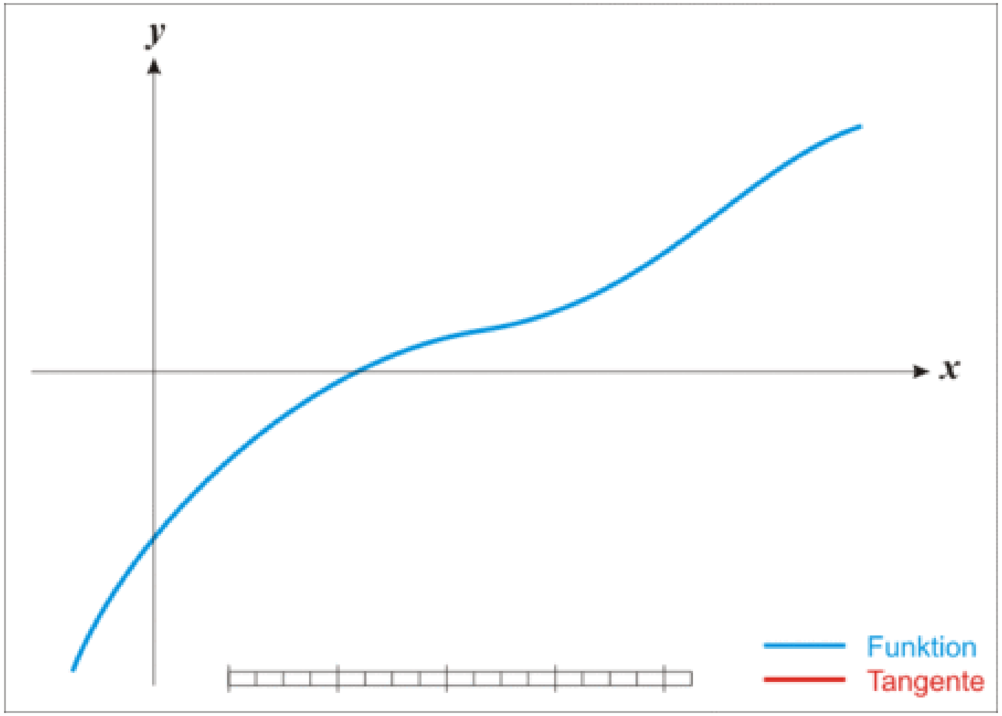
= $$-f(x_n) = f'(x_n)(xnp_1 \_ xn)$$

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

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

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

$...$

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

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

...

- 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

Recap

- Different architectures of computers
  - von Neumann, Harvard (, differences, pros and cons)
  - Modern computers and the memory hierarchy
- Implications of memory hierarchy on programmer
  - Desirable to exploit *principle of locality* to get better performance of programs
- Examples of scientific software
- Toward scientific software – steps and skills
  - dry run: toy code sample (never call it software!)
  - Demo

Scientific Software - Motifs

noun

1. a decorative image or design, especially a repeated one forming a pattern.
   "the colourful hand-painted motifs which adorn narrowboats"

   Similar: design | pattern | decoration | figure | shape | logo | monogram

2. a dominant or recurring idea in an artistic work.
   "superstition is a recurring motif in the book"

1. Finite State Machines
2. Combinatorial
3. Graph Traversal
4. Structured Grid
5. Dense Matrix
6. Sparse Matrix
7. FFT
8. Dynamic Programming
9. N-Body (/ particle)
10. MapReduce
11. Backtrack / B&B
12. Graphical Models
13. Unstructured Grid

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 <u>continuous quantity</u> 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:
  - $\pi = 3.14159$,
  - $6.03*10^{23}$
  - $1.60217733*10^{-19}$

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

exponent

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

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

Floating Point System - Terminology

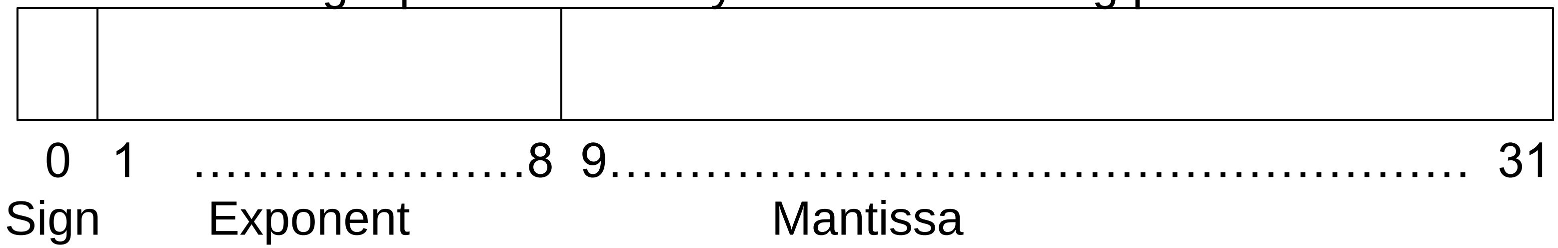
- **Precision (p)** - Length of mantissa
  - E.g. p=3 in 1.00×10^{-1}
- **Unit roundoff (u)** – smallest positive number where the *computed* value of 1+u is different from 1
  - E.g. suppose p=4 and we wish to compute
    1.0000+ 0.0001=?
  - result = 1.0001. But we can't store result exactly (since p=4). We end up storing 1.000. => computed result of 1+u is same as 1
  - Add 0.0005 instead and round. 1.0000+0.0005 = 1.0005 = 1.001 => u =0.0005
- **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. **usually ϵ_{mach} = 2u**

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



Curious case of 0.1

- The decimal number 0.1 cannot be represented exactly in binary even with p=24
  - 1.100 110 011 001 100 110 011 01 x 2^{-4} is the approximation

Exercise

- What is the largest possible *non-negative integer* number representable in 4 bits?
- What is the smallest possible *negative integer* number representable in 4 bits?
- What is the largest possible number possible in IEEE 754 single-precision floating point format?
  - Smallest?

Suggested reading: Numerical Computing with IEEE Floating Point Arithmetic, Michael Overton (chapter 4)

Bits, Nibble, ..Giga Word

- Bit – smallest unit of information storage can be 1 or 0
- Nibble – 4 bits
- Byte – 8 bits
- Half-word – 2 bytes
- Word – 4 bytes
- Giga word – 8 bytes

Number Bases

- We use decimal (base-10), Computers use binary (base-2).
- Binary is difficult to read. So, we use Hexadecimal (base-16).
- Octal (base-8) is the other popular number format.

Number Bases - Hexadecimal

- Hexadecimal uses 16 digits: 0 to 9 and A to F. A to F represent decimal numbers 10 to 15.
- A digit in hexadecimal needs 4 bits. Therefore, a byte of information (8 bits) represents two digits.
- Example:

Decimal	Binary	Hexadecimal
10	1010	0xA
16	1 0000	0x10
43981	1010 1011 1100 1101	0xABCD

How are Numbers Stored in Memory? - Endianness

– Assume an integer needs 4 bytes of storage
  - E.g. 1193 in Hexadecimal = 0x4A9 = 0x 00 00 04 A9 when stored in 4 bytes of memory.
  - How are those 4 bytes ordered in memory? – Endianness

– Two popular formats: Big-Endian and Little-Endian

Big-Endian

– Most-significant-byte (MSB) at low-address and least-significant-byte (LSB) at high-address

- E.g. 1193 = **0x00 00 04 A9** (= $4 * 16^2 + A * 16 + 9$)
- MSB (0x00) is written at lower address, LSB (0xA9) is written at higher address.

Address:	0x00000001	0x00000002	0x00000003	0x00000004
	0000 0000 **(00)**	0000 0000 **(00)**	0000 0100 **(04)**	1010 1001 **(A9)**

- Motorola 68000 Series, IBM-Z Mainframes.

Little-Endian

– Most-significant-byte (MSB) at high-address and least-significant-byte (LSB) at low-address

- E.g. 1193 = **0x00 00 04 A9** (= 4 * 16^2 + A * 16 + 9)
- MSB (0x00) is written at higher address, LSB (0xA9) is written at lower address.

Address:	0x00000001	0x00000002	0x00000003	0x00000004
	1010 1001 **(A9)**	0000 0100 **(04)**	0000 0000 **(00)**	0000 0000 **(00)**

- Intel x86 Architecture

Endianness

– Fortunately, we don’t have to worry about endianness.
    - You don’t have to reverse bytes when you read an integer.
    - Processor and Compiler do the job for you.

Processor

- Hardware component
- Massive collection of `and` and `or` gates
- CPU only knows how to perform operations `and`, `or`, `xor`.
- Has a small set of instructions (machine language) it can execute.
- Number of instructions per second is determined by clock speed. 1 clock tick = cycle. Modern CPUs execute more than 1 instruction per cycle.

Translation Systems

- Software components: Compilers, preprocessor, loader, linker, assembler, interpreters
- All programs ultimately need to be translated to set of instructions that CPU can understand

Operating System

- Software component
- Controls everything about how the computer works
  - E.g. Input/Output (IO), memory management
    - E.g. the OS should keep track of which parts of memory are being used and which parts are still free for use by programs and data
- Not tied to processor mostly
- Programming: depending on the language used, OS interface may or may not be important