AGEC 652 - Lecture 2.1

Numerical arithmetic

Diego S. Cardoso

Spring 2022

Course roadmap

1. Intro to Scientific Computing
2. **Numerical operations and representations**
    1. **Numerical arithmetic** $\leftarrow$ You are here
    2. Numerical differentiation and integration
    3. Function approximation
3. Systems of equations
4. Optimization
5. Structural estimation

*These slides are based on Bryant & O'Hallaron (2011), Miranda & Fackler (2001), Judd (1998), and course materials by Ivan Rudik.

Briefest review of Computer Systems

(or How do computers work?)

Why bother?

Knowing the basics of how computers perform the instructions we give them is essential to

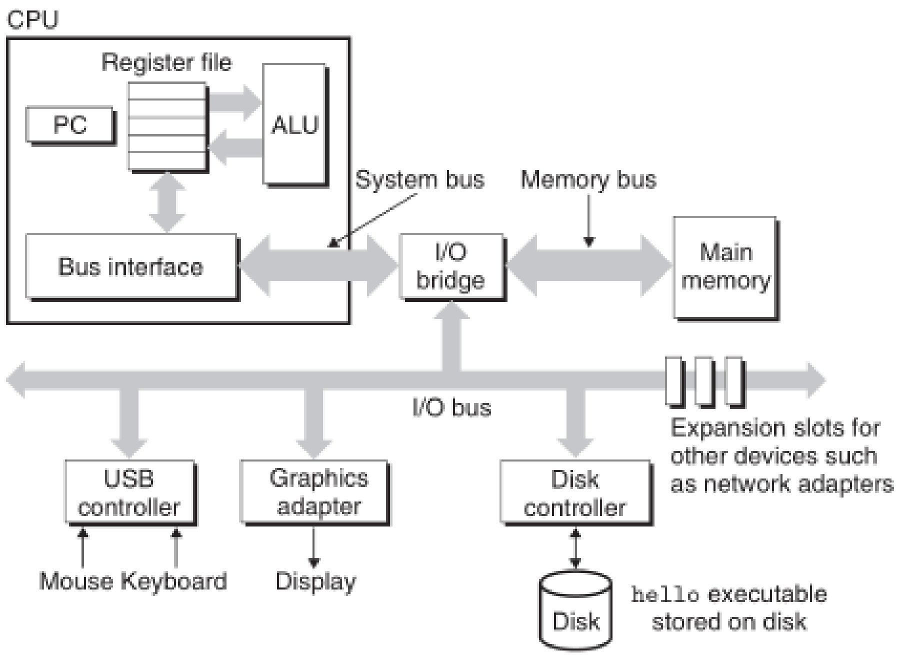
- Understand why we write code in certain ways
- Why your program fails or takes forever to run
- Write more efficient routines
- Understand the limits of computational methods in quantitative research

The building blocks of a computer

Buses: where information flows between components. They are design to transport fixed-size chunks known as *words* (32 or 64 bytes in most computers)

I/O devices: connect the system to the external world

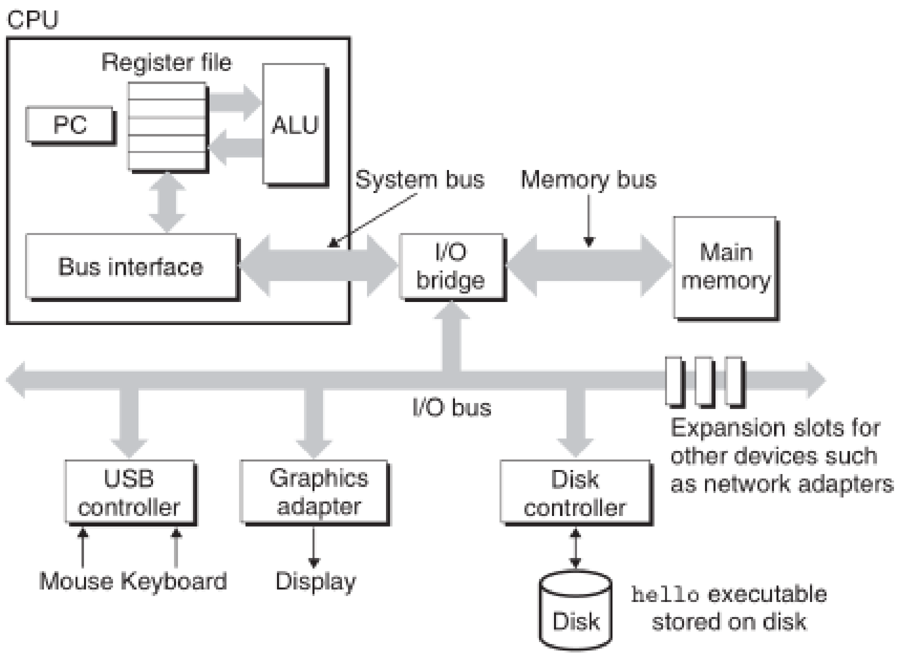
- Input: keyboard, mouse, disks, networks
- Output: display, disks, networks



The building blocks of a computer

Main memory: temporary storage of *instructions* (i.e. programs) *and data*

- Physically, a collection of DRAM chips
- Logically, a **linear array of bytes** (8 bits), each with its *unique address*



When we allocate a variable of type `Int64`, our program will determine an available address (where that variable starts in memory) and reserve a size (64 bits = 8 bytes, in this case)

When we assign `x = 10`, our program writes the binary equivalent of integer `10` in that memory address

The building blocks of a computer

```julia
x = [1 2 3]
```

```
## 1×3 Matrix{Int64}:
##  1  2  3
```

```julia
# This is how we get the physical address (0x means it's in hexadecimal base)
pointer_from_objref(x)
```

```
## Ptr{Nothing} @0x000001fb2a8be6d0
```

16 digits of hexadecimal number (4 bits): $16 \times 4 = 64$-bit address

```julia
# Get the size
sizeof(x)
```

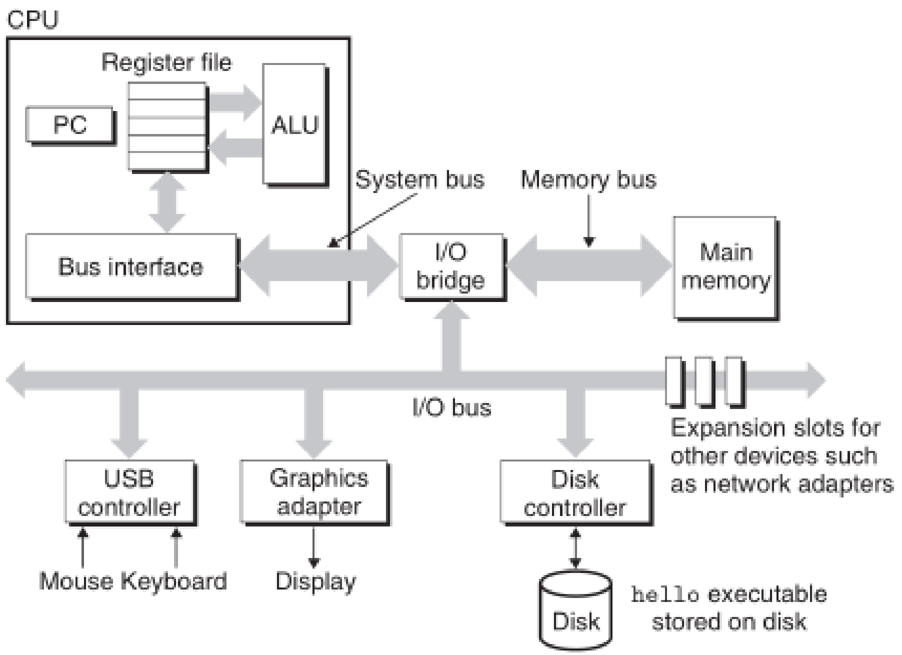
```
## 24
```

$3\times$ `Int64` (8 bytes) $= 24$ bytes

The building blocks of a computer

CPU, or processor: it's the computer's engine. It has 3 parts:

- *Program Counter* (**PC**): keeps the memory address of the current instruction
- **Registers**: tiny but *ultra-fast memory* (e.g.: 16 64-bit registers on Intel i7)
- *Arithmetic/logic unit* (**ALU**): performs operations over the values stored in registers



The building blocks of a computer

CPUs generally operate with a limited number of simple instructions. Any complex calculation translated by the compiler into many simple steps

There are four main types of instructions

- *Load*: copy a value from main memory into a register
- *Store*: copy a value from a register to main memory
- *Operate*: copy 2 values from register to ALU, perform an arithmetic operation on them, store the result in a register
- *Jump*: point the PC to an instruction in another memory address

What happens when we run a program?

```
function hello()
  print("hello, world\n")
end;

hello()
```

```
## hello, world
```

What happens when we run a program?

1) Our function `hello` is compiled: another program will read our human language and convert it to *machine language* (instructions)

- I.e., our code goes from a sequence of bits (`0`s and `1`s) that can be represented as characters in a display to another sequence of bits that represents instructions which the **CPU** can understand

2) The compiled function is placed into the **main memory**

- Compiled code can also be stored in permanent memory (hard drives, SSD, disks). Those are the *executable* files. But before they are executed, they must be loaded into the main memory anyway

What happens when we run a program?

3) We type `hello()` in Julia REPL. Our **input** prompts Julia and the Operating System (OS) to prepare those instructions for execution

- E.g.: by *jumping* the PC to the address where they are stored

4) The **CPU** executes those instructions and **outputs** the results back in the command line, which is then converted into image by a graphics adapter and our display

Moral of the story: the computer spends A LOT of time just moving information around and very little time actually doing "real work!"

Running fast

The computer spends A LOT of time just moving information around and very little time actually doing "real work!" What to do?

Efficient computation needs to

1. Minimize the need to pass information around
    - This is what we do, for example, when we preallocate variables, avoid temporary allocations, vectorize
2. Make information travel faster
    - Faster hardware (not always affordable/available)
    - **Cache memory**!

Cache memory

Physics law: bigger memory = slower memory

- Your hard disk can be $1,000\times$ larger than the main memory but it might take $10,000,000\times$ longer for the processor to read

But the main memory is too big/slow. The CPU actually has its own internal layers of fast memory with different sizes

- Modern processors have registers + 3 levels of cache memory

Cache memory

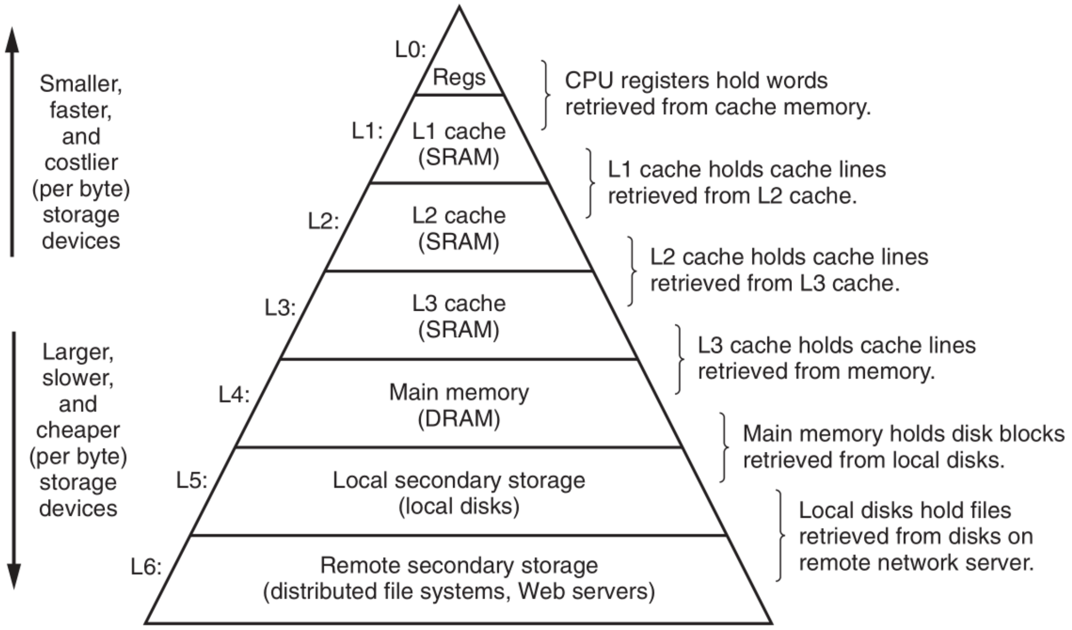


Table 2-4. Cache Parameters of the Skylake Microarchitecture

Level	Capacity / Associativity	Line Size (bytes)	Fastest Latency[1]	Peak Bandwidth (bytes/cyc)	Sustained Bandwidth (bytes/cyc)	Update Policy
First Level Data	32 KB/ 8	64	4 cycle	96 (2x32B Load + 1*32B Store)	~81	Writeback
Instruction	32 KB/8	64	N/A	N/A	N/A	N/A
Second Level	256KB/4	64	12 cycle	64	~29	Writeback
Third Level (Shared L3)	Up to 2MB per core/Up to 16 ways	64	44	32	~18	Writeback

These are Intel specifications →

Writing cache-friendly code

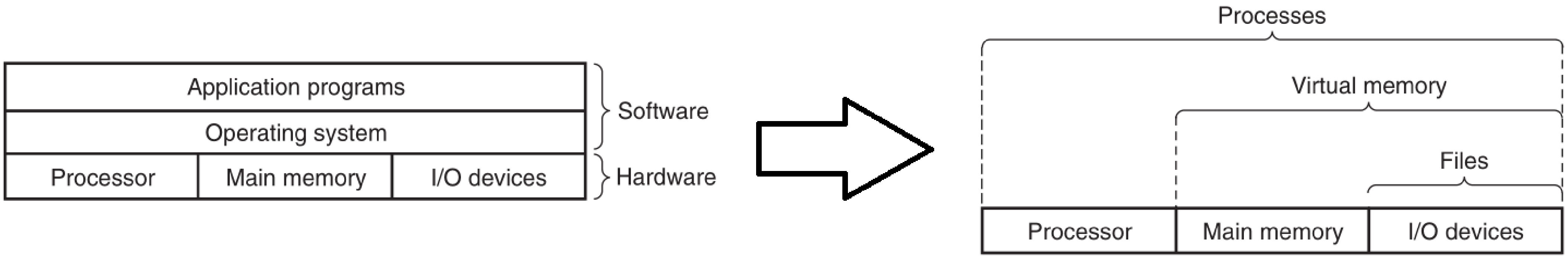
Now that you knowing about the memory hierarchy, you can use that knowledge to write *more efficient, cache-friendly code*. That means:

- Making the common case go fast
  - Programs often spend most of the time in a few loops. **Focus on the inner loops of your core functions**
- Minimize the chances of needing to bring variables from slow memory
  - Reference local variables
  - If you need to reference the same variable in multiple iterations, think about a way order your loops in a way that you make those **repeated references in sequence**

Operating System

There's an important component between your software and the hardware: the *Operating System* (OS)

The OS is a software layer that manages and makes accessible all the hardware components. It transforms those components in simple abstractions that can be easily manipulated with much simpler software instructions



Numerical Arithmetic

Why bother?

Simple arithmetic: $(10^{-20} + 1) - 1 = 10^{-20} + (1 - 1)$, right?

Let's check:

```
x = (1e-20 + 1) - 1;
y = (1e-20) + (1 - 1);
x == y
```

```
## false
```

What happened?!

Why bother?

Welcome to the world of **finite precision**

```
x
```

```
## 0.0
```

```
y
```

```
## 1.0e-20
```

Numbers are an ideal/abstract concept that works perfectly in analytic operations

Computers are a physical machine with limited resources: they can't have infinite precision

Understanding this limitation is crucial in numerical analysis

Number storage

Q: How are numbers physically represented by a computer?

A: In a binary (or Base 2) system. This means each digit can only take on 0 or 1

The system we're most used to is Base 10: each digit can take on 0-9

Q: So which numbers can be represented by a computer?

A: A **subset** *of the rational numbers*

But computers have finite memory and hard disk space, there are infinite rational numbers

This imposes a strict limitation on the storage of numbers

Number storage

Numbers are stored as: $\pm m b^{\pm n}$

- m is the *significand*
- b is the *base*,
- n is the *exponent*

All three are integers

- The *significand* typically gives the significant digits

- The *exponent* scales the number up or down in magnitude

Number storage

The size of numbers a computer can represent is limited by how much space is allocated for a number

Space allocations are usually 64 bits: 53 for m and 11 for n

We've seen these types before

```
println(typeof(5.0))
```

```
## Float64
```

```
println(typeof(5))
```

```
## Int64
```

Number storage

- `Int64` means it is a integer with 64 bits of storage
- `Float64` means it is a floating point number with 64 bits of storage
    - Floating point just means $b^{\pm n}$ can move the decimal point around in the significand

These two types take the same space in memory (64 bits) but are interpreted differently by the processor

The limits of computers

Limitations on storage suggest three facts

Fact 1: There exists a **machine epsilon**: the smallest relative quantity representable by a computer

Machine epsilon is the smallest ϵ such that a machine can always distinguish

$$N + \epsilon > N > N - \epsilon$$

The limits of computers

```
println("Machine epsilon ϵ is $(eps(Float64))")
```

```
## Machine epsilon ϵ is 2.220446049250313e-16
```

```
println("Is 1 + ϵ/2 > 1? $(1.0 + eps(Float64)/2 > 1.0)")
```

```
## Is 1 + ϵ/2 > 1? false
```

The function `eps` gives the distance between 1.0 and the next representable `Float64`

The limits of computers

```
println("The smallest representable number larger than 1.0 is $(nextfloat(1.0))")
```

```
## The smallest representable number larger than 1.0 is 1.0000000000000002
```

```
println("The largest representable number smaller than 1.0 is $(prevfloat(1.0))")
```

```
## The largest representable number smaller than 1.0 is 0.9999999999999999
```

The limits of computers

The machine epsilon changes depending on the amount of storage allocated

```
println("32 bit machine epsilon is $(eps(Float32))")
```

```
## 32 bit machine epsilon is 1.1920929e-7
```

```
println("Is 1 + ϵ/2 > 1? $(Float32(1) + eps(Float32)/2 > 1)")
```

```
## Is 1 + ϵ/2 > 1? false
```

There is a trade-off between precision and storage requirements

- This matters for low-memory systems like GPUs

The limits of computers

Fact 2: There is a **smallest representable number**

```
println("64 bit smallest float is $(floatmin(Float64))")
```

```
## 64 bit smallest float is 2.2250738585072014e-308
```

```
println("32 bit smallest float is $(floatmin(Float32))")
```

```
## 32 bit smallest float is 1.1754944e-38
```

```
println("16 bit smallest float is $(floatmin(Float16))")
```

```
## 16 bit smallest float is 6.104e-5
```

The limits of computers

Fact 3: There is a **largest representable number**

```
println("64 bit largest float is $(floatmax(Float64))")
```

```
## 64 bit largest float is 1.7976931348623157e308
```

```
println("32 bit largest float is $(floatmax(Float32))")
```

```
## 32 bit largest float is 3.4028235e38
```

```
println("16 bit largest float is $(floatmax(Float16))")
```

```
## 16 bit largest float is 6.55e4
```

The limits of computers: time is a flat circle

```
println("The largest 64 bit integer is $(typemax(Int64))")
```

```
## The largest 64 bit integer is 9223372036854775807
```

```
println("Add one to it and we get: $(typemax(Int64)+1)")
```

```
## Add one to it and we get: -9223372036854775808
```

```
println("It loops us around the number line: $(typemin(Int64))")
```

```
## It loops us around the number line: -9223372036854775808
```

The limits of computers

The scale of your problem matters

If a parameter or variable is $>$ `floatmax` or $<$ `floatmin`, you will have a very bad time

Scale numbers appropriately (e.g. millions of dollars, not millionths of cents)

Error

We can only represent a finite number of numbers

This means we will have error in our computations

Error comes in two major forms:

1. Rounding
2. Truncation

Rounding

We will always need to round numbers to the nearest computer representable number. This introduces error

```
println("Half of π is: $(π/2)")
```

```
## Half of π is: 1.5707963267948966
```

The computer gave us a rational number, but $\pi/2$ should be irrational

Truncation

Lots of important numbers are defined by infinite sums $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$

It turns out that computers can't add up infinitely many terms because there is finite space $\rightarrow$ we need to truncate the sum

Why does this matter?

Errors are small, who cares?

You should!

Because errors can propagate and grow as you keep applying an algorithm (e.g. function iteration)

Error example 1

Let's get back to the example we saw earlier

```
x = (1e-20 + 1) - 1
```

```
## 0.0
```

```
y = 1e-20 + (1 - 1)
```

```
## 1.0e-20
```

```
println("The difference is: $(x-y).")
```

```
## The difference is: -1.0e-20.
```

Error example 1

Why did we get $y > x$?

- For $(10^{-20} + 1) - 1$: when we added 10^{-20} to 1, it got rounded away
- For $10^{-20} + (1 - 1)$: here $1 - 1$ was evaluated first and return 0, as we would expect

Adding numbers of greatly different magnitudes does not always works like you would want

Error example 2

Consider a simple quadratic eq. $x^2 - 26x + 1 = 0$ with solution $x = 13 - \sqrt{168}$

```
println("64 bit: 13 - √168 = $(13-sqrt(168))")
```

```
## 64 bit: 13 - √168 = 0.03851860318427924
```

```
println("32 bit: 13 - √168 = $(convert(Float32,13-sqrt(168)))")
```

```
## 32 bit: 13 - √168 = 0.038518604
```

```
println("16 bit: 13 - √168 = $(convert(Float16,13-sqrt(168)))")
```

```
## 16 bit: 13 - √168 = 0.0385
```

Error example 2

Let's check whether they solve the equation

```
x64 = 13-sqrt(168); x32 = convert(Float32,13-sqrt(168)); x16 = convert(Float16,13-sqrt(168));
f(x) = x^2 - 26x + 1;
println("64 bit: $(f(x64))")
```

```
## 64 bit: 7.66053886991358e-15
```

```
println("32 bit: $(f(x32))")
```

```
## 32 bit: 0.0
```

```
println("16 bit: $(f(x16))")
```

```
## 16 bit: 0.0004883
```

Error example 3

Let's just subtract two numbers: `100000.2 - 100000.1`

We know the answer is `0.1`

```
println("100000.2 - 100000.1 is: $(100000.2 - 100000.1)")
```

```
## 100000.2 - 100000.1 is: 0.09999999999126885
```

```
if (100000.2 - 100000.1) == 0.1
    println("and it is equal to 0.1")
else
    println("and it is not equal to 0.1")
end
```

```
## and it is not equal to 0.1
```

Error example 3

Why do we get this error?

Neither of the two numbers can be precisely represented by the machine!

$$
\begin{aligned}
100000.1 &\approx 858993545099345 9 \times 2^{-33} = 100000.0999999999767169356346130 \\
100000.2 &\approx 8589936309986918 \times 2^{-33} = 100000.1999999999534338712692261
\end{aligned}
$$

So their difference won't necessarily be 0.1

Error example 3

There are tools for approximate equality. Remember the `$\approx$` operator we saw last lecture? You can use it

```
(100000.2 - 100000.1) ≈ 0.1 # You type \approx then hit TAB
```

```
## true
```

This is equivalent to:

```
isapprox(100000.2 - 100000.1, 0.1)
```

```
## true
```

This matters, particularly when you're trying to evaluate logical expressions of equality

Course roadmap

This concludes the first lecture of Unit 2. Up next:

1. Intro to Scientific Computing
2. **Numerical operations and representations**
    1. Numerical arithmetic
    2. **Numerical differentiation and integration** $\leftarrow$
    3. Function approximation
3. Systems of equations
4. Optimization
5. Structural estimation