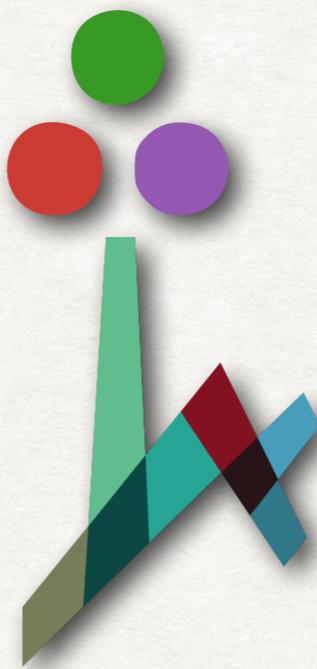


Julia for Doing Science



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

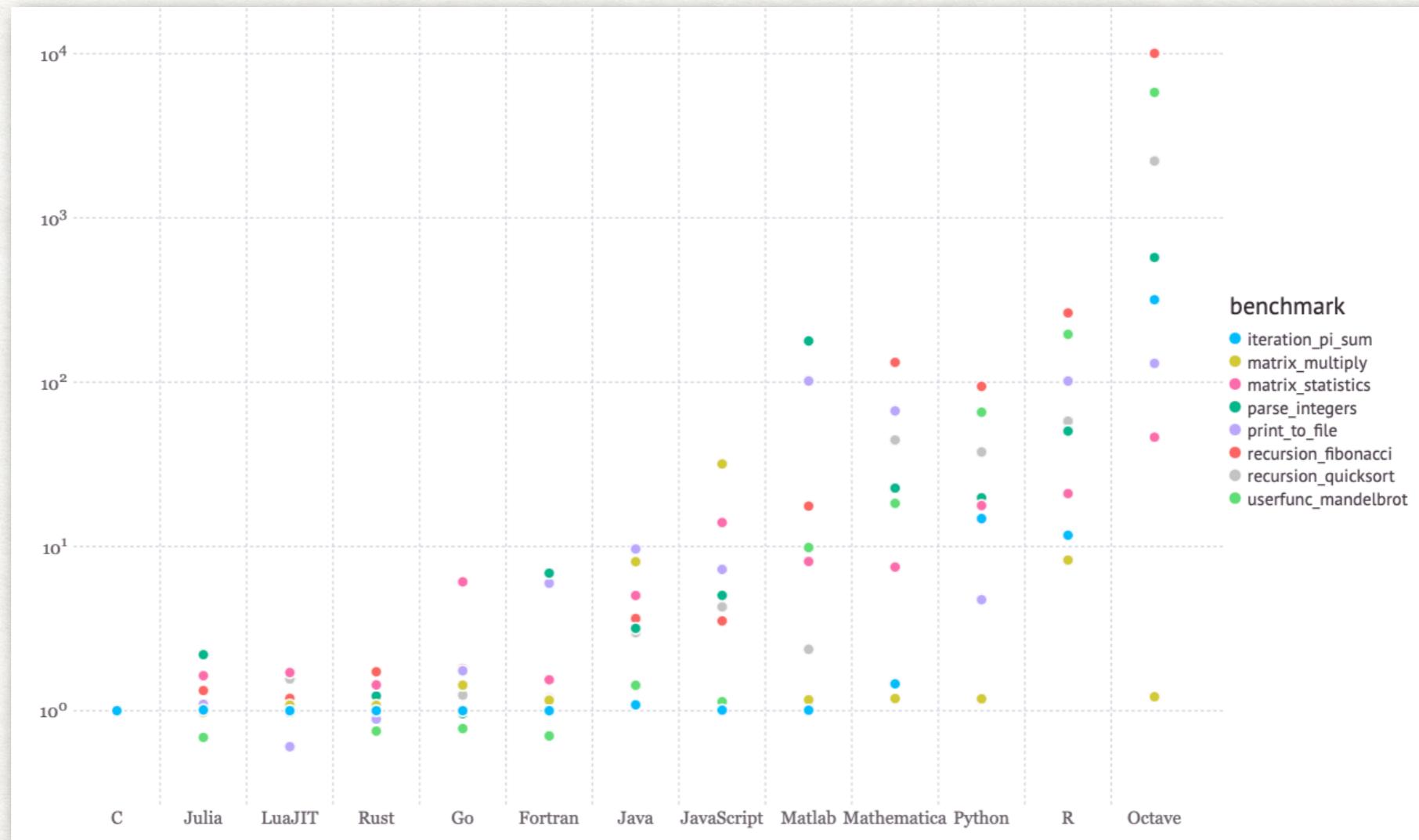
Julia provides [asynchronous I/O](#), [metaprogramming](#), [debugging](#), [logging](#), [profiling](#), a [package manager](#), and more. One can build entire [Applications](#) and [Microservices](#) in Julia.

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

```
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

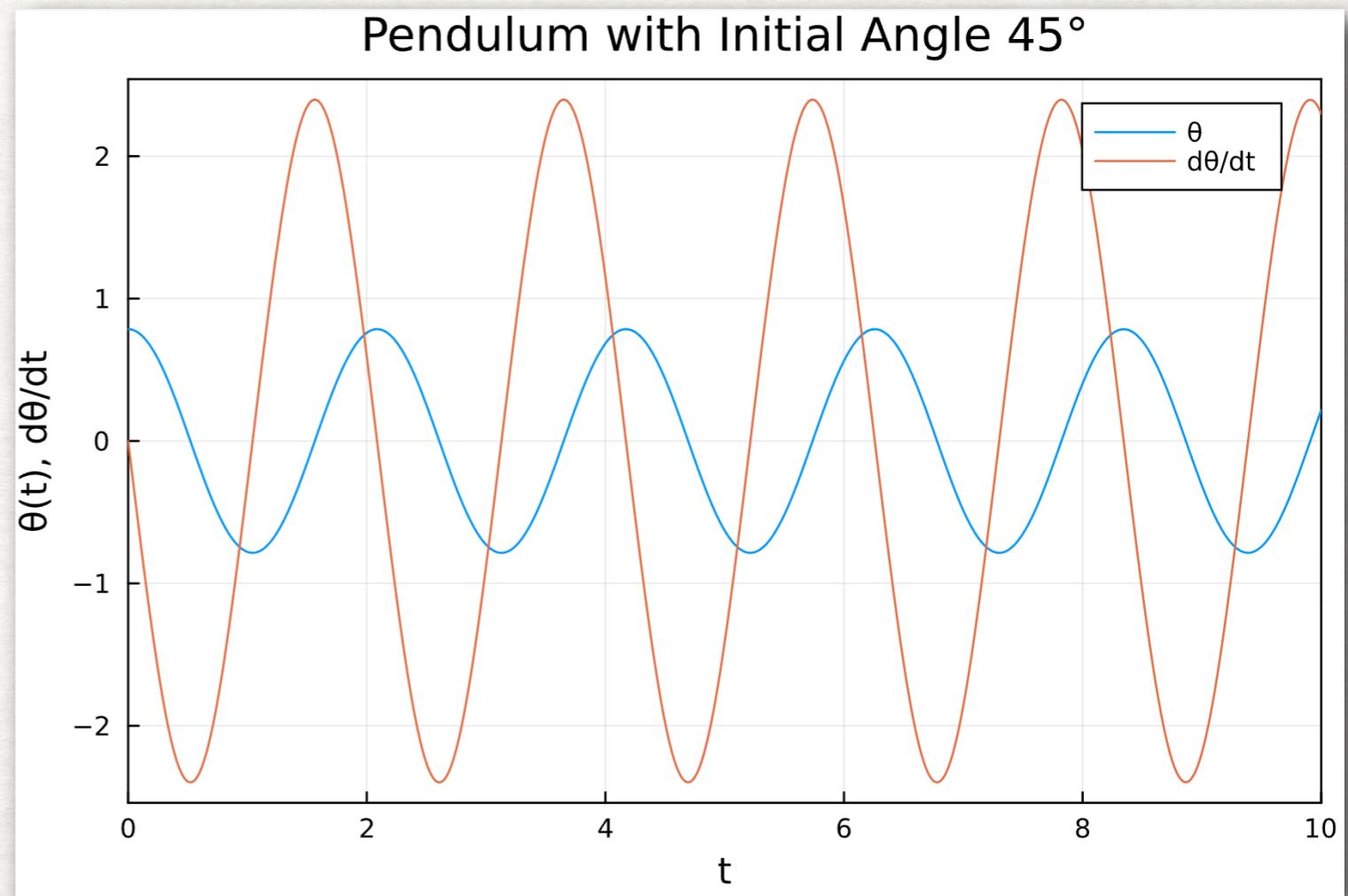
# Pendulum's equation of motion
function pendulum!(du, u, p, t)
    θ, ω = u # u[1] = θ (angle), u[2] = ω (angular velocity)
    du[1] = ω
    du[2] = -(g/L) * sin(θ)
end

# 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

$$F(x_1, x_2, \dots, x_n) = \left(\underbrace{\frac{x_1 + x_2 + \dots + x_n}{n}}_{\text{ARITHMETIC MEAN}}, \underbrace{\sqrt[n]{x_1 x_2 \dots x_n}}_{\text{GEOMETRIC MEAN}}, \underbrace{x_{\frac{n+1}{2}}}_{\text{MEDIAN}} \right)$$

$$GMDN(x_1, x_2, \dots, x_n) = \underbrace{F(F(F(\dots F(x_1, x_2, \dots, x_n) \dots)))}_{\text{GEOTHMETIC MEANDIAN}}$$

$$GMDN(1, 1, 2, 3, 5) \approx 2.089$$

STATS TIP: IF YOU AREN'T SURE WHETHER TO USE THE MEAN, MEDIAN, OR GEOMETRIC MEAN, JUST CALCULATE ALL THREE, THEN REPEAT UNTIL IT CONVERGES

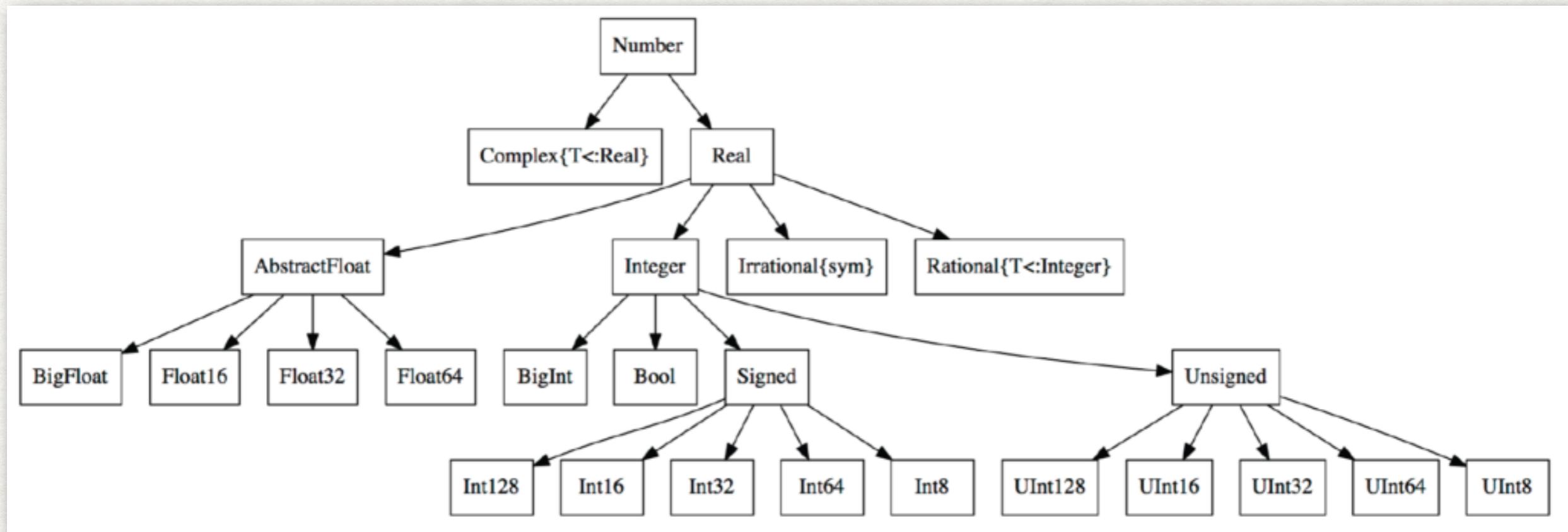
```
using Statistics  
F(x) = [mean(x), prod(x)^^(1/length(x)), median(x)]  
F^n(x, n) = ∘(fill(F, n)...)(x)  
F^n([1, 1, 2, 3, 5], 100)
```

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 \pm 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 Programming

- 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

```
using DifferentialEquations, Plots, Measurements

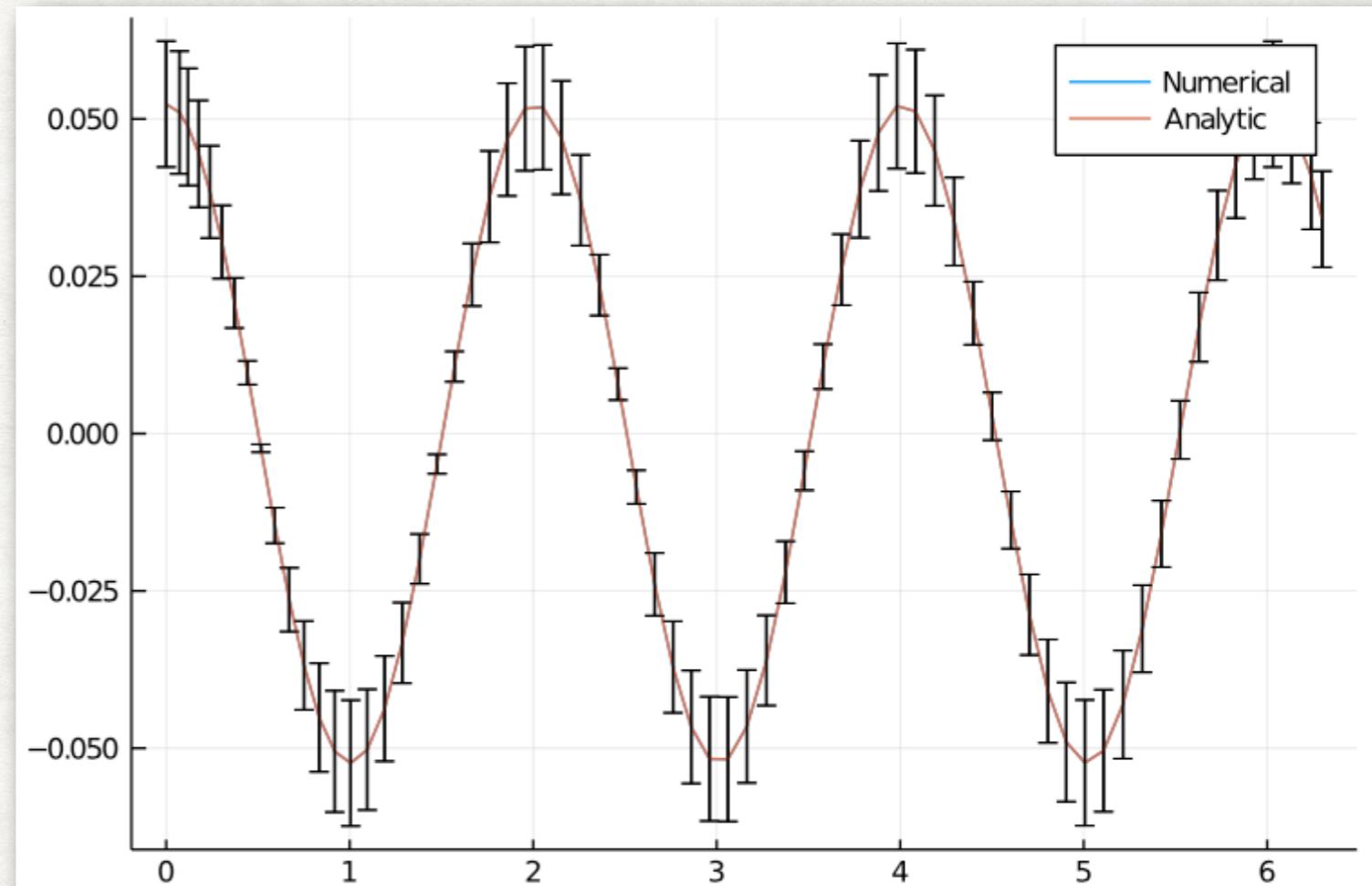
g = 9.79 ± 0.02; # Gravitational constants
L = 1.00 ± 0.01; # Length of the pendulum

#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

The screenshot shows the Visual Studio Code (VSCode) interface with the following components:

- EXPLORER**: Shows a file tree with several groups and a Julia course directory.
- CODEVIEW**: Displays a Julia script named `Pendulum.jl` containing code for solving a pendulum's equation of motion using ODEProblem and Plot objects.
- OUTPUT**: Shows the results of a Julia REPL session, including the initial conditions and the resulting numerical solution vectors.
- TERMINAL**: Shows the command `Julia REPL (v1.9.2)`.
- Plots**: A plot titled "Pendulum with Initial Angle 45°" showing the angle $\theta(t)$ (blue line) and angular velocity $d\theta/dt$ (orange line) over time t from 0 to 10.

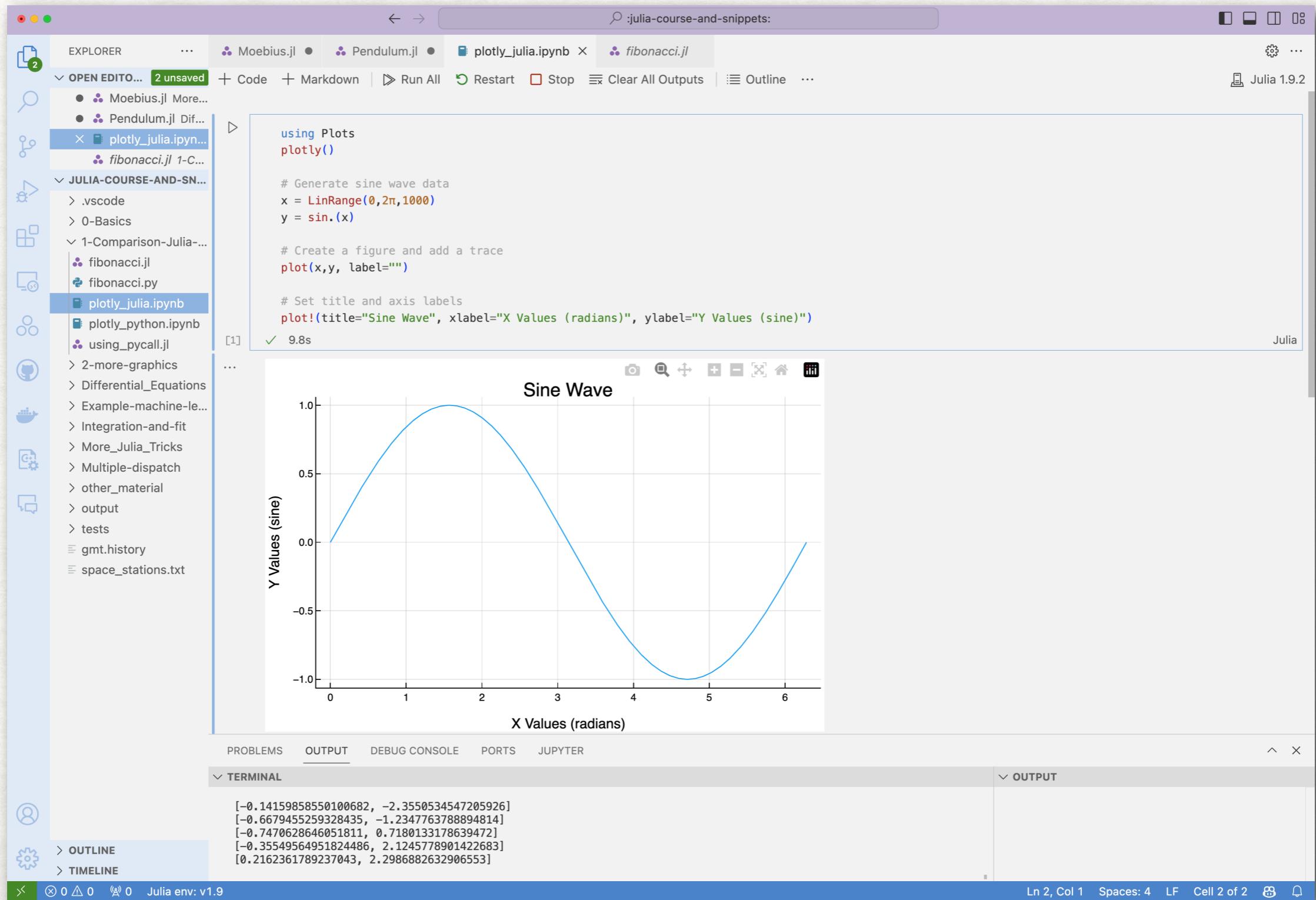
```
6
7 # Pendulum's equation of motion
8 function pendulum!(du, u, p, t)
9     θ, ω = u # u[1] = θ (angle), u[2] = ω (angular velocity)
10    du[1] = ω
11    du[2] = -(g/L) * sin(θ)
12 end | pendulum! (generic function with 1 method)
13
14 # Time span for the simulation
15 tspan = (0.0, 10.0) | (0.0, 10.0)
16
17 # Initial conditions
18 θ₀ = π / 4 # 45 degrees | 0.7853981633974483
19 ω₀ = 0 # initial angular velocity | 0
20
21 u₀ = [θ₀, ω₀] | 2-element Vector{Float64}:
22 prob = ODEProblem(pendulum!, u₀, tspan) | ODEProblem with uType Vector
23 sol = solve(prob) | retcode: Success
24
25 plot(sol, vars=(1), label="θ") | Plot{Plots.GRBackend()} n=1
26 plot!(sol, vars=(2), label="dθ/dt") | Plot{Plots.GRBackend()} n=2
27 plot!(title = "Pendulum with Initial Angle 45°", xlabel="t", ylabel=
28 plot!(frame=:box) | Plot{Plots.GRBackend()} n=2
29
```

```
9.754257362625822
10.0
u: 45-element Vector{Vector{Float64}}:
[0.7853981633974483, 0.0]
[0.7853980915004479, -0.0009987283555975368]
[0.7853894638729179, -0.010985980317064265]
[0.7845124512474613, -0.11082610982218406]
[0.7722623869618356, -0.4254834847069878]
:
[-0.14159858550100682, -2.3550534547205926]
[-0.6679455259328435, -1.2347763788894814]
[-0.7470628646051811, 0.7180133178639472]
[-0.35549564951824486, 2.1245778901422683]
[0.2162361789237043, 2.2986882632906553]
```

Bottom status bar: Ln 25, Col 15 (4 selected) Spaces: 4 UTF-8 LF Julia Main

Running Environments

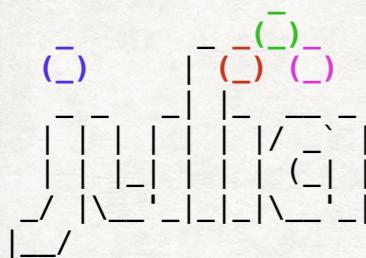
- Jupyter notebook (in VSCode)



Running Environments

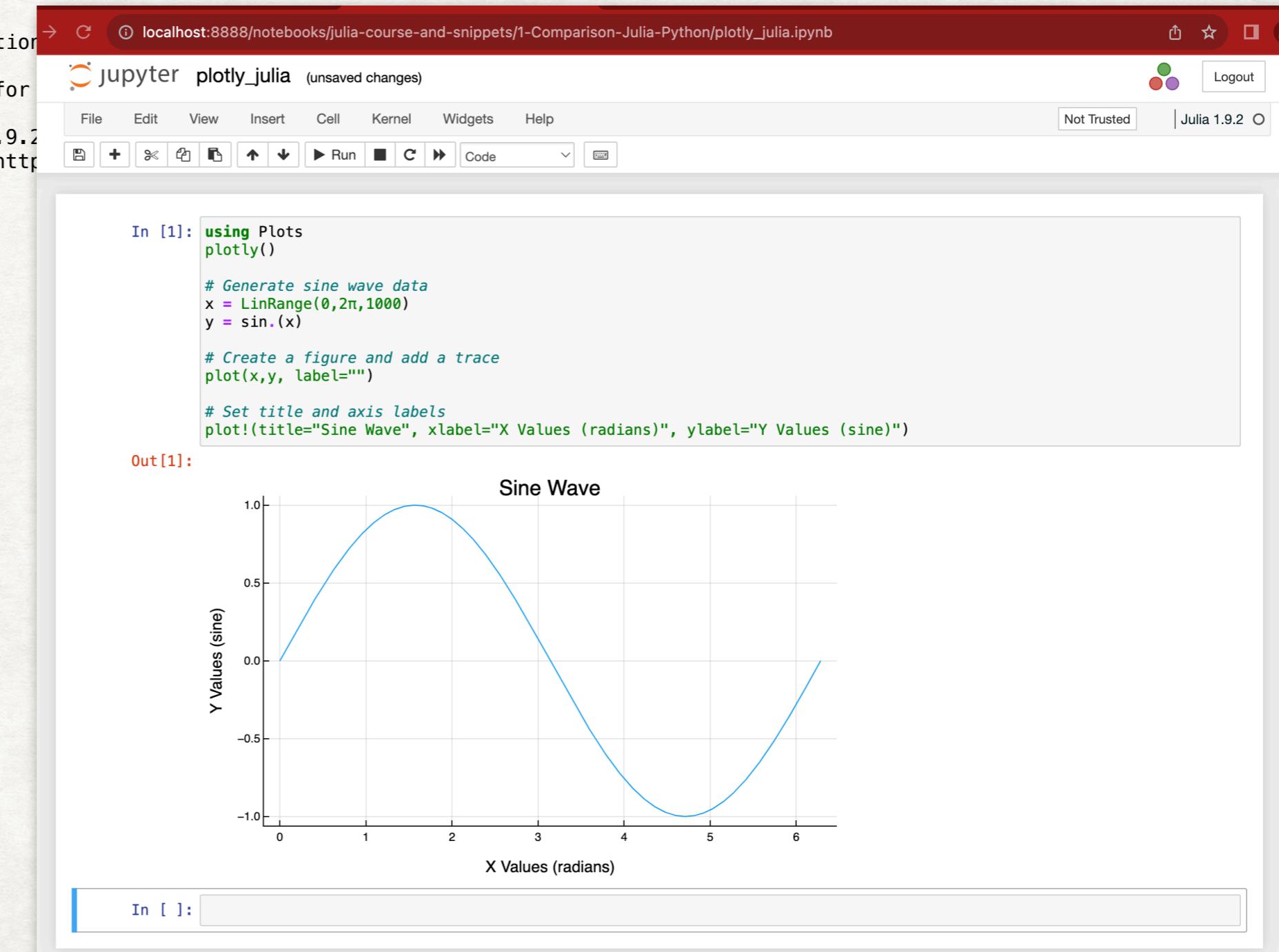
- Jupyter notebook (in browser)

```
shaviv@triton ~ % julia
```



```
julia> using IJulia
```

```
julia> notebook()
```



In [1]:

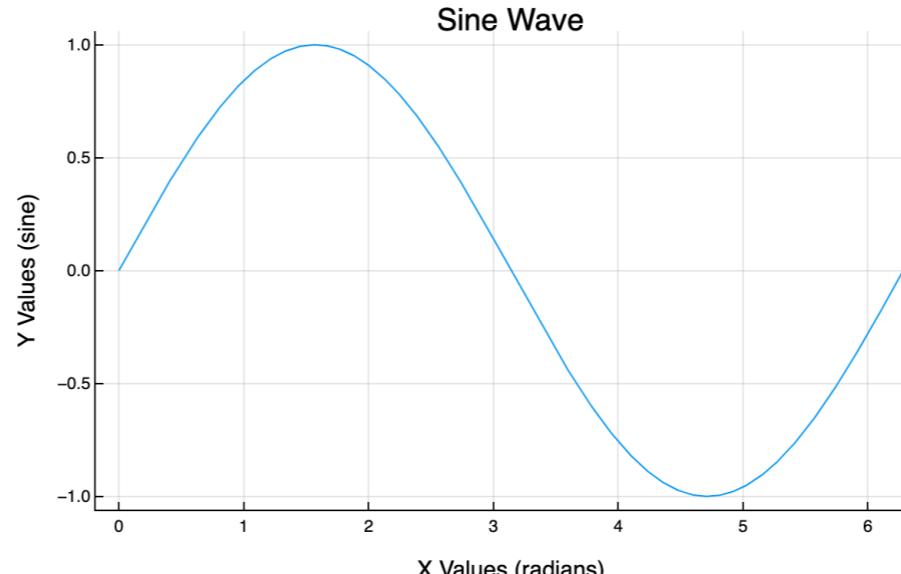
```
using Plots
plotly()

# Generate sine wave data
x = LinRange(0,2π,1000)
y = sin.(x)

# Create a figure and add a trace
plot(x,y, label="")

# Set title and axis labels
plot!(title="Sine Wave", xlabel="X Values (radians)", ylabel="Y Values (sine)")
```

Out [1]:



In []:

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

The screenshot shows the official Julia website at <https://julialang.org>. The top navigation bar includes links for Download, Documentation, Learn, Blog, Community, Contribute, and JSOC, along with a Sponsor button. The main content area is titled "Download Julia". It features a GitHub star count of 43,540 and a message encouraging users to star the project on GitHub, cite it, and consider sponsoring. A prominent section highlights the "Current stable release: v1.9.4 (November 14, 2023)". Below this, a table lists download links for various platforms, including Windows, macOS, Linux, and FreeBSD, along with source tarballs and GPG signatures.

Platform	Architectures	Links
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

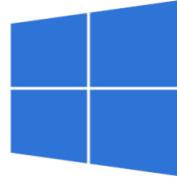
- Go to <https://code.visualstudio.com/download>
- Download and install!

Visual Studio Code Docs Updates Blog API Extensions FAQ Learn Search Docs [Download](#)

[Version 1.84](#) is now available! Read about the new features and fixes from October.

Download Visual Studio Code

Free and built on open source. Integrated Git, debugging and extensions.

[↓ Windows](#)
Windows 10, 11

[↓ .deb](#)
Debian, Ubuntu

[↓ .rpm](#)
Red Hat, Fedora, SUSE

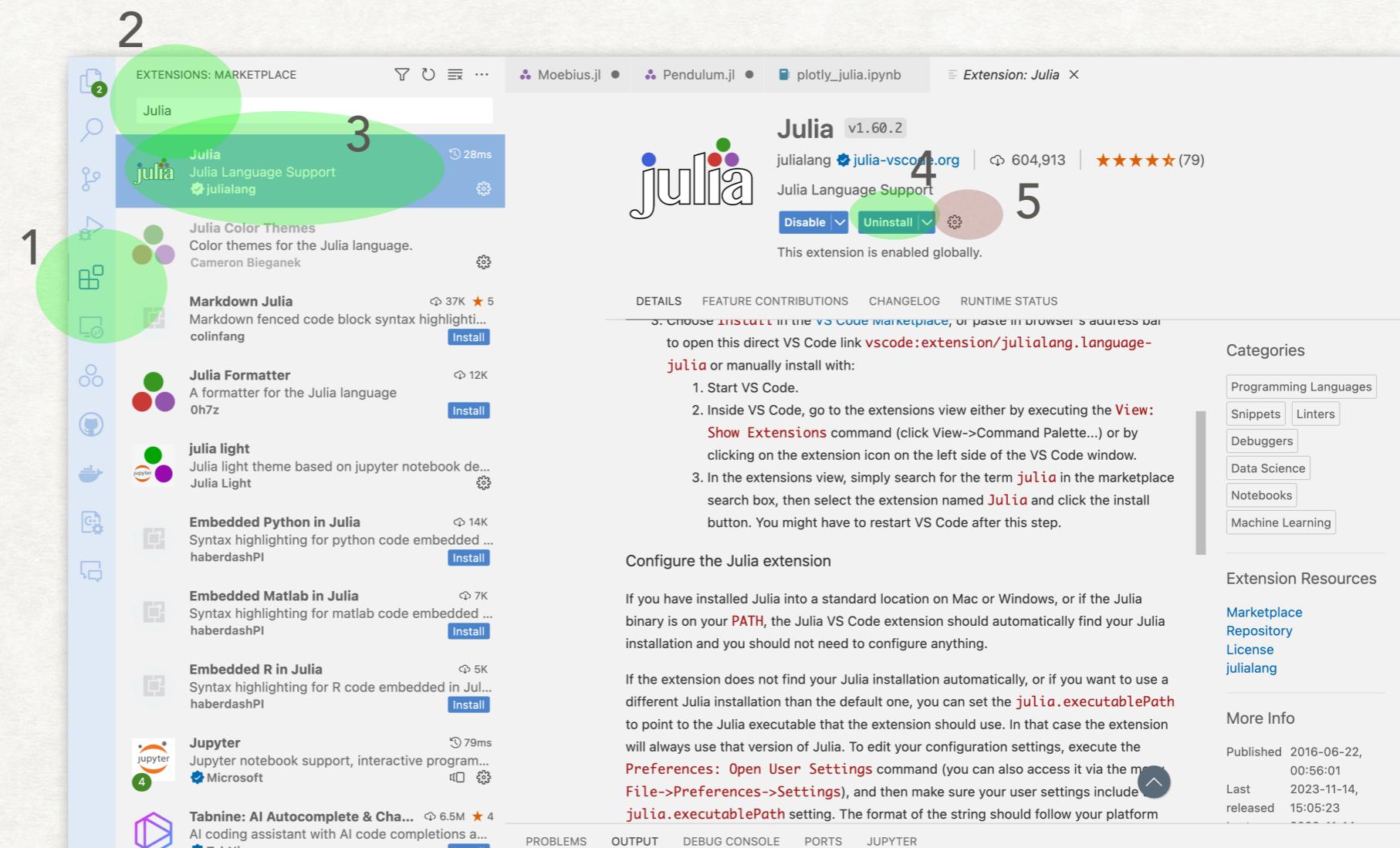
[↓ Mac](#)
macOS 10.15+

User Installer [x64](#) [Arm64](#)
System Installer [x64](#) [Arm64](#)
.zip [x64](#) [Arm64](#)
CLI [x64](#) [Arm64](#)

.deb [x64](#) [Arm32](#) [Arm64](#)
.rpm [x64](#) [Arm32](#) [Arm64](#)
.tar.gz [x64](#) [Arm32](#) [Arm64](#)
Snap [Snap Store](#)
CLI [x64](#) [Arm32](#) [Arm64](#)

.zip [Intel chip](#) [Apple silicon](#) [Universal](#)
CLI [Intel chip](#) [Apple silicon](#)

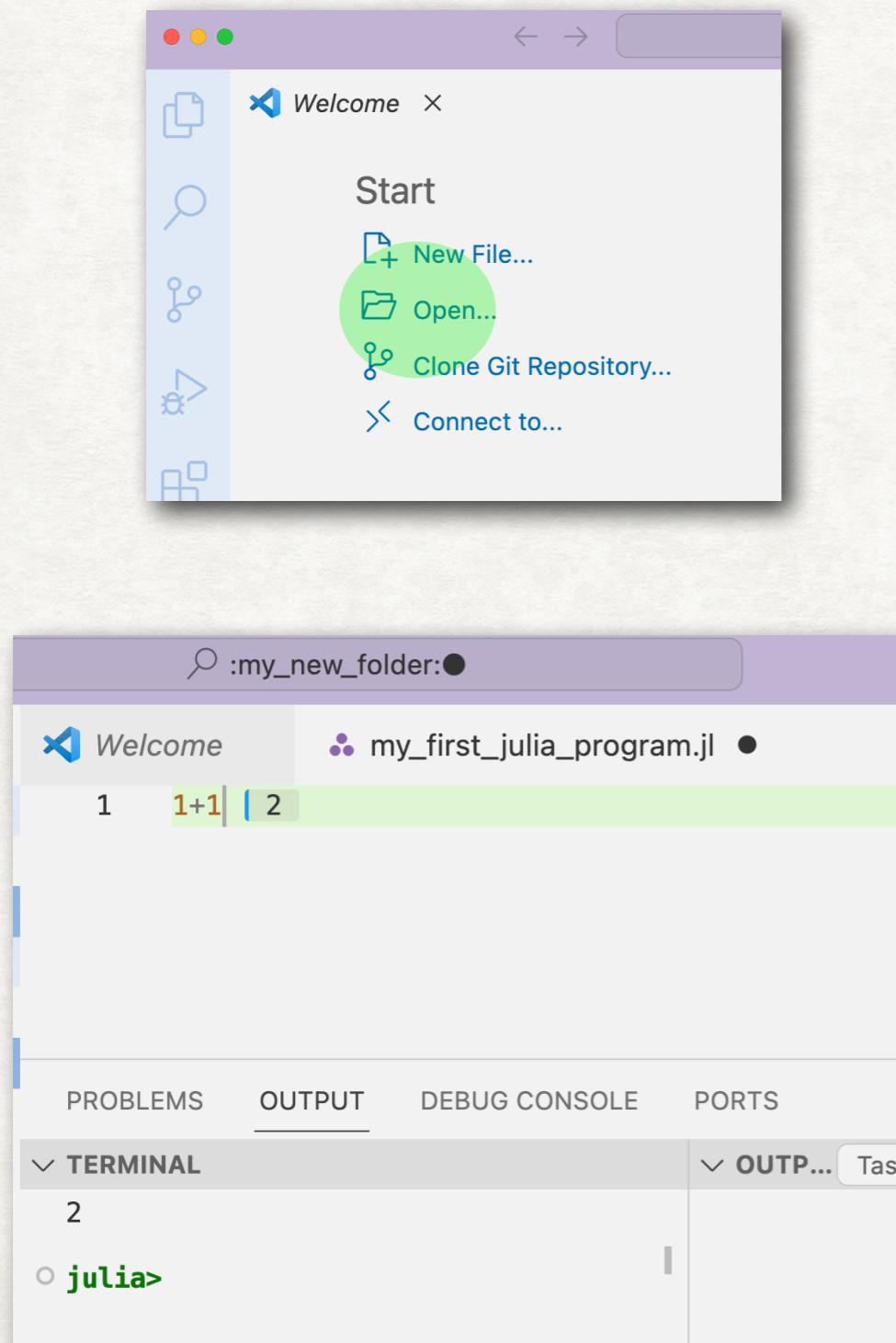
Installing Julia extension on VSCode



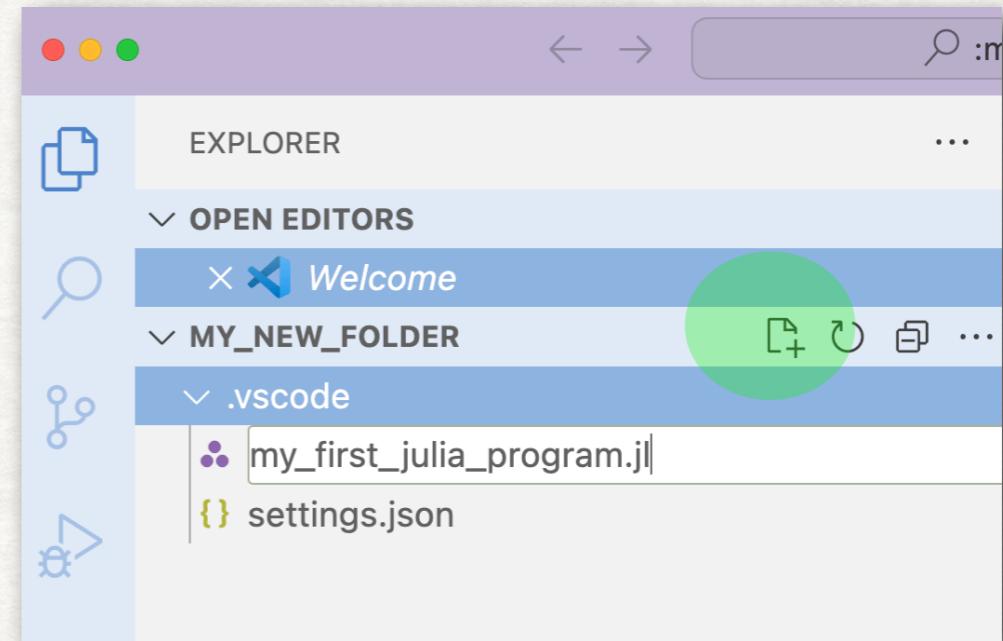
1. Go to extensions
2. Search “Julia”
3. Click Julia / Julia Language Support
4. Click install
5. Supposed to work. If not, you will need to change `julia.executablePath`

Open first julia file

- In VSCode: Open a new folder



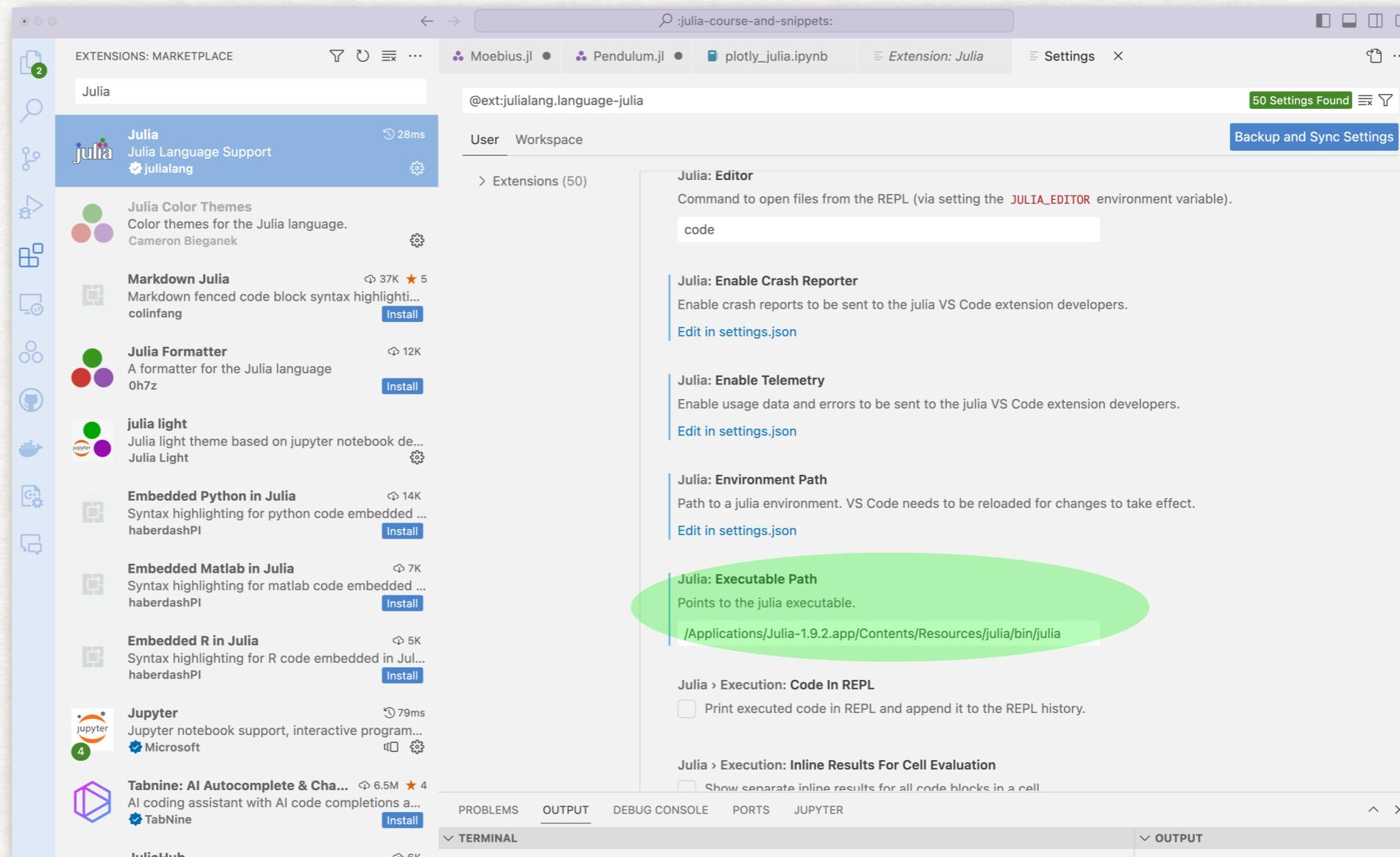
- Click on new file



- 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`