Table of Contents

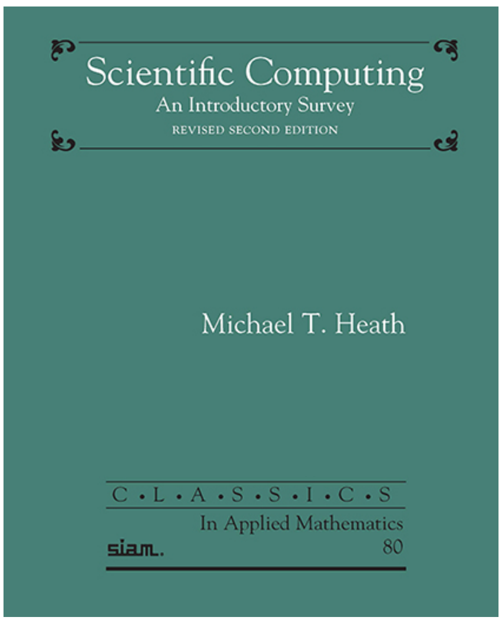
About this course (10min)

What is scientific computing?

Scientific computing is the collection of tools, techniques and theories required to solve on a computer the mathematical models of problems in science and engineering.

– Gene H. Golub and James M. Ortega

Textbook



Chapters

1. Scientific Computing
2. Systems of linear equations
3. Linear least squares
4. Eigenvalue problems
5. Nonlinear equations
6. Optimization
7. Interpolation
8. ~~Numerical integration and differentiation~~
9. ~~Initial value problems for ordinary differential equations~~
10. ~~Boundary value problems for ordinary differential equations~~
11. ~~Partial differential equations~~
12. Fast fourier transform
13. Random numbers and stochastic simulation

Lecture 1: Understanding our computing devices

✎ What is inside a computer? (40min)

Get hardware information (Linux)

</> Get hardware information

$ lscpu

$ lsmem

$ top

show_cpuinfo = ☐

```
if show_cpuinfo run(`lscpu`) end
```

show_meminfo = ☐

```
if show_meminfo run(`lsmem`) end
```

show_processinfo = ☐

```
if show_processinfo run(`top -n 1 -b`) end
```

Number system (20min)

Integers

Integer type range is

```
-9223372036854775808
```

```julia
typemin(Int64)
```

```
"1000000000000000000000000000000000000000000000000000000000000000"
```

```julia
bitstring(typemin(Int64))
```

```
"1000000000000000000000000000000000000000000000000000000000000001"
```

```julia
bitstring(typemin(Int64) + 1)
```

```
"0000000000000000000000000000000000000000000000000000000000000000"
```

```julia
bitstring(0)
```

☐ Show the bitstring for -1 (binded to `show_minus1`)

```julia
if show_minus1 bitstring(-1) end
```

```
9223372036854775807
```

```julia
typemax(Int64)
```

```
"0111111111111111111111111111111111111111111111111111111111111111"
```

```julia
bitstring(typemax(Int64))
```

Floating point numbers

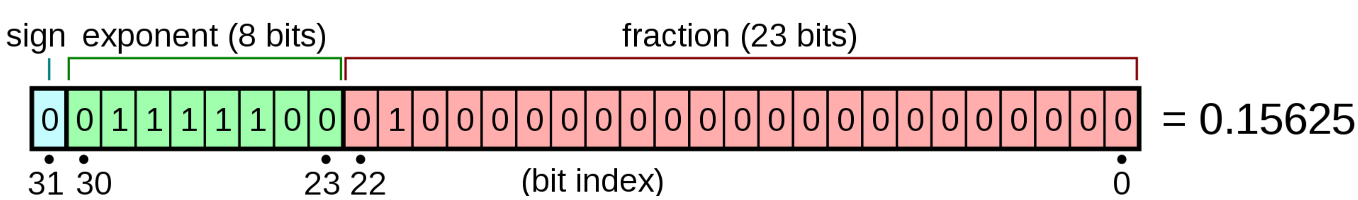


image source: https://en.wikipedia.org/wiki/IEEE_754

```
"00111110001000000000000000000000"
```

```julia
bitstring(0.15625f0)
```

```
-3
```

```julia
exponent(0.15625f0)
```

The significant is in range [1, 2)

```
1.25f0
```

```julia
significand(0.15625f0)  # the fraction
```

```
Inf
```

```julia
typemax(Float64)
```

```
"0111111111110000000000000000000000000000000000000000000000000000"
```

```julia
bitstring(Inf)
```

```
1.7976931348623157e308
```

```julia
prevfloat(Inf)
```

```
-Inf
```

```julia
typemin(Float64)
```

```
"1111111111110000000000000000000000000000000000000000000000000000"
```

```julia
bitstring(-Inf)
```

```
-5.0e-324
```

```julia
prevfloat(0.0)
```

```
5.0e-324
```

```julia
nextfloat(0.0)
```

```
NaN
```

```julia
Inf-Inf
```

```
NaN
```

```julia
0 * NaN
```

```
"0111111111111000000000000000000000000000000000000000000000000000"
```

```julia
bitstring(NaN)
```

The distribution of floating point numbers

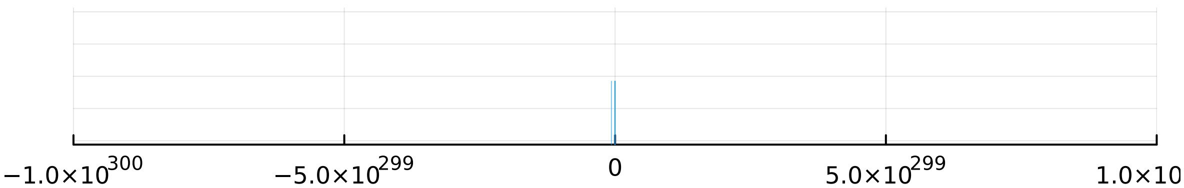
The distribution of floating point numbers

npoints = 1002

```julia
xs = filter(!isnan, reinterpret(Float64, rand(Int64, npoints)));
```

From the linear scale plot, you will see data concentrated around 0 (each vertical bar is a sample)

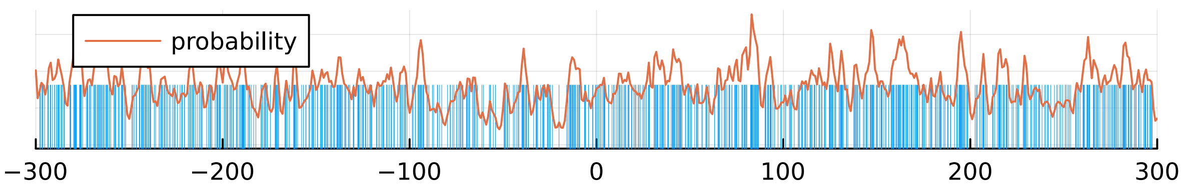
```julia
using Plots
```




```julia
scatter(xs, zeros(length(xs)), xlim=(-1e300, 1e300), label="", size=(600, 100),
yaxis=:off, markersize=30, markershape=:vline)
```

If we use logarithmic x-axis

```
smearing_factor =  1.0
```




```julia
let
    logxs = sign.(xs) .* log10.(abs.(xs))
    ax = scatter(logxs, zeros(length(xs)), xlim=(-300, 300), label="", size=(600,
    100), yaxis=:off, markersize=30, markershape=:vline)
    a = -300:300
    # smoothen the distribution with Lorentz function
    m = 1/π/smearing_factor ./ (((a' .- logxs) ./ smearing_factor) .^ 2 .+ 1)
    plot!(ax, a, dropdims(sum(m, dims=1), dims=1); label="probability")
end
```

Estimating the computing power of your devices (20min)

Example 1: Matrix multiplication

$$C_{ik} = \sum_j A_{ij} \times B_{jk}$$

Let the matrix size be `n x n`, the peudocode for general matrix multiply (GEMM) is

```
for i=1:n
    for j=1:n
        for k=1:n
            C[i, k] = A[i, j] * B[j, k]
        end
    end
end
```

GEMM is CPU bottlenecked

matrix_size = 1000

benchmark_example1 = ☐

Loading the package for benchmarking

```julia
using BenchmarkTools
```

Loading the matrix multiplication function

```julia
using LinearAlgebra: mul!
```

```julia
if benchmark_example1
    let
        # creating random vectors with normal distribution/zero elements
        A = randn(Float64, matrix_size, matrix_size)
        B = randn(Float64, matrix_size, matrix_size)
        C = zeros(Float64, matrix_size, matrix_size)
        @benchmark mul!($C, $A, $B)
    end
end
```

Calculating the **floating point operations per second**

✎ FLOPS for computing GEMM

the number of floating point operations / the number of seconds

Example 2: axpy

axpy! is memory I/O bottlenecked

```
axpy! (generic function with 1 method)
```

```
function axpy!(a::Real, x::AbstractVector, y::AbstractVector)
    @assert length(x) == length(y) "the input size of x and y mismatch, got
    $(length(x)) and $(length(y))"
    @inbounds for i=1:length(x)
        y[i] += a * x[i]
    end
    return y
end
```

axpy_vector_size = 1000

benchmark_axpy = ☐

```
if benchmark_axpy
    let
        x = randn(Float64, axpy_vector_size)
        y = randn(Float64, axpy_vector_size)
        @benchmark axpy!(2.0, $x, $y)
    end
end
```

✎ FLOPS for computing axpy

the number of floating point operations / the number of seconds

Example 3: modified axpy

```
bad_axpy! (generic function with 1 method)
```

```
function bad_axpy!(a::Real, x::AbstractVector, y::AbstractVector,
indices::AbstractVector{Int})
    @assert length(x) == length(y) == length(indices) "the input size of x and y
    mismatch, got $(length(x)), $(length(y)) and $(length(indices))"
    @inbounds for i in indices
        y[i] += a * x[i]
    end
    return y
end
```

I will show this function is latency bottlenecked

bad_axpy_vector_size = 1000

benchmark_bad_axpy = ☐

```
@xbind benchmark_bad_axpy CheckBox()
```

```
using Random
```

```
if benchmark_bad_axpy
    let
        x = randn(Float64, bad_axpy_vector_size)
        y = randn(Float64, bad_axpy_vector_size)
        indices = randperm(bad_axpy_vector_size)
        @benchmark bad_axpy!(2.0, $x, $y, $indices)
    end
end
```

✎ FLOPS for computing bad axpy

the number of floating point operations / the number of seconds

Programming on a device （30min）

You program are compiled to binary

```
	.text
	.file	"axpy!"
	.globl	"julia_axpy!_5041"              # -- Begin function julia_axpy!_5041
	.p2align	4, 0x90
	.type	"julia_axpy!_5041",@function
"julia_axpy!_5041":                     # @"julia_axpy!_5041"
; ┌ @ /home/leo/jcode/ModernScientificComputing/notebooks/1.understanding-our-computi
	.cfi_startproc
# %bb.0:                                # %top
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset %rbp, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register %rbp
	pushq	%r15
	pushq	%r14
	pushq	%rbx
	andq	$-32, %rsp
```

```
with_terminal() do
    x, y = randn(10), randn(10)
    @code_native axpy!(2.0, x, y)
end
```

Let us check an easier one

```
oneton (generic function with 1 method)
```

```julia
function oneton(n::Int)
    res = zero(n)
    for i = 1:n
         res+=i
    end
     return res
end
```

```
    .text
    .file   "oneton"
    .globl  julia_oneton_5153               # -- Begin function julia_oneton_5153
    .p2align    4, 0x90
    .type   julia_oneton_5153,@function
julia_oneton_5153:                      # @julia_oneton_5153
; ┌ @ /home/leo/jcode/ModernScientificComputing/notebooks/1.understanding-our-computi
    .cfi_startproc
# %bb.0:                                # %top
; │ @ /home/leo/jcode/ModernScientificComputing/notebooks/1.understanding-our-computi
; │┌ @ range.jl:5 within `Colon`
; ││┌ @ range.jl:397 within `UnitRange`
; │││┌ @ range.jl:404 within `unitrange_last`
    testq   %rdi, %rdi
; │└└└
    jle .LBB0_1
# %bb.2:                                # %L17.preheader
; │ @ /home/leo/jcode/ModernScientificComputing/notebooks/1.understanding-our-computi
```

```julia
with_terminal() do
     @code_native oneton(10)
end
```

An instruct has a binary correspondence: check the online decoder

Measuring the performance

```julia
using Profile
```

```
profile_axpy = ☐
```

```julia
if profile_axpy
     with_terminal() do
         # clear previous profiling data
         Profile.init(; n=10^6, delay=0.001)
         x, y = randn(100000000), randn(100000000)
         @profile axpy!(2.0, x, y)
         Profile.print()
     end
end
```

✎ How does profiling work?

* function call stack
* two approaches: instrumentation and sampling

Summarize

1. understanding the components of our computing devices
2. the bottlenecks of our computing devices
3. how to get our program compiled and executed
4. how to measure the performance of a program with profiling

Next lecture

We have have a coding seminar. I will show you some cheatsheets about

- Linux operation system
- Vim
- Git
- SSH
- Julia installation Guide

Please bring your laptops and get your hands dirty! If you are already an expert, please let me know, I need some help in preparing the cheatsheets.

Pre-reading

Strong recommended course: missing-semester

- 1/13: Course overview + the shell (1 - expert)
- 1/14: Shell Tools and Scripting (4 - basic)
- 1/15: Editors (Vim) (2 - basic)
- 1/16: Data Wrangling
- 1/21: Command-line Environment
- 1/22: Version Control (Git) (3 - expert)
- 1/23: Debugging and Profiling
- 1/27: Metaprogramming (build systems, dependency management, testing, CI)
- 1/28: Security and Cryptography
- 1/29: Potpourri
- 1/30: Q&A

Note

- Yellow backgrounded lectures are required by AMAT5315
- (n - basic) is the reading order and the level of familiarity