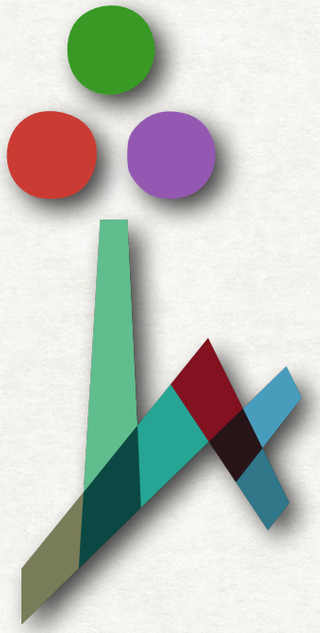
Julia for Doing Science



Nir Shaviv

Hebrew University of Jerusalem

Why Julia?

Julia was developed in MIT (launched 2012)

From the Julia website:

Fast

Julia was designed from the beginning for high performance. Julia programs compile to efficient native code for multiple platforms via LLVM.

Dynamic

Julia is dynamically typed, feels like a scripting language, and has good support for interactive use.

Reproducible

Reproducible environments make it possible to recreate the same Julia environment every time, across platforms, with pre-built binaries.

Composable

Julia uses multiple dispatch as a paradigm, making it easy to express many object-oriented and functional programming patterns. The talk on the Unreasonable Effectiveness of Multiple Dispatch explains why it works so well.

General

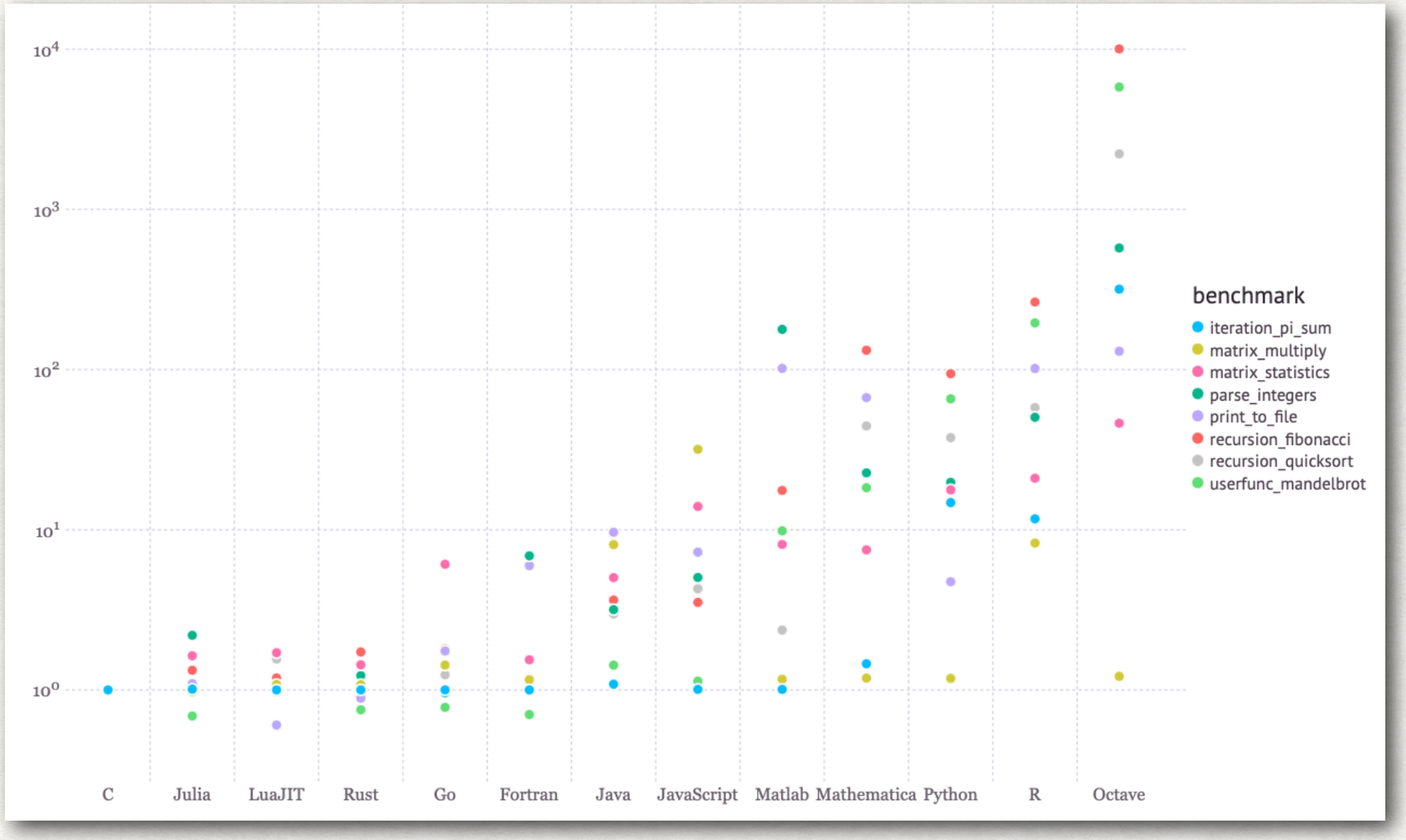
Julia provides asynchronous I/O, metaprogramming, debugging, logging, profiling, a package manager, and more. One can build entire Applications and Microservices in Julia.

Open source

Julia is an open source project with over 1,000 contributors. It is made available under the MIT license. The source code is available on GitHub.

Why Julia?

It's High Level and interactive (like Matlab, Python, etc.) but fast (like Fortran and c), thus solving the 2 language problem.



It's fast because every line/function is compiled to LLVM and then to machine language.

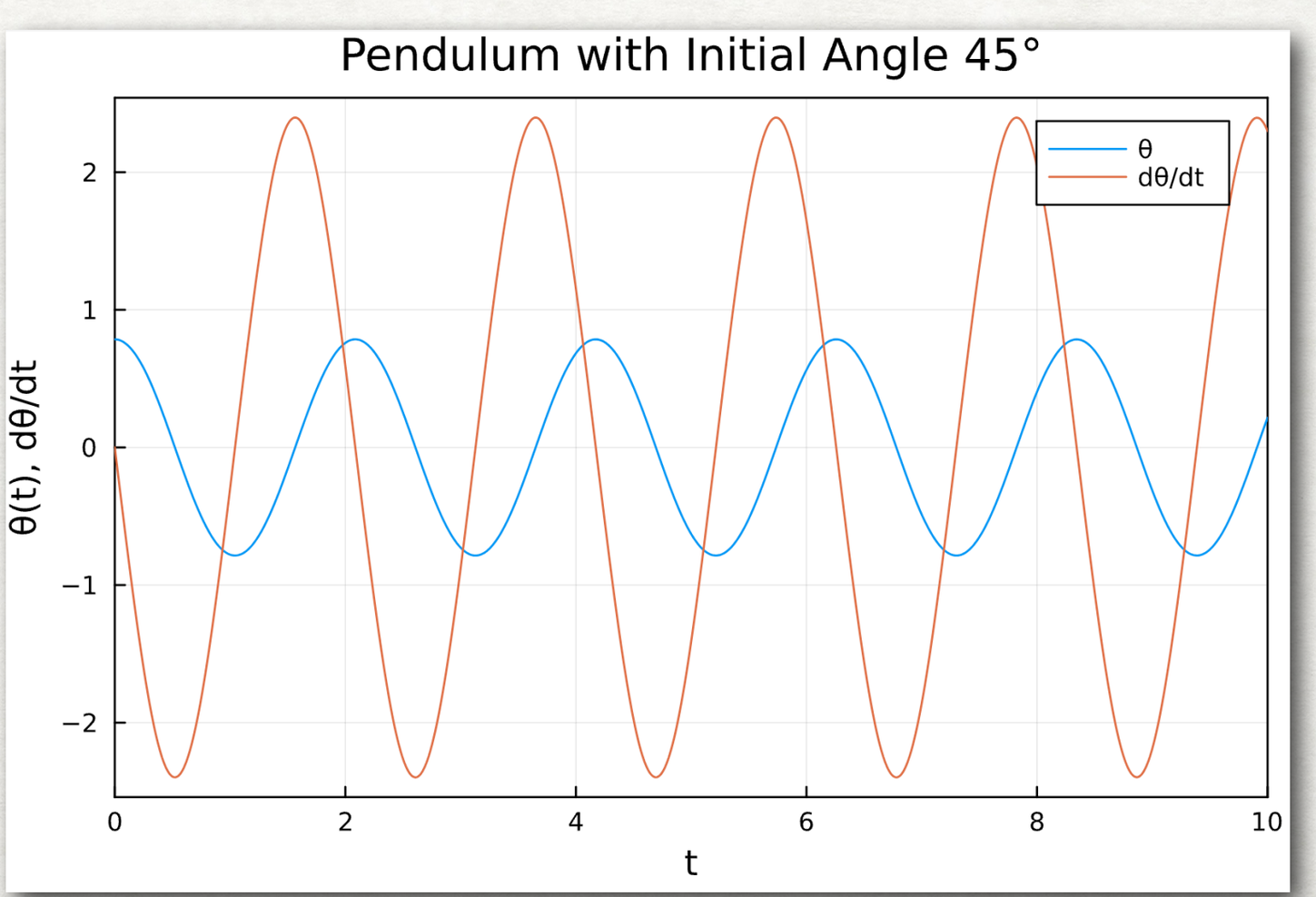
Why Julia?

- It is easy to install
- Easy package management
- The same code runs on all platforms(*)
- Unicode friendly (makes neat codes)
- Math like (and not too verbose)
- Multiple Dispatch
- Easily integrate with Fortran, C, Python and other languages

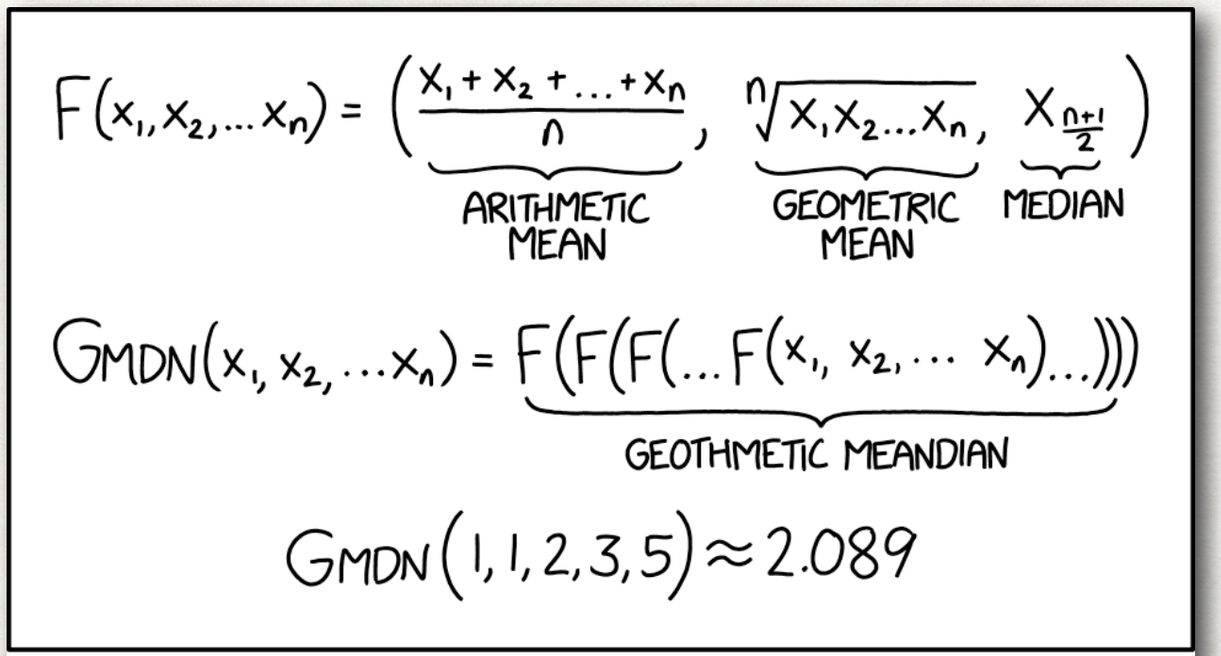
(*) except perhaps NVIDIA vs. non-NVIDIA GPUs if you use the CUDA package, or other hardware specific packages, such as MKL

Example code in Julia

```julia
using DifferentialEquations, Plots

# Constants for the pendulum (length and gravitational acceleration)
L = 1.0  # length of the pendulum
g = 9.81  # acceleration due to gravity

# Pendulum's equation of motion
function pendulum!(du, u, p, t)
    θ, ω = u  # u[1] = θ (angle), u[2] = ω (angular velocity)
    du[1] = ω
    du[2] = -(g/L) * sin(θ)
end

# Time span for the simulation
tspan = (0.0, 10.0)

# Initial conditions
θ0 = π / 4  # 45 degrees
ω0 = 0      # initial angular velocity

u0 = [θ0, ω0]
prob = ODEProblem(pendulum!, u0, tspan)
sol = solve(prob)

plot(sol, vars=(1), label="θ")
plot!(sol, vars=(2), label="dθ/dt")
plot!(title = "Pendulum with Initial Angle 45°”)
plot!(xlabel="t", ylabel="θ(t), dθ/dt”)
plot!(frame=:box)
```



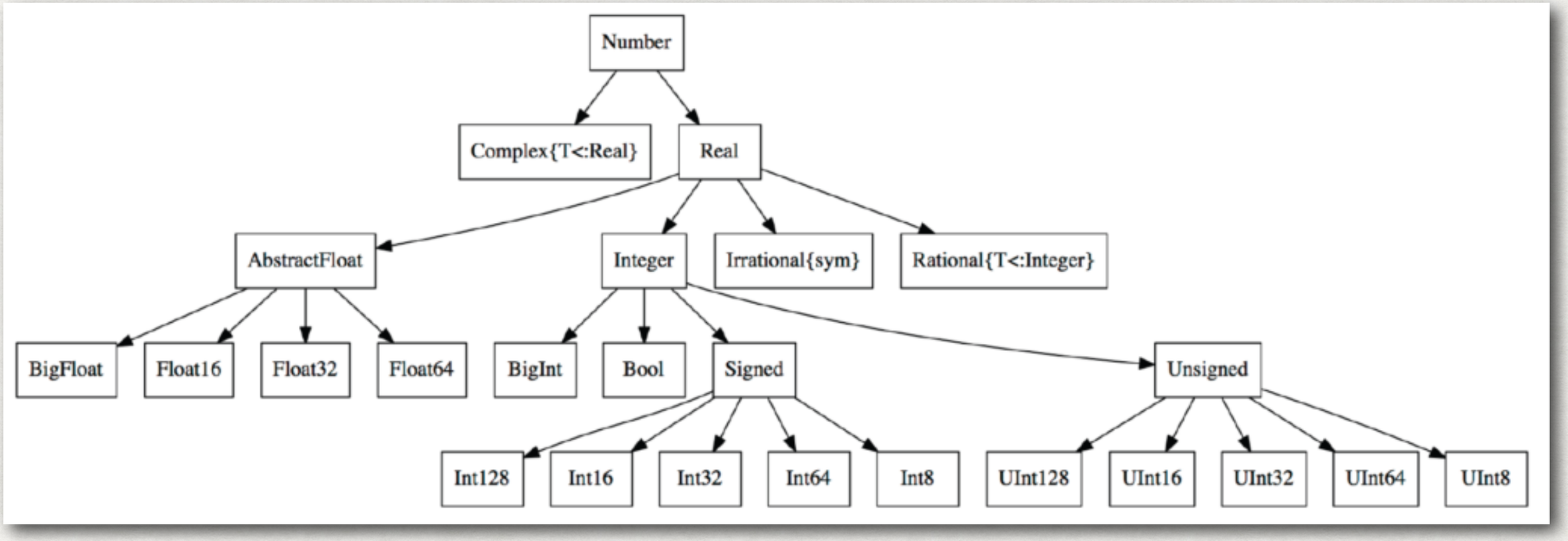
Example code in Julia




```julia
using Statistics
F(x) = [mean(x), prod(x)^(1/length(x)), median(x)]
Fⁿ(x,n) = ∘(fill(F,n)...)(x)
Fⁿ([1,1,2,3,5],100)

3-element Vector{Float64}:
 2.089057949736859
 2.089057949736859
 2.089057949736859
```

Data Types

- Many out of the Box, you can define your own.



- Strings
- Vectors of above, n-dimensional arrays of above
- Vectors of vectors, arrays of vectors, ...
- Tuples, dictionaries, symbols, iterators, ranges, ...

Multiple Dispatch

- A function can be defined to be general, or specific for specific types (Float64, Complex Numbers, numbers ± errors, Strings, Arrays, Structs, RGB Images)
- e.g., a+b, + has 207 methods defined out of the box.
- e.g., The package error measurements defines a+b with a and b having errors. It also defines Plot(vector of errors) to plot with error bars.

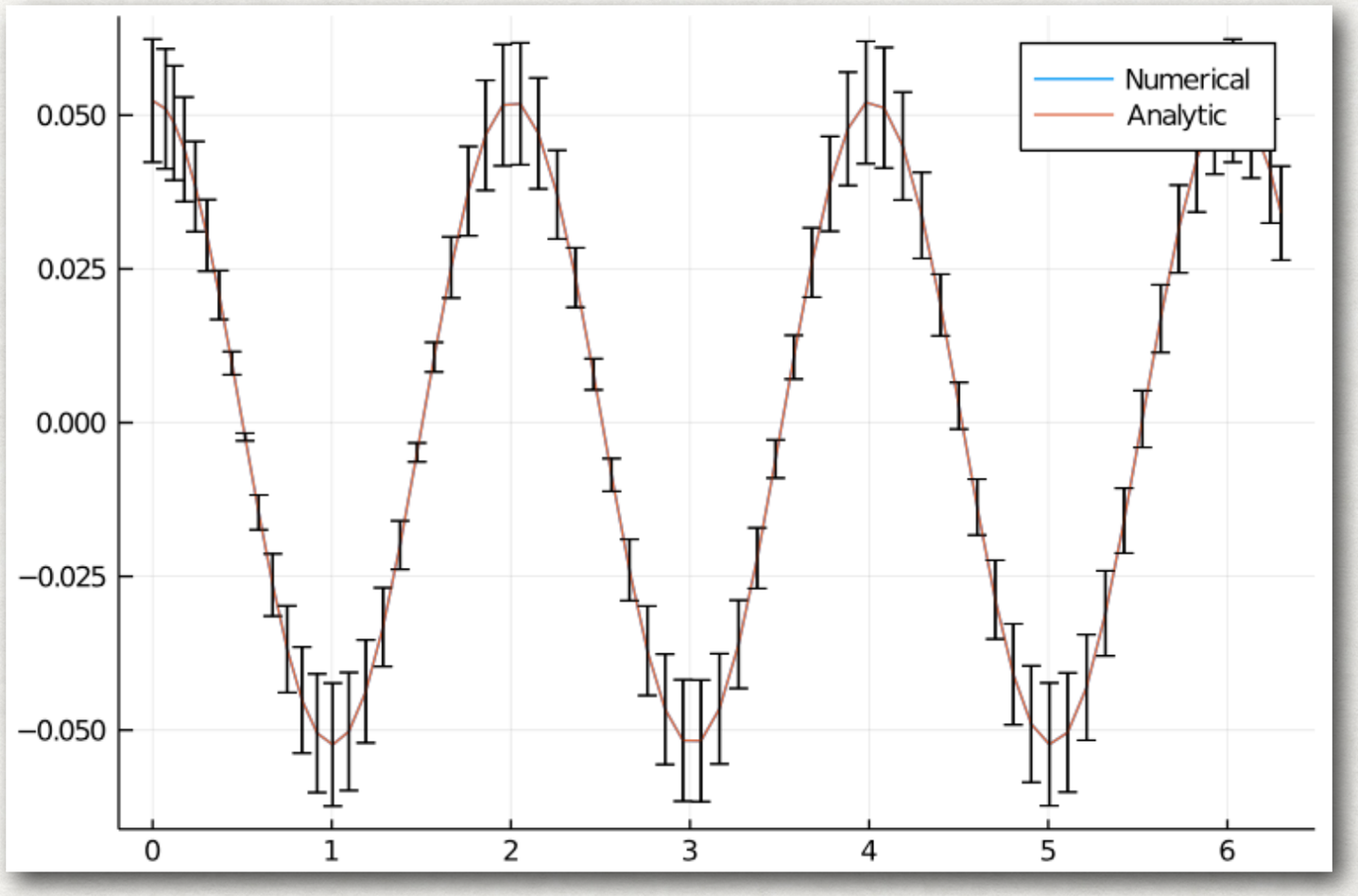
The Unreasonable Effectiveness of Multiple Dispatch:
https://www.youtube.com/watch?v=kc9HwsxE1OY

OOP vs. Functional Programing

- It isn't an Object Oriented Programming language per se. It is function oriented.
- Suppose you have a game with a "wall" and a "ball". You will have one class for walls, and one for balls. In OOP you will have a function
- wall.collision(ball) or perhaps more likely
- ball.collision(wall) and another
- ball.collision(ball).
- In Julia, you will have a function
- collision(a_wall::Wall, a_ball::Ball) and another defined for
- collision(a_ball::Ball, a_ball::Ball).

Example code in Julia

```julia
using DifferentialEquations, Plots, Measurements

g = 9.79 ± 0.02; # Gravitational constants
L = 1.00 ± 0.01; # Length of the pendulum

#Initial Conditions
u₀ = [0 ± 0, π / 60 ± 0.01]
#Define the problem
function pendulum!(du,u,p,t)
    θ  = u[1]
    dθ = u[2]
    du[1] = dθ
    du[2] = -(g/L)*θ
end

#Pass to solvers
prob = ODEProblem(pendulum!, u₀, tspan)
sol = solve(prob, Tsit5(), reltol = 1e-6)

# Analytic solution
u = u₀[2] .* cos.(sqrt(g / L) .* sol.t)

plot(sol.t, getindex.(sol.u, 2), label = "Numerical")
plot!(sol.t, u, label = "Analytic")
```



The Unreasonable Effectiveness of Multiple Dispatch:
https://www.youtube.com/watch?v=kc9HwsxE1OY

Running Environments

- As a command

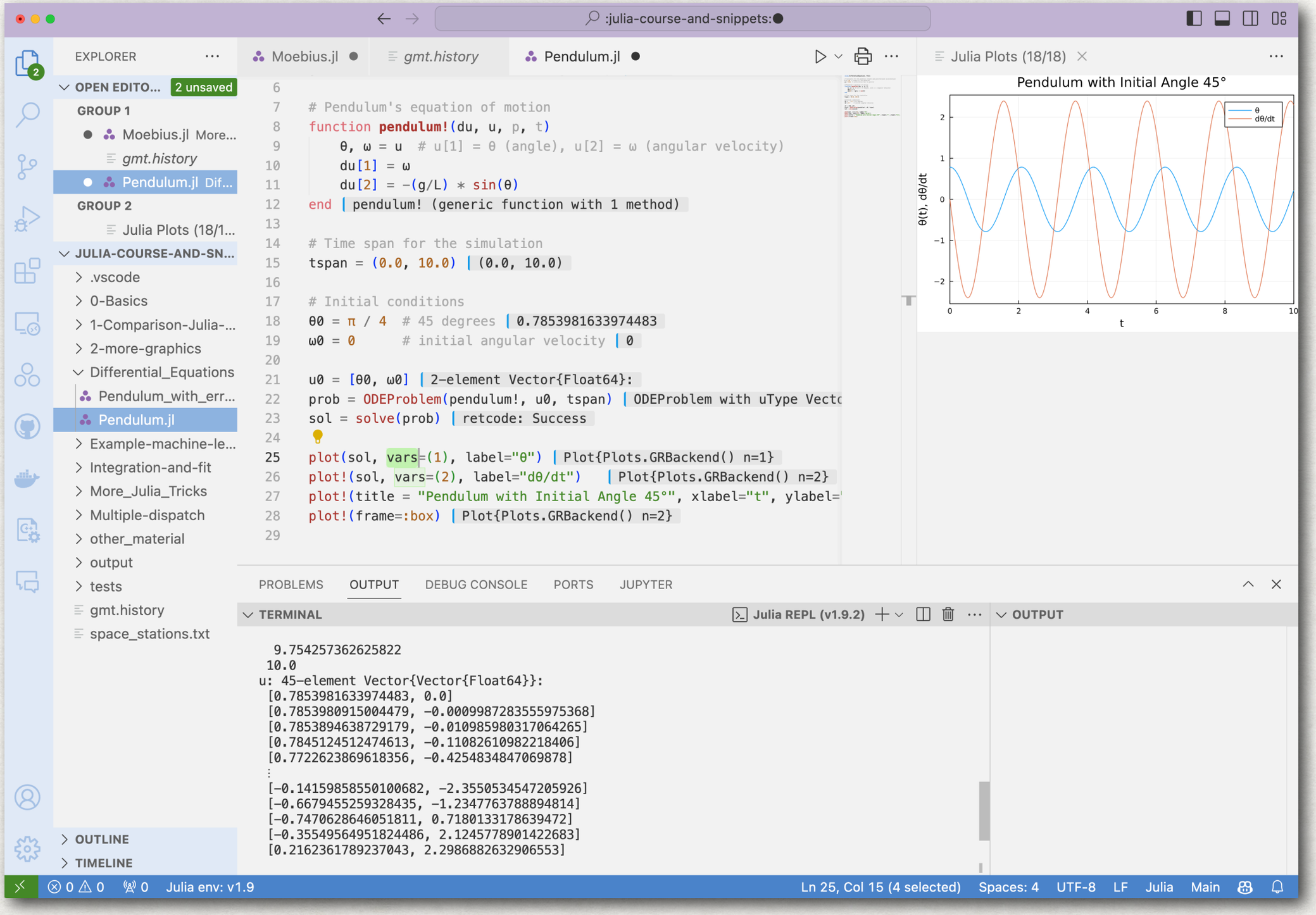
```
shaviv@triton ~  % cat hello_world.jl
println("Hello World!")
shaviv@triton ~  % julia hello_world.jl
Hello World!
shaviv@triton ~  %
```

- As an interactive "interpreter":

```
shaviv@triton ~  % julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.9.2 (2023-07-05)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> println("Hello world")
Hello world

julia>
```

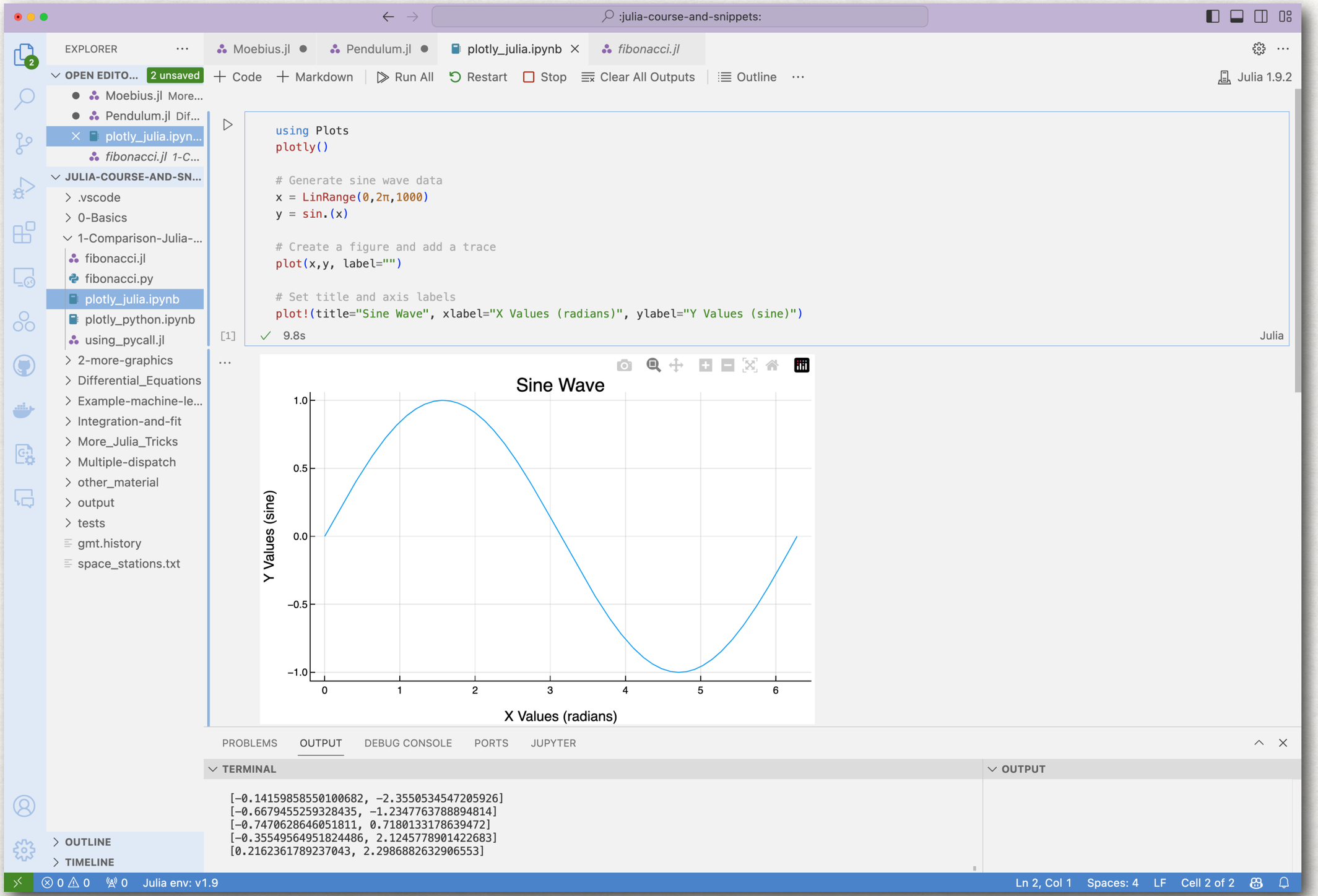
Running Environments

- VSCode



Running Environments

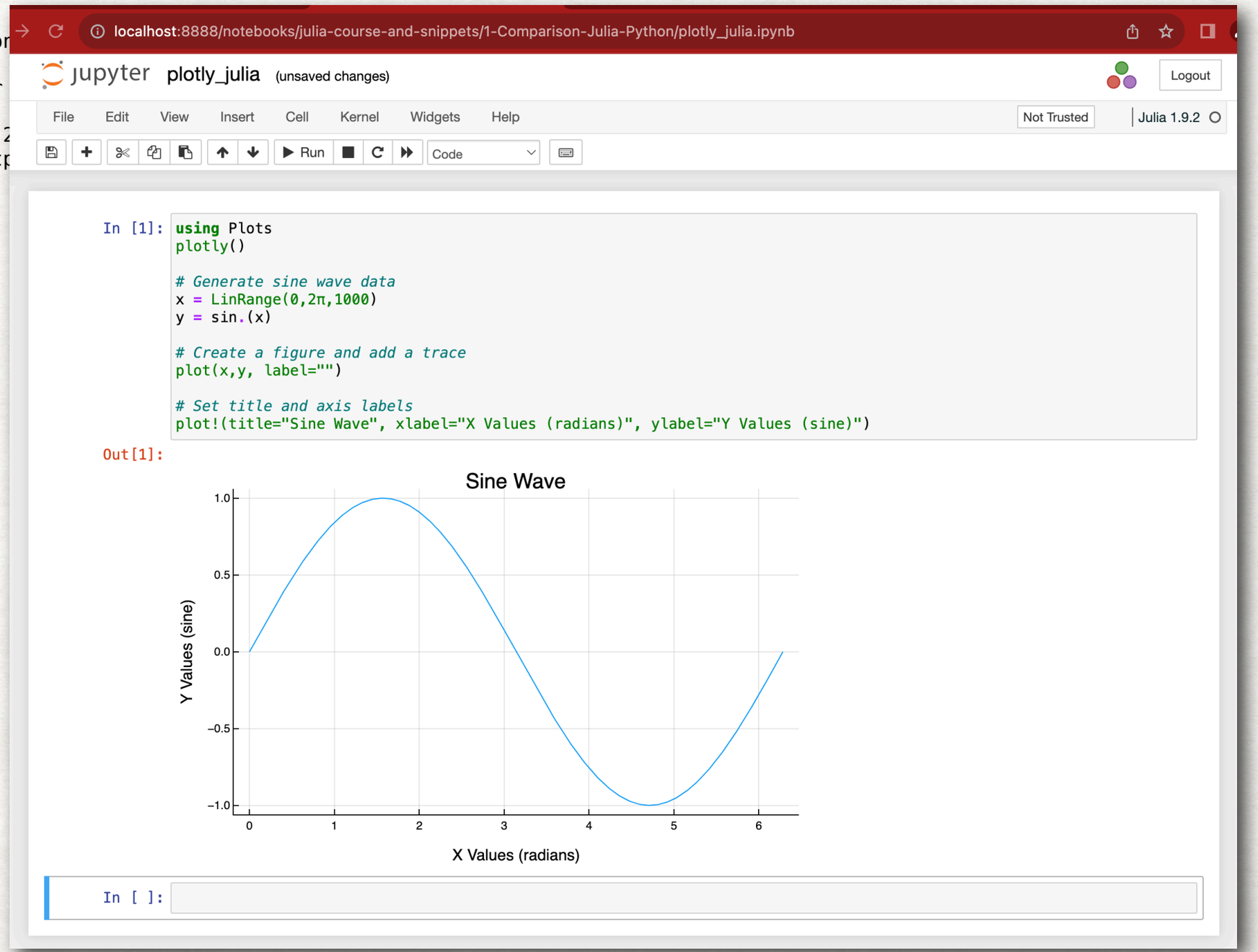
- Jupyter notebook (in VSCode)



Running Environments

- Jupyter notebook (in browser)

```
shaviv@triton ~  % julia
               _
   _       _ _(_)_     |  Documentation
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.9.2
 _/ |\__'_|_|_|\__'_|  |  Official http
|__/                   |

julia> using IJulia

julia> notebook()
```



What now?

- We'll install Julia & VSCode
- We'll see a few examples
- Go over the language
- Use it for applications we'll encounter in astrophysics (plotting, integrating, ODEs, machine learning, astronomical data manipulation, etc)

Install Julia

- Go to https://julialang.org/downloads/
- Download julia for your platform

julia Download Documentation Learn Blog Community Contribute JSoC Sponsor

Download Julia

Star 43,540

Please star us on GitHub. If you use Julia in your research, please cite us. If possible, do consider sponsoring us.

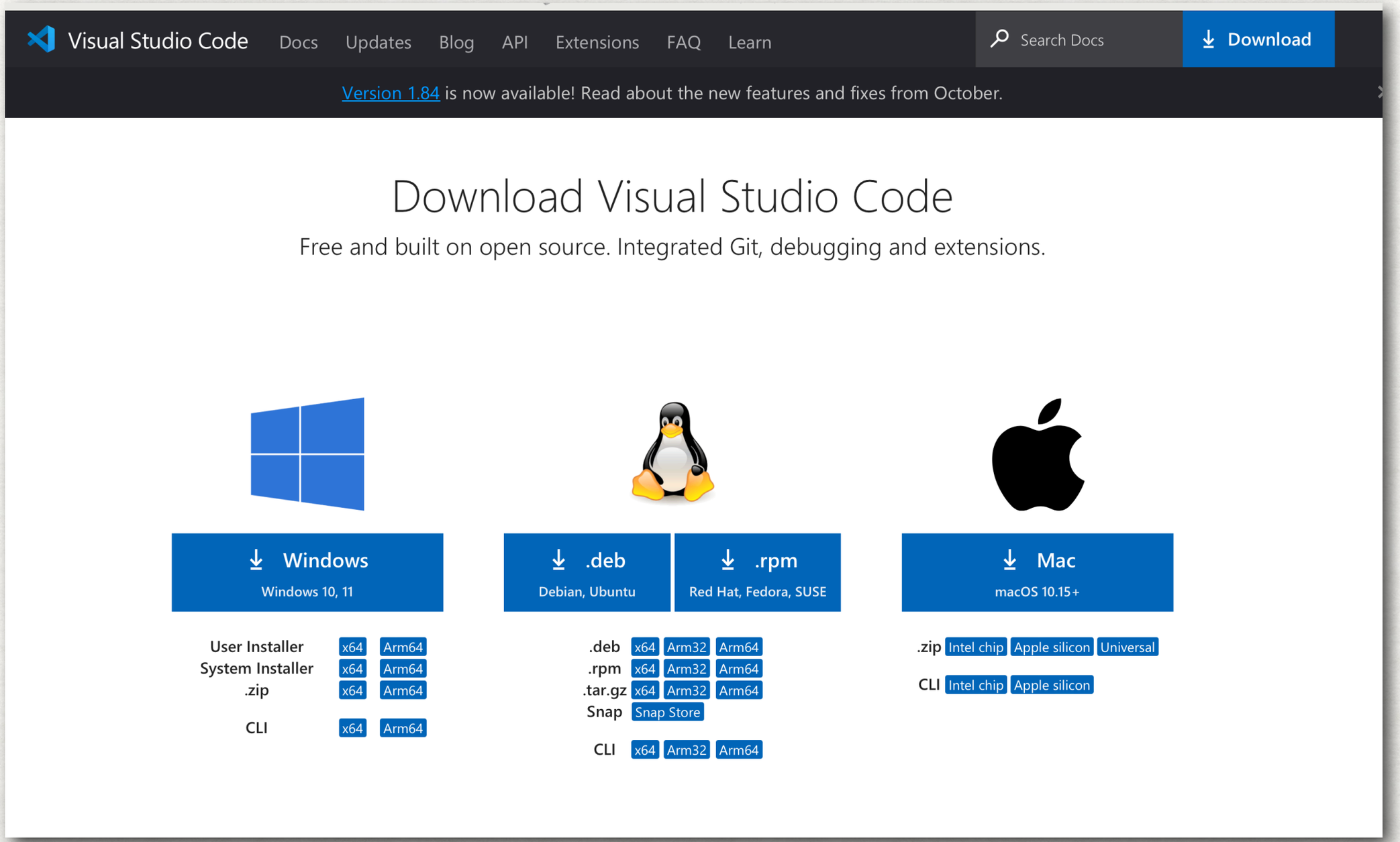
Current stable release: v1.9.4 (November 14, 2023)

Checksums for this release are available in both MD5 and SHA256 formats.

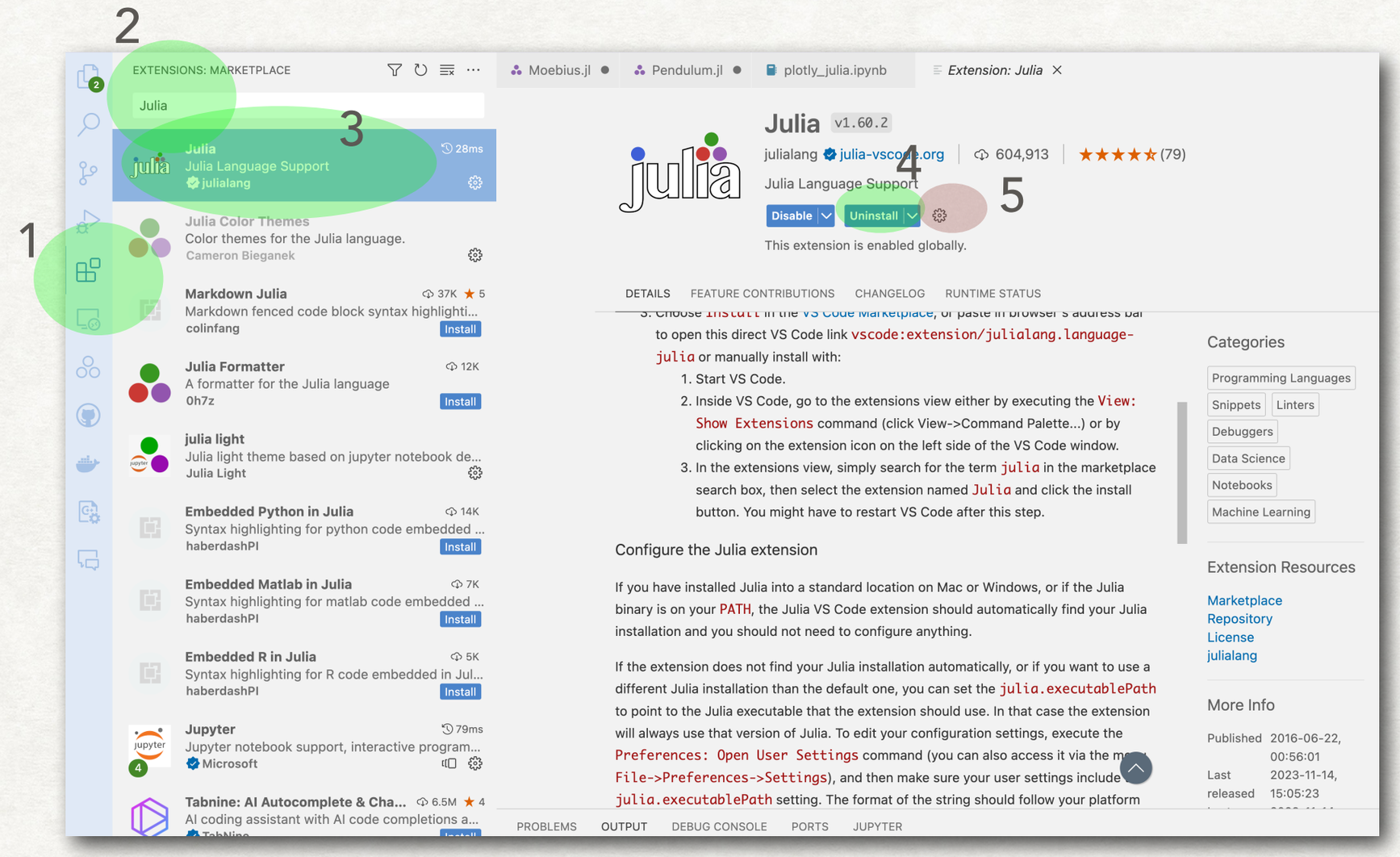
Windows [help]	64-bit (installer), 64-bit (portable)	32-bit (installer), 32-bit (portable)	
macOS x86 (Intel or Rosetta) [help]	64-bit (.dmg), 64-bit (.tar.gz)		
macOS (Apple Silicon) [help]	64-bit (.dmg), 64-bit (.tar.gz)		
Generic Linux on x86 [help]	64-bit (glibc) (GPG), 64-bit (musl)[1] (GPG)	32-bit (GPG)	
Generic Linux on ARM [help]	64-bit (AArch64) (GPG)		
Generic Linux on PowerPC [help]	64-bit (little endian) (GPG)		
Generic FreeBSD on x86 [help]	64-bit (GPG)		
Source	Tarball (GPG)	Tarball with dependencies (GPG)	GitHub

Installing VSCode

- Go to https://code.visualstudio.com/download
- Download and install!



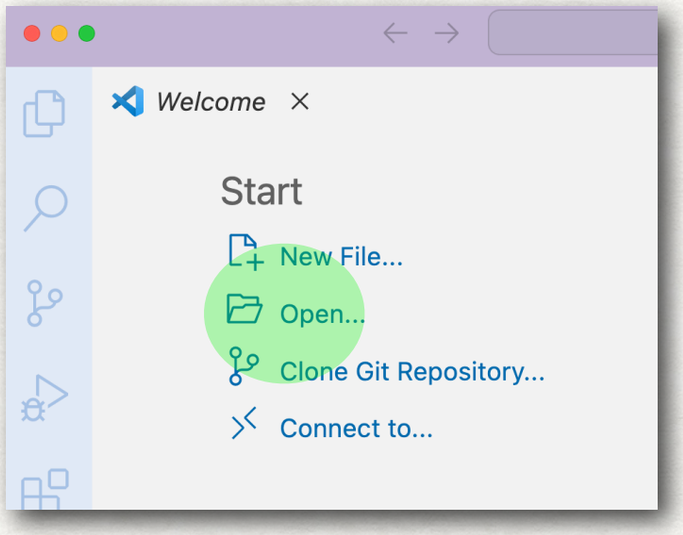
Installing Julia extension on VSCode



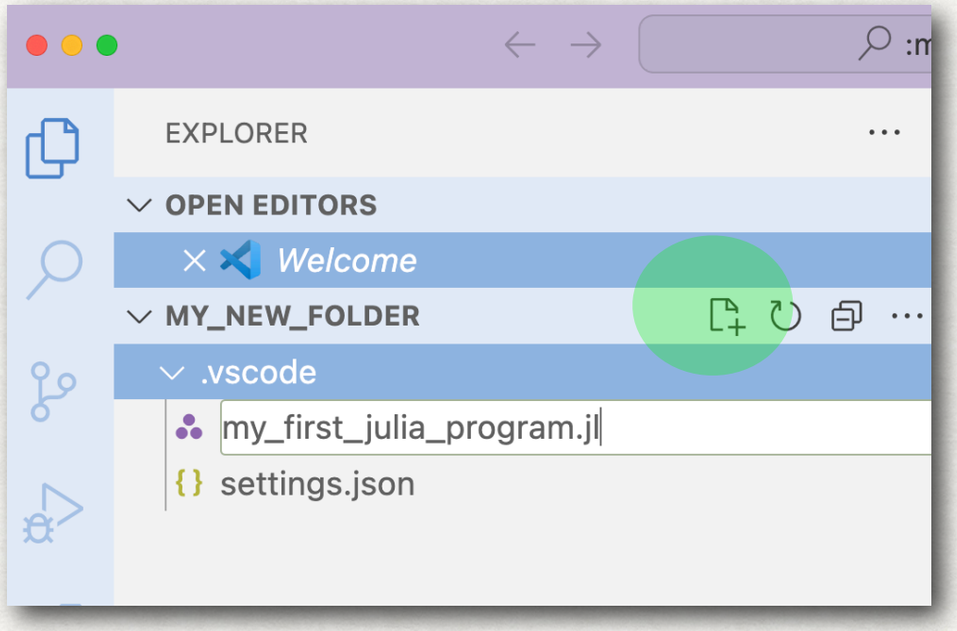
1. Go to extensions
2. Search “Julia”
3. Click Julia / Julia Language Support
4. Click install
5. Supposed to work. If not, you will need to change julia.executablePath

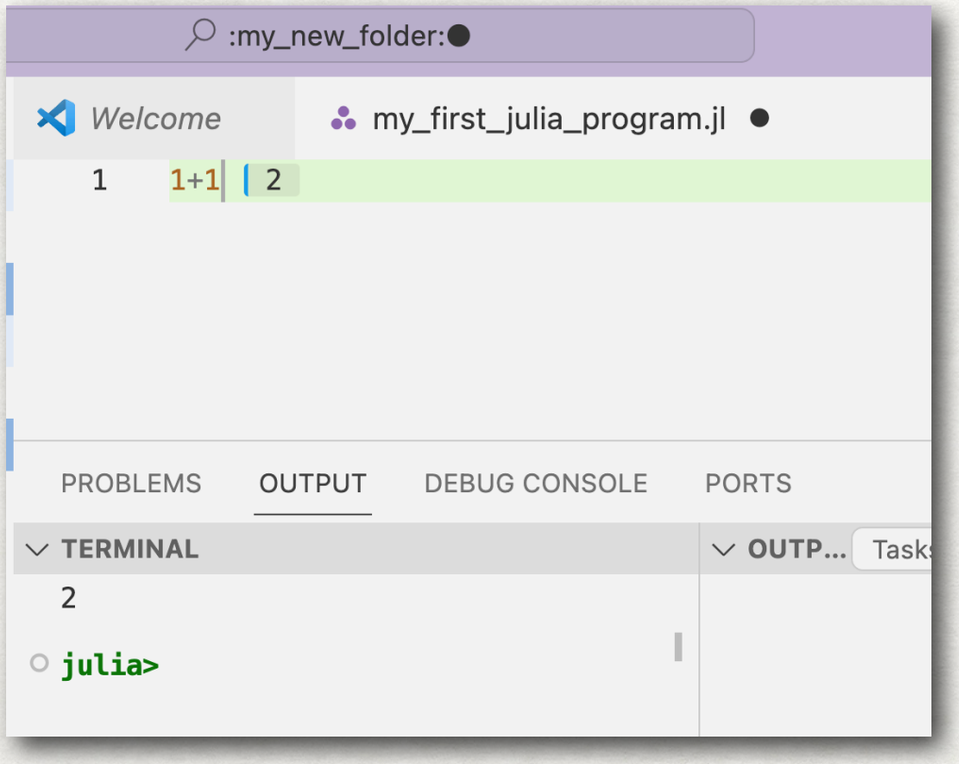
Open first julia file

- In VSCode: Open a new folder



- Click on new file

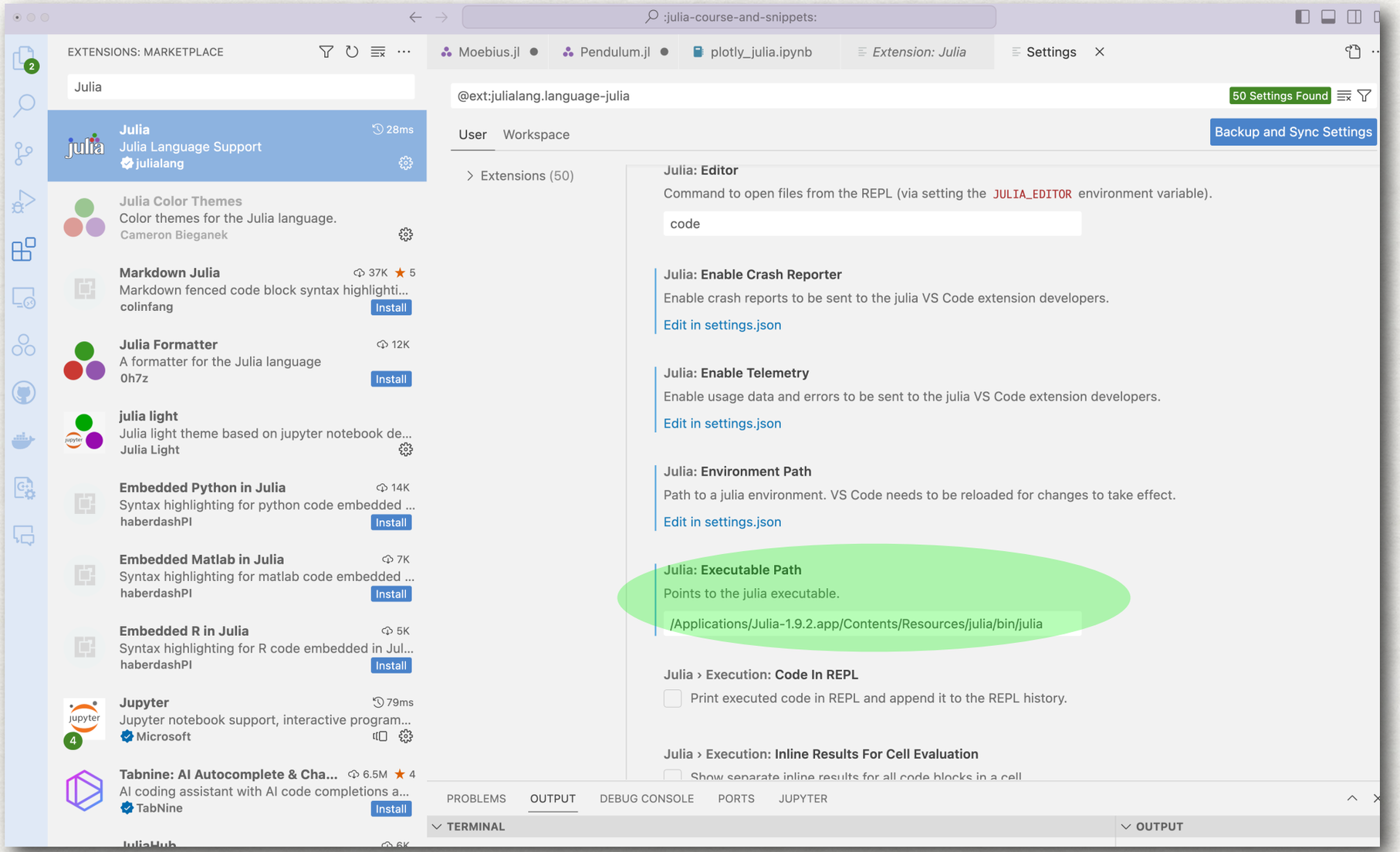




- Click on file to open it
- Type 1+1 <shift>+<enter>
- You will see the result in the file and in the REPL

Installing Julia extension on VSCode

In case Julia doesn't run...



change julia.executablePath