```julia
using Pkg, Luxor
```

```julia
try
    using PlutoLecturing
catch
    Pkg.develop(path="https://github.com/GiggleLiu/PlutoLecturing.jl.git")
    using PlutoLecturing
end
```

Table of Contents

Announcement

An Introduction to the Julia programming language
- A survey

What is JuliaLang?
- A modern, open-source, high performance programming lanaguage
- Reference

The two language problem
- Executing a C program
- Executing a Pyhton Program
- Two languages, e.g. Python & C/C++?

Julia's solution
- Julia compiling stages
- The key ingredients of performance

Julia's type system
- Numbers
- Case study: Vector element type and speed

Multiple dispatch
- Multiple dispatch is more powerful than object-oriented programming!
- Summary

Tuple, Array and broadcasting

Julia package development
- Unit Test
- Case study: Create a package like HappyMolecules

Homework

Live coding

```
TableOfContents(depth=2)
```

Present

Announcement

HKUST-GZ Zulip is online!

ZULIP

https://zulip.hkust-gz.edu.cn

It is **open source**, **self-hosted (5TB storage)** , **history kept & backuped** and allows you to detect community and services.

An Introduction to the Julia programming language

A survey

What programming language do you use? Do you have any pain point about this language?

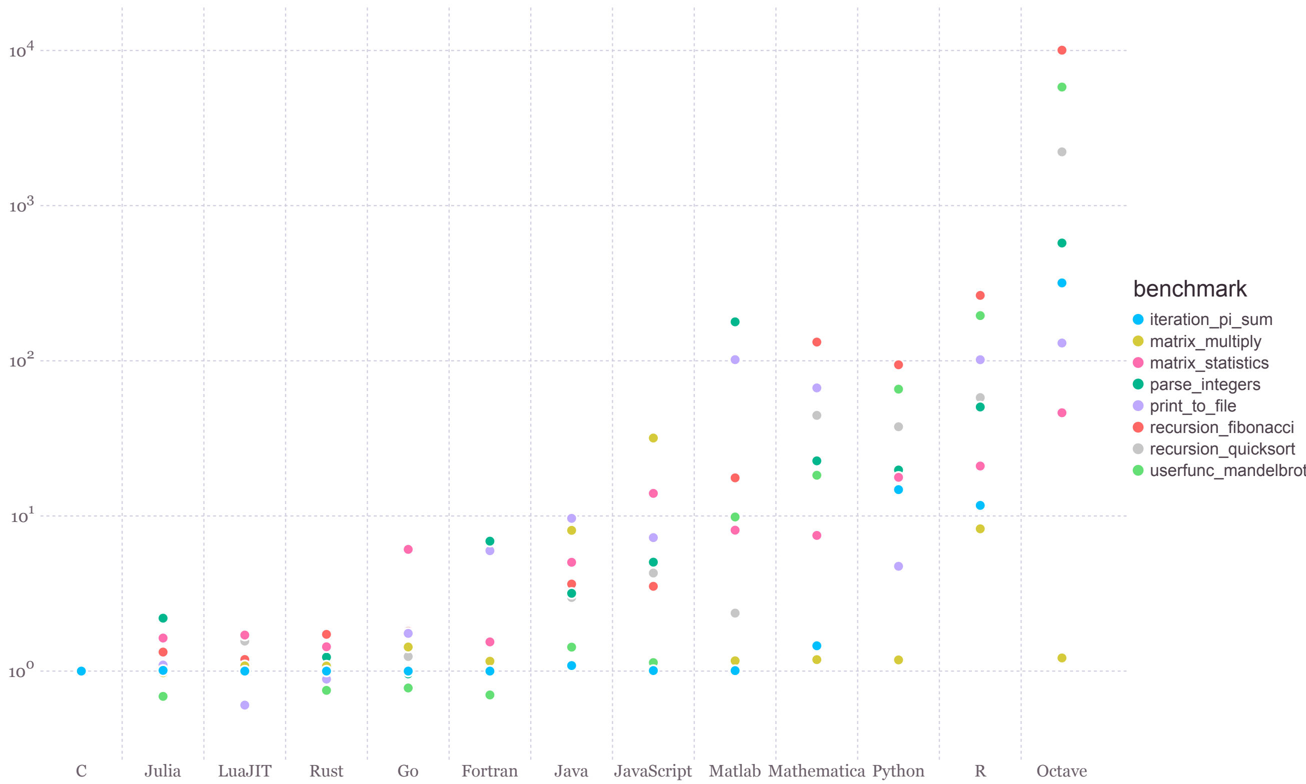
What is JuliaLang?

A modern, open-source, high performance programming lanaguage

JuliaLang was born in 2012 in MIT, now is maintained by Julia Computing Inc. located in Boston, US. Founders are Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah.

JuliaLang is open-source, its code is maintained on Github(https://github.com/JuliaLang/julia) and it open source LICENSE is MIT. Julia packages can be found on JuliaHub, most of them are open-source.

It is designed for speed



Reference

arXiv:1209.5145

Julia: A Fast Dynamic Language for Technical Computing – Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman

Dynamic languages have become popular for scientific computing. They are generally considered highly productive, but lacking in performance. This paper presents Julia, a new dynamic language for technical computing, designed for performance from the beginning by adapting and extending modern programming language techniques. A design based on generic functions and a rich type system simultaneously enables an expressive programming model and successful type inference, leading to good performance for a wide range of programs. This makes it possible for much of the Julia library to be written in Julia itself, while also incorporating best-of-breed C and Fortran libraries.

Terms explained

- *dynamic programming language*: In computer science, a *dynamic programming language* is a class of high-level programming languages, which at runtime execute many common programming behaviours that static programming languages perform during compilation. These behaviors could include an extension of the program, by adding new code, by extending objects and definitions, or by modifying the type system.
- *type*: In a programming language, a *type* is a description of a set of values and a set of allowed operations on those values.
- *generic function*: In computer programming, a *generic function* is a function defined for polymorphism.
- *type inference*: *Type inference* refers to the automatic detection of the type of an expression in a formal language.

The two language problem

Executing a C program

C code is typed.

```
Process(`cat clib/demo.c`, ProcessExited(0))
```

```
#include <stddef.h>
int c_factorial(size_t n) {
    int s = 1;
    for (size_t i=1; i<=n; i++) {
        s *= i;
    }
    return s;
}
```

```julia
# A notebook utility to run code in a terminal style
with_terminal() do
    # display the file
    run(`cat clib/demo.c`)
end
```

C code needs to be compiled

```
Process(`gcc clib/demo.c -fPIC -O3 -msse3 -shared -o clib/demo.so`, ProcessExited(0))
```

```julia
# compile to a shared library by piping C_code to gcc;
# (only works if you have gcc installed)
run(`gcc clib/demo.c -fPIC -O3 -msse3 -shared -o clib/demo.so`)
```

```
Process(`ls clib`, ProcessExited(0))
```

```
demo.c
demo.so
```

```julia
with_terminal() do
    # list all files
    run(`ls clib`)
end
```

One can use `Libdl` package to open a shared library

```julia
# for opening a shared library file (*.so), with zero run-time overhead
using Libdl
```

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

```julia
# @ccall is a julia macro
# a macro is a program for generating programs, just like the template in C++
# In Julia, we use `::` to specify the type of a variable
c_factorial(x) = Libdl.@ccall "clib/demo".c_factorial(x::Csize_t)::Int
```

Typed code may overflow, but is fast!

```
3628800
```

```julia
c_factorial(10)
```

0

```
c_factorial(1000)
```

```
using BenchmarkTools
```

benchmark_ccode = ☑

```
BenchmarkTools.Trial: 10000 samples with 89 evaluations.
 Range (min … max):  809.427 ns …  1.217 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     810.281 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   817.491 ns ± 23.064 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
  809 ns          Histogram: log(frequency) by time        903 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```
if benchmark_ccode @benchmark c_factorial(1000) end
```

learn more about calling C code in Julia

Discussion: not all type specifications are nessesary.

Executing a Pyhton Program

Dynamic programming language does not require compiling

```
using PyCall
```

```
# py"..." is a string literal, it is defined as a special macro: @py_str
py"""
def factorial(n):
    x = 1
    for i in range(1, n+1):
        x = x * i
    return x
"""
```

```
#py"factorial"(1000)
```

Dynamic typed language is more flexible, but slow!

9223372036854775807

```
# `typemax` to get the maximum value
typemax(Int)
```

🤔

benchmark_pycode = ☑

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  275.389 μs … 677.122 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     285.648 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   286.375 μs ±   8.127 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

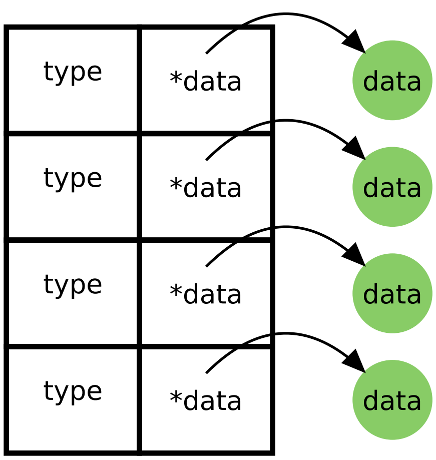
```
  275 μs           Histogram: frequency by time          311 μs <

 Memory estimate: 7.45 KiB, allocs estimate: 11.
```

```
if benchmark_pycode @benchmark $(py"factorial")(1000) end
```

The reason why dynamic typed language is slow is related to caching.

Dynamic typed language uses `Box(type, *data)` to represent an object.



Cache miss!

Two languages, e.g. Python & C/C++?

From the maintainance's perspective

- Requires a build system and configuration files,
- Not easy to train new developers.

There are many problems can not be vectorized

- Monte Carlo method and simulated annealing method,
- Generic Tensor Network method: the tensor elements has tropical algebra or finite field algebra,
- Branching and bound.

Julia's solution

Julia compiling stages

NOTE: I should open a Julia REPL now!

1. You computer gets a Julia program

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

```julia
function jlfactorial(n)
    x = 1
    for i in 1:n
        x = x * i
    end
    return x
end
```

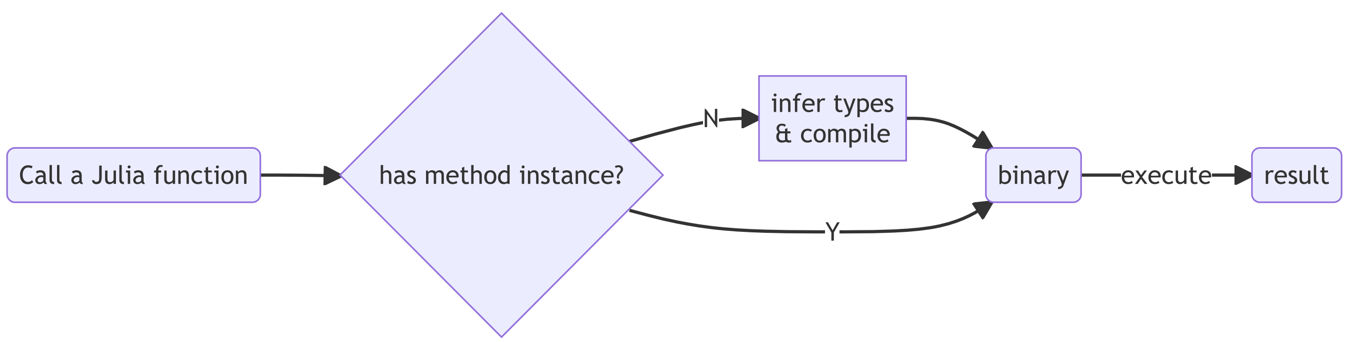
Method instance is a compiled binary of a function for specific input types. When the function is written, the binary is not yet generated.

```julia
using MethodAnalysis  # a package to analyse functions
```

```
[]
```

```julia
methodinstances(jlfactorial)
```

2. When calling a function, the Julia compiler infers types of variables on an intermediate representation (IR)



One can use `@code_warntype` or `@code_typed` to show this intermediate representation.

```
MethodInstance for Main.var"workspace#691".jlfactorial(::Int64)
  from jlfactorial(n) in Main.var"workspace#691" at /home/user1/ModernScientificCompu
Arguments
  #self#::Core.Const(Main.var"workspace#691".jlfactorial)
  n::Int64
Locals
  @_3::Union{Nothing, Tuple{Int64, Int64}}
  x::Int64
  i::Int64
Body::Int64
1 ─       (x = 1)
│   %2  = (1:n)::Core.PartialStruct(UnitRange{Int64}, Any[Core.Const(1), Int64])
│         (@_3 = Base.iterate(%2))
│   %4  = (@_3 === nothing)::Bool
│   %5  = Base.not_int(%4)::Bool
└──       goto #4 if not %5
2 ┄ %7  = @_3::Tuple{Int64, Int64}
│         (i = Core.getfield(%7, 1))
```

```julia
with_terminal() do
    @code_warntype jlfactorial(10)  # or @code_typed without warning
end
```

`::` means type assertion in Julia.

Sometimes, type can not be uniquely determined at the runtime. This is called "type unstable".

```
MethodInstance for (::Main.var"workspace#10".var"#unstable#5")(::Int64)
  from (::Main.var"workspace#10".var"#unstable#5")(x) in Main.var"workspace#10" at /hom
Arguments
  #self#::Core.Const(Main.var"workspace#10".var"#unstable#5"())
  x::Int64
Body::Union{Float64, Int64}
1 ─ %1 = (x > 3)::Bool
└──      goto #3 if not %1
2 ─      return 1.0
3 ─      return 3
```

```julia
with_terminal() do
    unstable(x) = x > 3 ? 1.0 : 3
    @code_warntype unstable(4)
end
```

3. The typed program is then compiled to LLVM IR

LLVM is a set of compiler and toolchain technologies that can be used to develop a front end for any programming language and a back end for any instruction set architecture. LLVM is the backend of multiple languages, including Julia, Rust, Swift and Kotlin.

```
;  @ /home/user1/ModernScientificComputing/notebooks/3.julia.jl#==#d2429055-58e9-4d84
define i64 @julia_jlfactorial_18226(i64 signext %0) #0 {
top:
;  @ /home/user1/ModernScientificComputing/notebooks/3.julia.jl#==#d2429055-58e9-4d84
; ┌ @ range.jl:5 within `Colon`
; │┌ @ range.jl:393 within `UnitRange`
; ││┌ @ range.jl:400 within `unitrange_last`
     %.inv = icmp sgt i64 %0, 0
     %. = select i1 %.inv, i64 %0, i64 0
; └└└
  br i1 %.inv, label %L18.preheader, label %L35

L18.preheader:                                    ; preds = %top
;  @ /home/user1/ModernScientificComputing/notebooks/3.julia.jl#==#d2429055-58e9-4d84
  %min.iters.check = icmp ult i64 %., 16
  br i1 %min.iters.check, label %L18, label %vector.ph

vector.ph:                                        ; preds = %L18.preheader
```

```julia
with_terminal() do
    @code_llvm jlfactorial(10)
end
```

4. LLVM IR does some optimization, and then compiled to binary code.

```
	.text
	.file	"jlfactorial"
	.section	.rodata.cst8,"aM",@progbits,8
	.p2align	3                               # -- Begin function julia_jlfactorial
.LCPI0_0:
	.quad	1                               # 0x1
.LCPI0_2:
	.quad	4                               # 0x4
.LCPI0_3:
	.quad	8                               # 0x8
.LCPI0_4:
	.quad	12                              # 0xc
.LCPI0_5:
	.quad	16                              # 0x10
	.section	.rodata.cst32,"aM",@progbits,32
	.p2align	5
.LCPI0_1:
	.quad	1                               # 0x1
```

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

Aftering calling a function, a method instance will be generated.

```
0
```

```julia
jlfactorial(1000)
```

```
[MethodInstance for Main.var"workspace#691".jlfactorial(::Int64)]
```

```julia
methodinstances(jlfactorial)
```

A new method will be generatd whenever there is a new type as the input.

```
3628800
```

```julia
jlfactorial(UInt32(10))
```

```
[MethodInstance for Main.var"workspace#691".jlfactorial(::Int64), MethodInstance for Main
```

```julia
methodinstances(jlfactorial)
```

Dynamically generating method instances is also called Just-in-time compiling (JIT), the secret why Julia is fast!

benchmark_jlcode = ☑

```
BenchmarkTools.Trial: 10000 samples with 323 evaluations.
 Range (min … max):  266.771 ns … 382.895 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     267.257 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   269.565 ns ±   6.856 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
  267 ns        Histogram: log(frequency) by time        293 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```
if benchmark_jlcode @benchmark jlfactorial(x) setup=(x=1000) end
```

The key ingredients of performance

- Rich **type** information, provided naturally by **multiple dispatch**;
- aggressive code specialization against **run-time** types;
- **JIT** compilation using the **LLVM** compiler framework.

Julia's type system

1. Abstract types, which may have declared subtypes and supertypes (a subtype relation is declared using the notation Sub <: Super)
2. Composite types (similar to C structs), which have named fields and declared supertypes
3. Bits types, whose values are represented as bit strings, and which have declared supertypes
4. Tuples, immutable ordered collections of values
5. Union types, abstract types constructed from other types via set union

Numbers

Type hierachy in Julia is a tree (without multiple inheritance)

```
Number
├─ Complex{T<:Real}
├─ Real
│  ├─ AbstractFloat
│  │  ├─ BigFloat
│  │  ├─ Float16
│  │  ├─ Float32
│  │  └─ Float64
│  ├─ AbstractIrrational
│  │  └─ Irrational{sym}
│  ├─ FixedPointNumbers.FixedPoint{T<:Integer, f}
│  │  ├─ FixedPointNumbers.Fixed{T<:Signed, f}
│  │  └─ FixedPointNumbers.Normed{T<:Unsigned, f}
│  ├─ Integer
│  │  ├─ Bool
│  │  ├─ Signed
│  │  │  ├─ BigInt
│  │  │  ├─ Int128
│  │  │  ├─ Int16
│  │  │  ├─ Int32
│  │  │  ├─ Int64
│  │  │  └─ Int8
│  │  └─ Unsigned
│  │     ├─ UInt128
│  │     ├─ UInt16
│  │     ├─ UInt32
│  │     ├─ UInt64
│  │     └─ UInt8
│  └─ Rational{T<:Integer}
├─ TropicalNumbers.CountingTropical{T, CT}
└─ TropicalNumbers.Tropical{T}
```

```julia
PlutoLecturing.print_type_tree(Number)
```

```
[Complex, Real, CountingTropical, Tropical]
```

```julia
subtypes(Number)
```

```
AbstractFloat
```

```julia
supertype(Float64)
```

```
true
```

```julia
AbstractFloat <: Real
```

Abstract types does not have fields, while composite types have

```
true
```

```julia
Base.isabstracttype(Number)
```

```
true
```

```julia
# concrete type is more strict than composit
Base.isconcretetype(Complex{Float64})
```

```
ArgumentError: type does not have a definite number of fields

 1. fieldcount(::Any) @ reflection.jl:804
 2. fieldnames(::DataType) @ reflection.jl:185
 3. top-level scope @ Local: 1  [inlined]
```

```julia
fieldnames(Number)
```

```
(:re, :im)
```

```julia
fieldnames(Complex)
```

We have only finite primitive types on a machine, they are those supported natively by computer instruction.

```
true
```

```julia
Base.isprimitivetype(Float64)
```

`Any` is a super type of any other type

```
true
```

```julia
Number <: Any
```

A type contains two parts: type name and type parameters

```
ComplexF64 (alias for Complex{Float64})
```

```julia
# TypeName{type parameters...}
Complex{Float64}  # a commplex number with real and imaginary parts being Float64
```

ComplexF64 is a bits type, it has fixed size.

```
true
```

```julia
isbitstype(Complex{Float64})
```

```
8
```

```julia
sizeof(Complex{Float32})
```

```
16
```

```julia
sizeof(Complex{Float64})
```

But `Complex{BigFloat}` is not

```
16
```

```julia
sizeof(Complex{BigFloat})
```

```
false
```

```julia
isbitstype(Complex{BigFloat})
```

The size of Complex{BigFloat} is not true! It returns the pointer size!

A type can be neither abstract nor concrete.

To represent a complex number with its real and imaginary parts being floating point numbers

```
Complex{<:AbstractFloat}
```

```julia
Complex{<:AbstractFloat}
```

```
true
```

```julia
Complex{Float64} <: Complex{<:AbstractFloat}
```

```
false
```

```julia
Base.isabstracttype(Complex{<:AbstractFloat})
```

```
false
```

```julia
Base.isconcretetype(Complex{<:AbstractFloat})
```

We use `Union` to represent the union of two types

```
true
```

```julia
Union{AbstractFloat, Complex} <: Number
```

```
false
```

```julia
Union{AbstractFloat, Complex} <: Real
```

NOTE: it is similar to multiple inheritance, but `Union` can not have subtype!

You can make an alias for a type name if you think it is too long

```
Union{Complex{T}, T} where T<:AbstractFloat
```

```julia
FloatAndComplex{T} = Union{T, Complex{T}} where T<:AbstractFloat
```

Case study: Vector element type and speed

Any type vector is flexible. You can add any element into it.

vany = []

```
vany = Any[]  # same as vany = []
```

Vector{Any} (alias for Array{Any, 1})

```
typeof(vany)
```

["a"]

```
push!(vany, "a")
```

["a", 1]

```
push!(vany, 1)
```

Fixed typed vector is more restrictive.

vfloat64 = []

```
vfloat64 = Float64[]
```

Vector{Float64} (alias for Array{Float64, 1})

```
vfloat64 |> typeof
```

```
MethodError: Cannot `convert` an object of type String to an object of type Float64
Closest candidates are:
convert(::Type{T}, !Matched::ColorTypes.Gray24) where T<:Real at
~/.julia/packages/ColorTypes/1dGw6/src/conversions.jl:114
convert(::Type{T}, !Matched::ColorTypes.Gray) where T<:Real at
~/.julia/packages/ColorTypes/1dGw6/src/conversions.jl:113
convert(::Type{T}, !Matched::T) where T<:Number at number.jl:6
...
```

1. push!(::Vector{Float64}, ::String) @ array.jl:1057
2. top-level scope @ Local: 1 [inlined]

```
push!(vfloat64, "a")
```

But type stable vectors are faster!

run_any_benchmark = ☑

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  150.204 μs …   3.752 ms  ┊ GC (min … max): 0.00% … 94.94%
 Time  (median):     168.311 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   183.749 μs ± 183.274 μs  ┊ GC (mean ± σ):  5.13% ±  4.91%
```

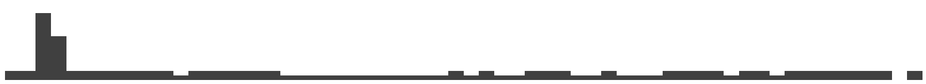
```
 150 μs           Histogram: frequency by time          218 μs <

 Memory estimate: 156.25 KiB, allocs estimate: 10000.
```

```
if run_any_benchmark
    let biganyv = collect(Any, 1:2:20000)
        @benchmark for i=1:length($biganyv)
            $biganyv[i] += 1
        end
    end
end
```

```
run_float_benchmark = ☑
```

```
BenchmarkTools.Trial: 10000 samples with 10 evaluations.
 Range (min … max):  1.215 μs …  3.756 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     1.249 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.261 μs ± 89.968 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```



```
 1.22 μs        Histogram: frequency by time        1.88 μs <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```
if run_float_benchmark
    let bigfloatv = collect(Float64, 1:2:20000)
        @benchmark for i=1:length($bigfloatv)
            $bigfloatv[i] += 1
        end
    end
end
```

Multiple dispatch

```
# the definition of an abstract type
# L is the number of legs
abstract type AbstractAnimal{L} end
```

```
# the definition of a concrete type
struct Dog <: AbstractAnimal{4}
    color::String
end
```

`<:` is the symbol for sybtyping, `A <: B` means A is a subtype of B.

```julia
struct Cat <: AbstractAnimal{4}
    color::String
end
```

```julia
struct Cock <: AbstractAnimal{2}
    gender::Bool
end
```

```julia
struct Human{FT <: Real} <: AbstractAnimal{2}
    height::FT
    function Human(height::T) where T <: Real
        if height <= 0 || height > 300
            error("The tall of a Human being must be in range 0~300, got $(height)")
        end
        return new{T}(height)
    end
end
```

One can implement the same function on different types

The most general one as the fall back method

```
fight (generic function with 4 methods)
```

```julia
fight(a::AbstractAnimal, b::AbstractAnimal) = "draw"
```

```
"draw"
```

```julia
fight(Cock(true), Cat("red"))
```

The most concrete method is called

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

```julia
fight(dog::Dog, cat::Cat) = "win"
```

```
"win"
```

```julia
fight(Dog("blue"), Cat("white"))
```

```
fight (generic function with 5 methods)
```

```julia
fight(hum::Human, a::AbstractAnimal) = "win"
```

```
fight (generic function with 2 methods)
```

```julia
fight(hum::Human, a::Union{Dog, Cat}) = "loss"
```

```
"loss"
```

```julia
fight(Human(180), Cat("white"))
```

Be careful about the ambiguity error!

```
fight (generic function with 6 methods)
```

```julia
fight(hum::AbstractAnimal, a::Human) = "loss"
```

The combination of two types.

```
"loss"
```

```julia
fight(Human(170), Human(180))
```

define_human_fight = ☑

```
fight (generic function with 6 methods)
```

```julia
if define_human_fight
    fight(hum::Human{T}, hum2::Human{T}) where T<:Real = hum.height > hum2.height ?
    "win" : "loss"
end
```

Quiz: How many method instances are generated for fight so far?

```
[MethodInstance for Main.var"workspace#249".fight(::Human{Int64}, ::Cat), MethodInstance
```

```julia
methodinstances(fight)
```

A final comment: do not abuse the type system, otherwise the main memory might explode for generating too many functions.

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

```julia
# NOTE: this is not the best way of implementing fibonacci sequencing
fib(x::Int) = x <= 2 ? 1 : fib(x-1) + fib(x-2)
```

run_dynamic_benchmark = ☑

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  17.938 μs … 44.094 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     17.950 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   18.136 μs ±  1.130 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
  17.9 μs       Histogram: log(frequency) by time      25.3 μs <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```julia
if run_dynamic_benchmark @benchmark fib(20) end
```

A "zero" cost implementation

Val()

```julia
Val(3.0) # just a type
```

addup (generic function with 1 method)

```julia
addup(::Val{x}, ::Val{y}) where {x, y} = Val(x + y)
```

f (generic function with 1 method)

```julia
f(::Val{x}) where x = addup(f(Val(x-1)), f(Val(x-2)))
```

f (generic function with 2 methods)

```julia
f(::Val{1}) = Val(1)
```

f (generic function with 3 methods)

```julia
f(::Val{2}) = Val(1)
```

run_static_benchmark = ☑

```julia
@xbind run_static_benchmark CheckBox()
```

```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  1.157 ns … 10.639 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     1.162 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.175 ns ±  0.281 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

  ▁ ▂ ▃ █ █ █ ▅ ▆ ▄ ▃ ▂ ▃ ▂ ▂ ▁ ▁ ▂
  1.16 ns        Histogram: frequency by time        1.17 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```julia
if run_static_benchmark @benchmark f(Val(20)) end
```

However, this violates the Performance Tips, since it transfers the run-time to compile time.

Multiple dispatch is more powerful than object-oriented programming!

Implement addition in Python.

```python
class X:
  def __init__(self, num):
    self.num = num

  def __add__(self, other_obj):
    return X(self.num+other_obj.num)

  def __radd__(self, other_obj):
    return X(other_obj.num + self.num)

  def __str__(self):
    return "X = " + str(self.num)

class Y:
  def __init__(self, num):
    self.num = num

  def __radd__(self, other_obj):
    return Y(self.num+other_obj.num)

  def __str__(self):
    return "Y = " + str(self.num)

print(X(3) + Y(5))


print(Y(3) + X(5))
```

Implement addition in Julia

```julia
# Julian style
struct X{T}
    num::T
end
```

```julia
struct Y{T}
    num::T
end
```

```julia
Base.:(+)(a::X, b::Y) = X(a.num + b.num)
```

```julia
Base.:(+)(a::Y, b::X) = X(a.num + b.num)
```

```julia
Base.:(+)(a::X, b::X) = X(a.num + b.num)
```

```julia
Base.:(+)(a::Y, b::Y) = Y(a.num + b.num)
```

Multiple dispatch is easier to extend!

If `C` wants to extend this method to a new type `Z`.

```
class Z:
  def __init__(self, num):
    self.num = num

  def __add__(self, other_obj):
    return Z(self.num+other_obj.num)

  def __radd__(self, other_obj):
    return Z(other_obj.num + self.num)

  def __str__(self):
    return "Z = " + str(self.num)

print(X(3) + Z(5))

print(Z(3) + X(5))
```

```julia
struct Z{T}
    num::T
end
```

```julia
Base.:(+)(a::X, b::Z) = Z(a.num + b.num)
```

```julia
Base.:(+)(a::Z, b::X) = Z(a.num + b.num)
```

```julia
Base.:(+)(a::Y, b::Z) = Z(a.num + b.num)
```

```julia
Base.:(+)(a::Z, b::Y) = Z(a.num + b.num)
```

```julia
Base.:(+)(a::Z, b::Z) = Z(a.num + b.num)
```

```
X(8)
```

```julia
X(3) + Y(5)
```

```
X(8)
```

```julia
Y(3) + X(5)
```

```
Z(8)
```

```julia
X(3) + Z(5)
```

```
Z(8)
```

```julia
Z(3) + Y(5)
```

Julia function space is exponetially large!

Quiz: If a function f has k parameters, and the module has t types, how many different functions can be generated?

```
f(x::T1, y::T2, z::T3...)
```

If it is an object-oriented language like Python？

```
class T1:
    def f(self, y, z, ...):
        self.num = num
```

Summary

- *Multiple dispatch* is a feature of some programming languages in which a function or method can be dynamically dispatched based on the run-time type.
- Julia's mutiple dispatch provides exponential abstraction power comparing with an object-oriented language.
- By carefully designed type system, we can program in an exponentially large function space.

Tuple, Array and broadcasting

Tuple has fixed memory layout, but array does not.

```
tp =  (1, 2.0, 'c')
```

```
1 tp = (1, 2.0, 'c')
```

```
Tuple{Int64, Float64, Char}
```

```
1 typeof(tp)
```

```
true
```

```
1 isbitstype(typeof(tp))
```

```
arr =  [1, 2.0, 'c']
```

```
1 arr = [1, 2.0, 'c']
```

```
Vector{Any} (alias for Array{Any, 1})
```

```julia
typeof(arr)
```

```
false
```

```julia
isbitstype(typeof(arr))
```

Boardcasting

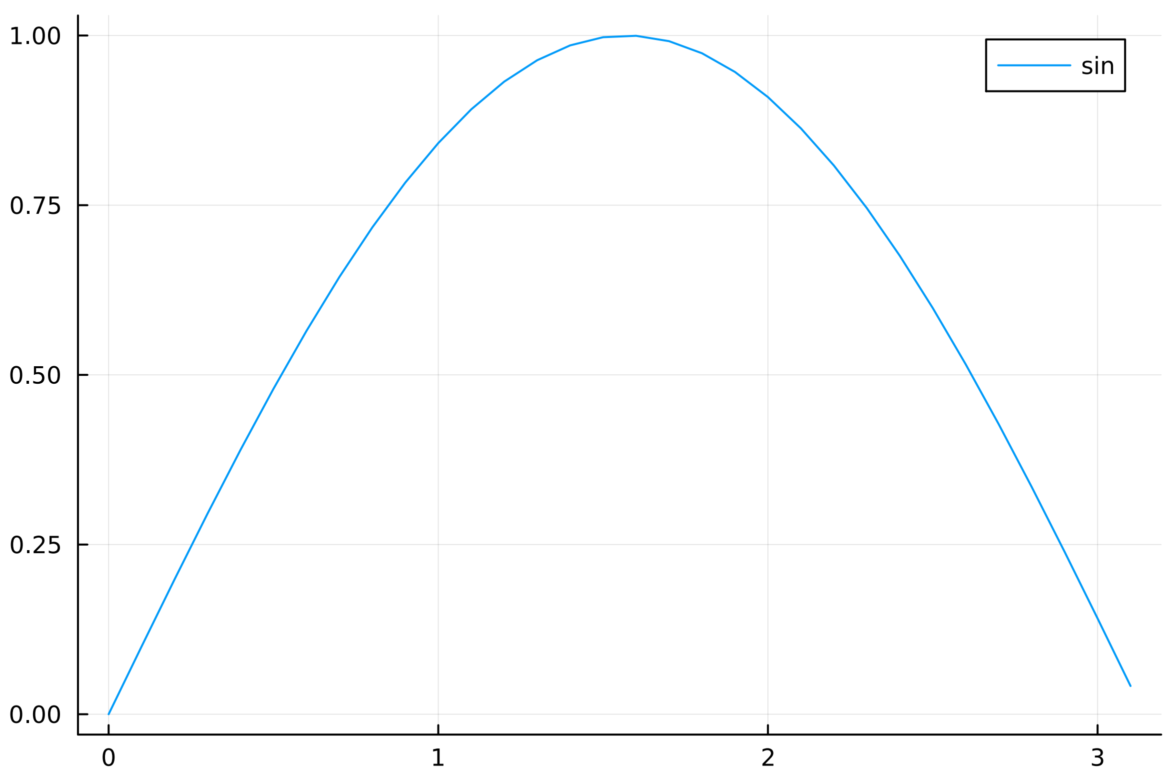
```
x = 0.0:0.1:3.1
```

```julia
x = 0:0.1:π
```

```
y =
 [0.0, 0.0998334, 0.198669, 0.29552, 0.389418, 0.479426, 0.564642, 0.644218, 0.717356, 0.78
```

```julia
y = sin.(x)
```

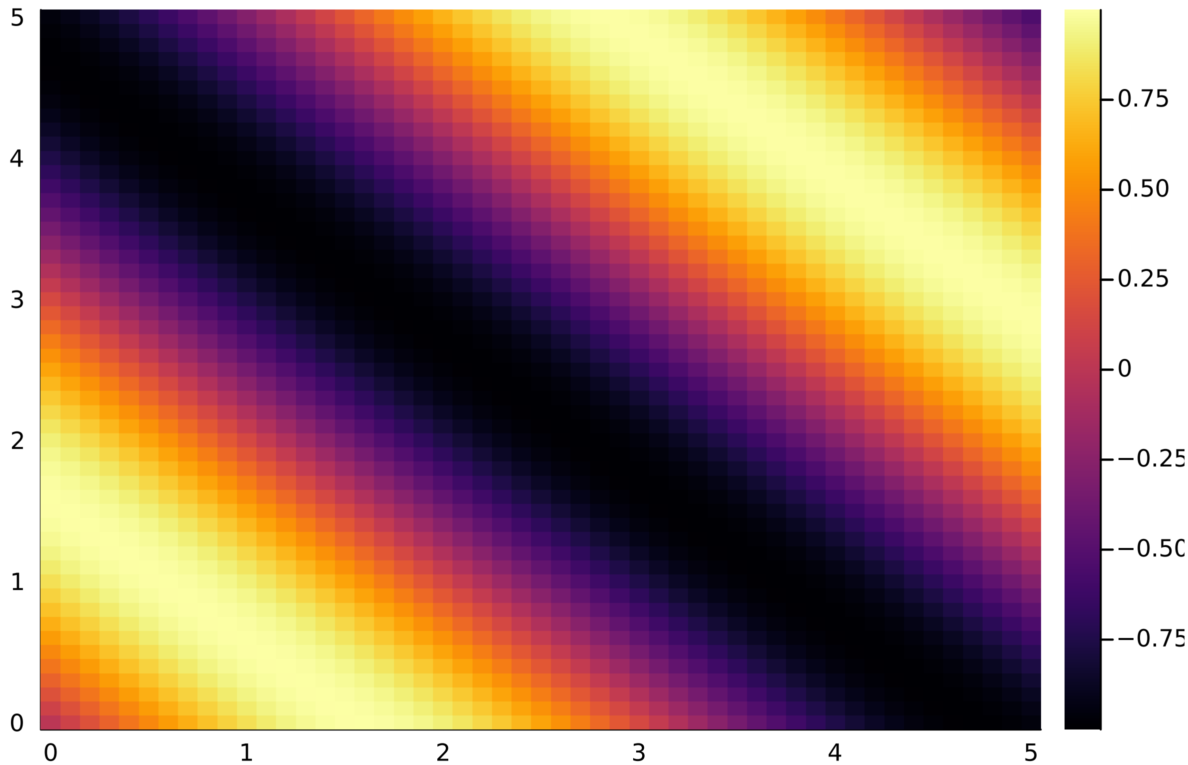
```julia
using Plots
```




```julia
plot(x, y; label="sin")
```

```
mesh =
1×100 adjoint(::UnitRange{Int64}) with eltype Int64:
 1  2  3  4  5  6  7  8  9  10  11  12  …  91  92  93  94  95  96  97  98  99  100
```

```julia
mesh = (1:100)'
```




```julia
let
    X, Y = 0:0.1:5, 0:0.1:5
    heatmap(X, Y, sin.(X .+ Y'))
end
```

Broadcasting is fast (loop fusing)!

benchmark_broadcast = ☑

```
BenchmarkTools.Trial: 10000 samples with 780 evaluations.
 Range (min … max):  161.182 ns …  14.006 μs  ┊ GC (min … max):  0.00% … 98.11%
 Time  (median):     263.499 ns               ┊ GC (median):     0.00%
 Time  (mean ± σ):   261.157 ns ± 664.432 ns  ┊ GC (mean ± σ):  12.41% ±  4.80%
```

```
 161 ns           Histogram: frequency by time          303 ns <

 Memory estimate: 336 bytes, allocs estimate: 1.
```

```julia
if benchmark_broadcast @benchmark $x .+ $y .+ $x .+ $y end
```

```
BenchmarkTools.Trial: 10000 samples with 369 evaluations.
 Range (min … max):  248.659 ns … 30.213 μs  ┊ GC (min … max):  0.00% … 98.25%
 Time  (median):     534.298 ns              ┊ GC (median):     0.00%
 Time  (mean ± σ):   536.529 ns ±  1.670 μs  ┊ GC (mean ± σ):  18.05% ±  5.71%
```

```
 249 ns           Histogram: frequency by time          617 ns <

 Memory estimate: 1008 bytes, allocs estimate: 3.
```

```julia
if benchmark_broadcast @benchmark $x + $y + $x + $y end
```

Broadcasting over non-concrete element types may be type unstable.

```
Any
```

```julia
eltype(arr)
```

```
[2, 3.0, 'd']
```

```julia
arr .+ 1
```

```
MethodInstance for (::Main.var"workspace#452".var"##dotfunction#1965#3")(::Vector{Any},
  from (::Main.var"workspace#452".var"##dotfunction#1965#3")(x1, x2) in Main.var"worksp
Arguments
  #self#::Core.Const(Main.var"workspace#452".var"##dotfunction#1965#3"())
  x1::Vector{Any}
  x2::Int64
Body::AbstractVector
1 ─ %1 = Base.broadcasted(Main.var"workspace#452".:+, x1, x2)::Base.Broadcast.Broadcast
│   %2 = Base.materialize(%1)::AbstractVector
└──      return %2
```

```julia
with_terminal() do
    @code_warntype (+).(arr, 1)
end
```

```
Any
```

```julia
eltype(tp)
```

```
MethodInstance for (::Main.var"workspace#454".var"##dotfunction#1970#3")(::Tuple{Int64,
  from (::Main.var"workspace#454".var"##dotfunction#1970#3")(x1, x2) in Main.var"worksp
Arguments
  #self#::Core.Const(Main.var"workspace#454".var"##dotfunction#1970#3"())
  x1::Tuple{Int64, Float64, Char}
  x2::Int64
Body::Tuple{Union{Char, Float64, Int64}, Union{Char, Float64, Int64}, Union{Char, Float
1 ─ %1 = Base.broadcasted(Main.var"workspace#454".:+, x1, x2)::Base.Broadcast.Broadcast
│   %2 = Base.materialize(%1)::Tuple{Union{Char, Float64, Int64}, Union{Char, Float64,
└──      return %2
```

```julia
with_terminal() do
    @code_warntype (+).(tp, 1)
end
```

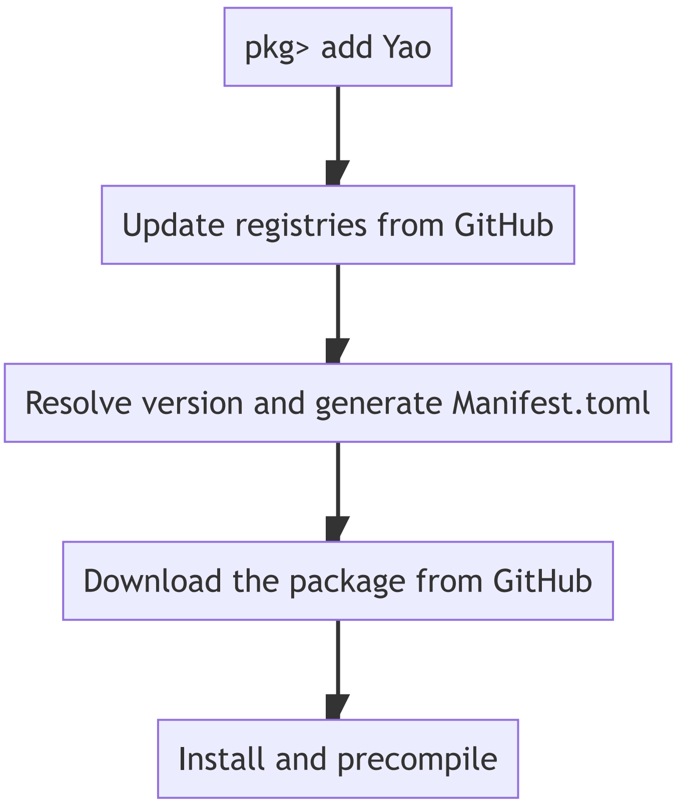
Julia package development

```julia
using TropicalNumbers
```

The file structure of a package

```
project_folder = "/home/user1/.julia/packages/TropicalNumbers/dCQLq"
```

```julia
project_folder = dirname(dirname(pathof(TropicalNumbers)))
```

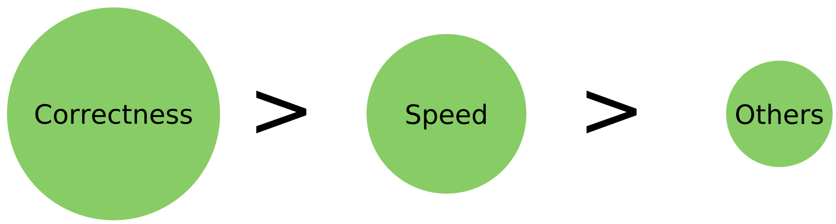



```
.
├─ .github
│  └─ .github/workflows
│     ├─ .github/workflows/TagBot.yml
│     └─ .github/workflows/ci.yml
├─ .gitignore
├─ LICENSE
├─ Project.toml
├─ README.md
├─ docs
│  └─ docs/src
│     └─ docs/src/index.md
├─ src
│  ├─ src/TropicalNumbers.jl
│  ├─ src/counting_tropical.jl
│  └─ src/tropical.jl
└─ test
   ├─ test/counting_tropical.jl
   ├─ test/runtests.jl
   └─ test/tropical.jl
```

```julia
print_dir_tree(project_folder)
```

Unit Test

```julia
using Test
```




```
Test Passed
```

```julia
@test Tropical(3.0) + Tropical(2.0) == Tropical(3.0)
```

```
Test Passed
      Thrown: BoundsError
```

```julia
@test_throws BoundsError [1,2][3]
```

```
Test Broken
  Expression: 3 == 2
```

```julia
@test_broken 3 == 2
```

```
DefaultTestSet("Tropical Number addition", [Test Broken          ], 2, false, false, true,
                                             Expression: 3 == 2
```

```julia
@testset "Tropical Number addition" begin
    @test Tropical(3.0) + Tropical(2.0) == Tropical(3.0)
    @test_throws BoundsError [1][2]
    @test_broken 3 == 2
end
```

```
Test Summary:            | Pass  Broken  Total  Time
Tropical Number addition |    2       1      3  0.0s
```

run_test = ☑

```
    Testing TropicalNumbers
     Status `/tmp/jl_IfdED3/Project.toml`
 [e30172f5] Documenter v0.27.24
 [b3a74e9c] TropicalNumbers v0.5.5
 [8dfed614] Test `@stdlib/Test`
     Status `/tmp/jl_IfdED3/Manifest.toml`
 [a4c015fc] ANSIColoredPrinters v0.0.1
 [ffbed154] DocStringExtensions v0.9.3
 [e30172f5] Documenter v0.27.24
 [b5f81e59] IOCapture v0.2.2
 [682c06a0] JSON v0.21.3
 [69de0a69] Parsers v2.5.7
 [21216c6a] Preferences v1.3.0
 [66db9d55] SnoopPrecompile v1.0.3
 [b3a74e9c] TropicalNumbers v0.5.5
 [2a0f44e3] Base64 `@stdlib/Base64`
 [ade2ca70] Dates `@stdlib/Dates`
 [b77e0a4c] InteractiveUtils `@stdlib/InteractiveUtils`
```

```julia
if run_test
    with_terminal() do
        Pkg.test("TropicalNumbers")
    end
end
```

Learn more

Case study: Create a package like HappyMolecules

With `PkgTemplates`.

https://github.com/CodingThrust/HappyMolecules.jl

Homework

Submit by making a **pull request** to the course github repository (the `courseworks/week2` folder).

1. Fill the following form

	is concrete	is primitive	is abstract	is bits type	is mutable
`ComplexF64`					
`Complex{AbstractFloat}`					
`Complex{<:AbstractFloat}`					
`AbstractFloat`					
`Union{Float64, ComplexF64}`					
`Int32`					
`Matrix{Float32}`					
`Base.RefValue`					

Task: Fill the form in a markdown file and include it in your pull request.

Hint: how to create a table in markdown.

2. Coding

Choose one: (a), (b) or (c).

(a - Easy). Task: Bellow you will find a live coding. Open an Julia REPL and type what the live coding types. Submit the `~/.julia/logs/repl_history.jl` file (only the related portion) as a proof of work.

(b - Hard). Two dimensional brownian motion Brownian motion in two dimension is composed of cumulated sumummation of a sequence of normally distributed random displacements, that is Brownian motion can be simulated by successive adding terms of random normal distribute numbernamely:

$$
\begin{aligned}
&\mathbf{x}(t=0) \sim N(\mathbf{0}, \mathbf{I})\\
&\mathbf{x}(t=1) \sim \mathbf{x}(t=0) + N(\mathbf{0}, \mathbf{I})\\
&\mathbf{x}(t=2) \sim \mathbf{x}(t=1) + N(\mathbf{0}, \mathbf{I})\\
&\ldots
\end{aligned}
$$

where $N(\mu, \mathbf{\Sigma})$ is a multivariate normal distribution.

Task: Simulate 2D brownian motion for 10000 steps, and include the code in your pull request. Please include the following content in the pull request description:

- the benchmark result

- a plot of the particle trajectory,

Hint: you can make a plot with `Plots` or `Makie`.

(c - Harder). The `3x+1` problem

> Suppose we start with a positive integer, and if it is odd then multiply it by 3 and add 1, and if it is even, divide it by 2. Then repeat this process as long as you can. Do you eventually reach the integer 1, no matter what you started with?

For instance, starting with 5, it is odd, so we apply 3x+1. We get 16, which is even, so we divide by 2. We get 8, and then 4, and then 2, and then 1. So yes, in this case, we eventually end up at 1.

Task: Verifier this hypothesis for all positive integers of Int32 type, and include your code with test in your pull request.

Hint: how to write tests.

Live coding

This script is for Julia code training