

# Julia: Your new secret romance?

Jonas Krimmer | Hack your Research @ ECOC 2023

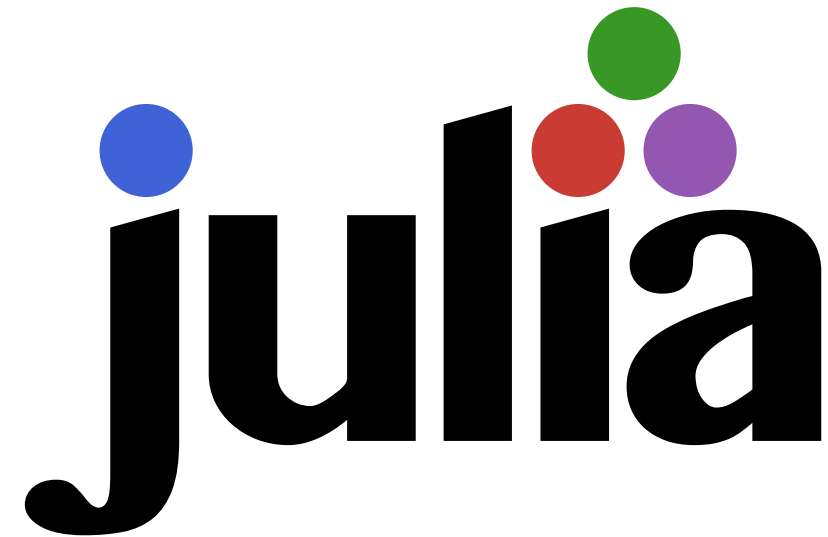
```
setheader(headers, "Transfer-Encoding") &&  
setheader(headers, "Upgrade")  
l = bodylength(body)  
if l == unknown_length  
    setheader(headers, "Content-Length" => string(l))  
elseif method == "GET" && iofunction isa Function  
    setheader(headers, "Content-Length" => "0")
```

# Who Is Julia?

- Dynamic programming language
- First release in 2012
- **Just-in-time compilation**  
implemented using LLVM
- Dynamical function dispatch based on  
the run-time (dynamic) argument  
type(s) → **Multiple dispatch**

```
f(x::Number) = 2x
f(x::String) = x * x

f(1) # → f(1) = 2
f("1") # → f("1") = "11"
```



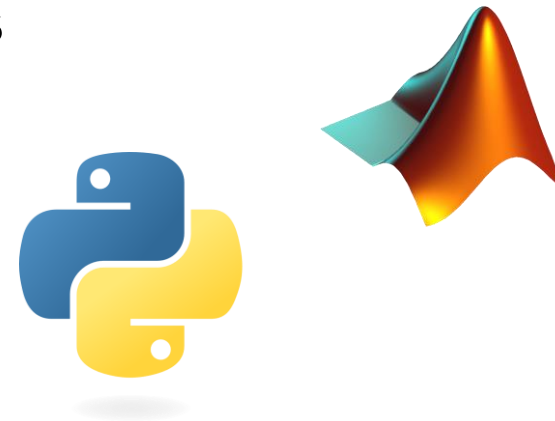
References:

[Bezanson, J. et al. SIREV 59, 2017](#)

Copyright (c) 2012-2022: [Stefan Karpinski](#), [CC BY-NC-SA 4.0](#), via [GitHub](#)

# Key Features of Julia

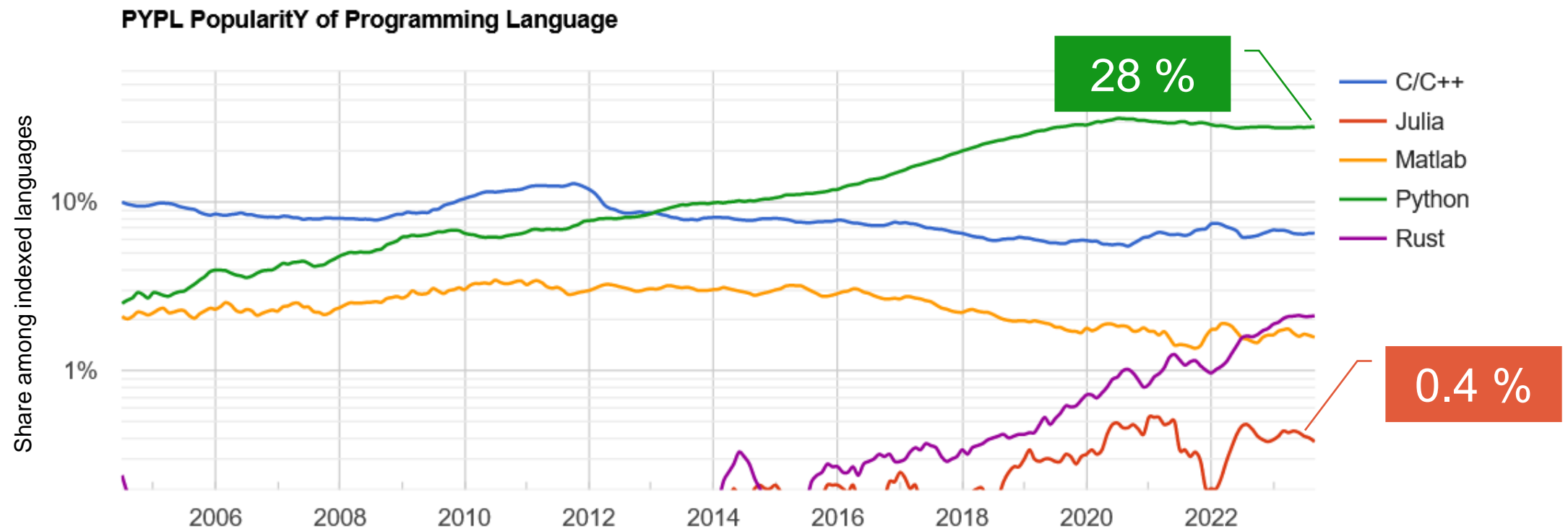
- No need to vectorize code for performance; **loops are fast!**
- Support for **parallelism** (multi-threading!) & distributed computation
- Call C functions without intermediate wrappers
- Macros and other **metaprogramming** facilities
- **Syntax** similar to Python or MATLAB
- High **interoperability** with Python



References:  
Python: [www.python.org](http://www.python.org), [GPL](https://www.gnu.org/licenses/gpl-3.0.html), via Wikimedia Commons  
Matlab: [Jarekt](https://www.mathworks.com/), Public domain, via Wikimedia Commons  
[Julia 1.9 Documentation, visited on 30.09.2023](#)

# Popularity of Programming Languages

PYPL popularity based on how often language tutorials are searched on Google



Reference: PYPL Popularity of Programming Language index, last visited on September 30, 2023. <https://pypl.github.io/PYPL.html>

# Why People Love Python?

- Simple and easy-to-learn **syntax**
- Powerful standard library
- Sheer endless number of **third-party packages**, e.g., numpy, tensorflow, ...
- Availability of countless **tutorials**
- Large community



⇒ **Jack of all trades, master of none?**

References:  
Numpy: [Isabela Presedo-Floyd, CC BY-SA 4.0](#), via Wikimedia Commons  
TensorFlow: [Google LLC](#), Public domain, via Wikimedia Commons

# Drawbacks of Python...

- Interpreted language → **significantly slower** than compiled languages
- Optimized external libraries such as numpy required for performance

```
import numpy as np
import timeit

x = [np.random.rand() for i in range(100_000)]
xn = np.array(x)

nruns = 100
Tpy = timeit.Timer(lambda: sum(x)).timeit(nruns) / nruns
Tnp = timeit.Timer(lambda: np.sum(xn)).timeit(nruns) / nruns
print("Runtime of pure-python sum (μs): " + str(Tpy * 1e6))
print("Runtime of numpy sum (μs): " + str(Tnp * 1e6))
print("Ratio pure-python/numpy: " + str(Tpy/Tnp))
```

```
Runtime of pure-python sum (μs): 639.9389999933192
Runtime of numpy sum (μs): 54.96499999935622
Ratio pure-python/numpy: 11.642663513159546
```

- Indents mess up code
- Zero-based indexing 🤪

## ... That Julia Avoids

- Just-in-time compiled code ⇒ **No external libraries** required for short runtime
- Reproduce summation example from python

```
x = [rand() for i in range(1, 100_000)]
nruns = 100
stats = @timed for i in range(1, nruns)
    sum(x)
end

display("Runtime (μs): " * string(stats.time / nruns * 1e6))
```

```
"Runtime (μs): 16.877000000000002"
```

⇒ About **3× faster** than numpy,  
almost **40× faster** than pure  
python!

- Indents **do not matter**, use end instead
- One-based indexing 😊



```
parent=nothing, iofunction=nothing, io... )  
DefaultHeader(headers, "Host" => url.host)  
if !hasheader(headers, "Content-Length") &&  
  hasheader(headers, "Transfer-Encoding") &&  
  hasheader(headers, "Upgrade")  
  l = bodylength(body)  
  if l => unknown_length  
    setheader(headers, "Content-Length" => string(l))  
  elseif method == "GET" && iofunction isa Function  
    setheader(headers, "Content-Length" => "0")  
  end  
end  
end  
request(method, target, headers, bodybytes(body);  
parent=parent, version=http_version)  
iofunction, io... )
```

## Live Demo 1: Simulation

Generation of phase screens for simulating light propagation through atmospheric turbulence

Reference: [McGlamery, Proc. SPIE 0074, 1976](#)



```
parent=nothing, iofunction=nothing, io... )  
DefaultHeader(headers, "Host" => url.host)  
if !headers["Content-Length"] &&  
  !headers["Transfer-Encoding"] &&  
  !headers["Upgrade"]  
  l = bodylength(body)  
  if l => unknown_length  
    setheader(headers, "Content-Length" => string(l))  
  elseif method == "GET" && iofunction isa Function  
    setheader(headers, "Content-Length" => "0")  
  end  
end  
request(method, target, headers, bodybytes(body);  
parent=parent, version=http_version)  
iofunction, io... )
```

## Live Demo 2: Digital-Signal Processing

Constant-Modulus-Algorithm for blind equalization of constant modulus payloads

Reference: [Godard, IEEE Trans. Commun. 28, 1980](#)

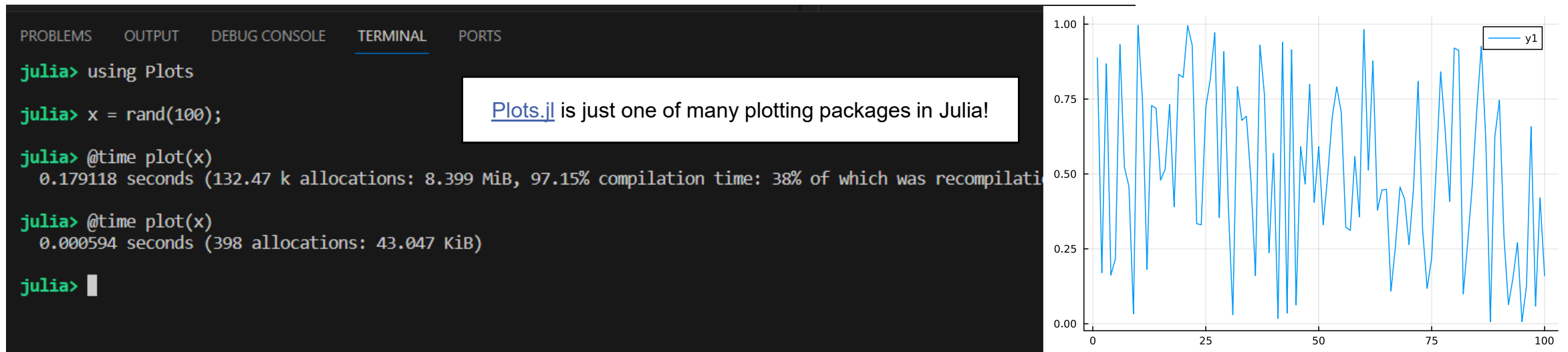
# Can You Substitute Julia for Python?

- Simple and easy-to-learn syntax? ✓
- Powerful standard library? ✓
- Sheer endless number of third-party packages? ✗
- Availability of countless tutorials? ✗
- Large community? ✗

⇒ **Clear answer:** It depends! 🤪

# Drawbacks of Julia That Python Avoids

- So-called „time to first plot“ (TTFP)
- Just-in-time compilation  $\Rightarrow$  Running functions for the first time slow
- $\Rightarrow$  **Initialize** with computationally less expensive trial before running simulations
- Luckily, TTFP keeps reducing with almost every Julia release



# Getting Started: Julia Installation

- Recommended approach: Cross-platform installer for Julia with version manager [juliaup](#)
- Manages updates and peaceful coexistence of multiple Julia versions

## Windows

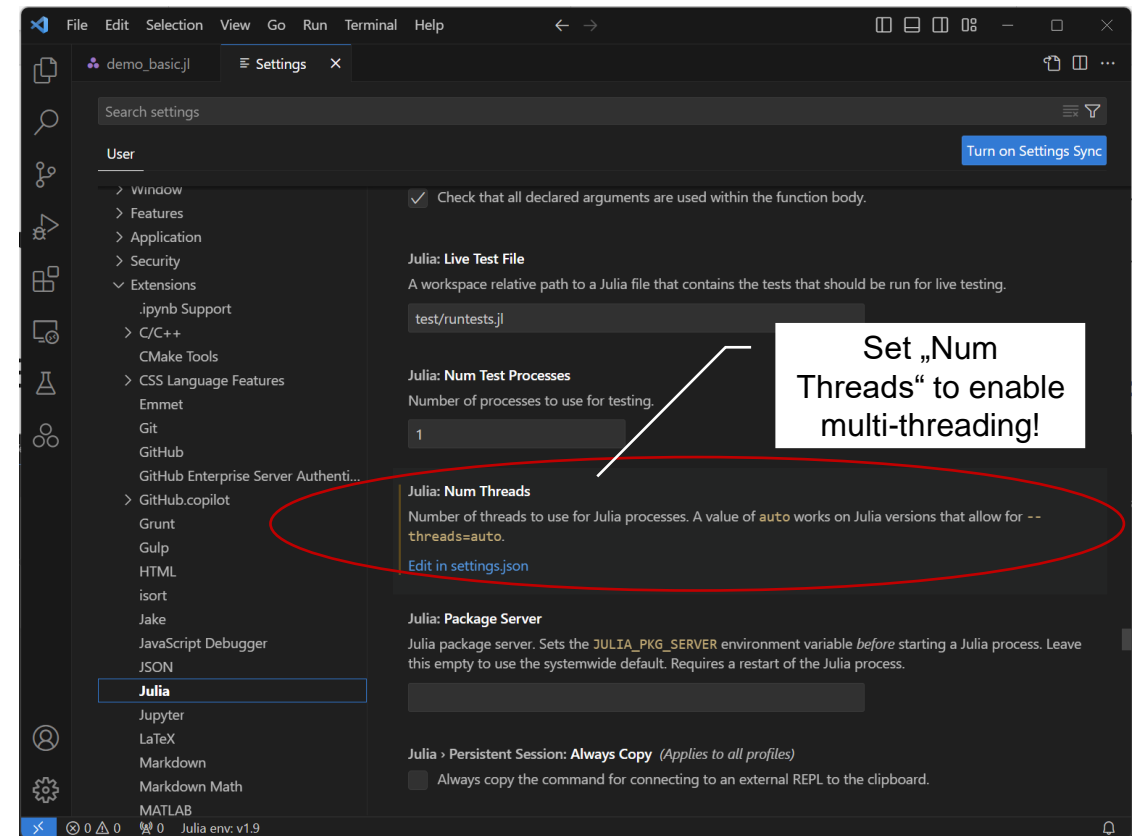
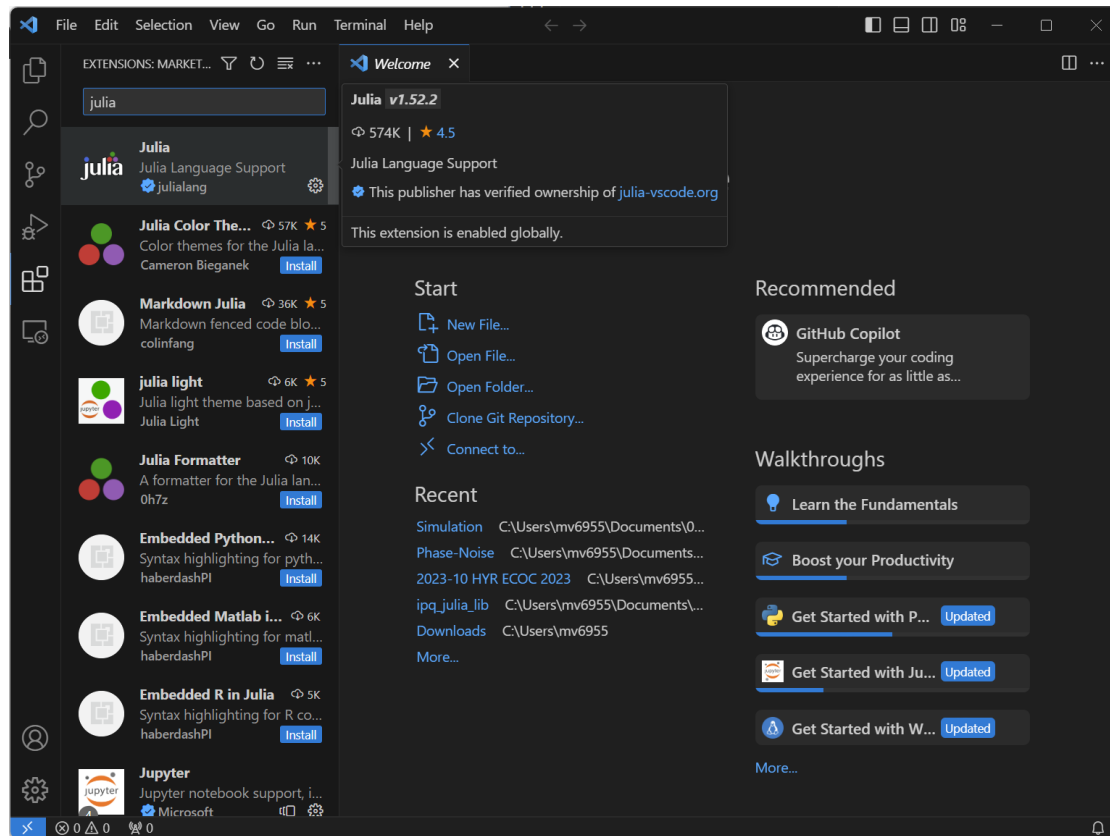
- Recommended: Install Julia and Juliaup directly from the Windows store
- Alternative (Terminal): `winget install julia -s msstore`

## Mac and Linux

- Execute the following command in a shell  
`curl -fsSL https://install.julialang.org | sh`

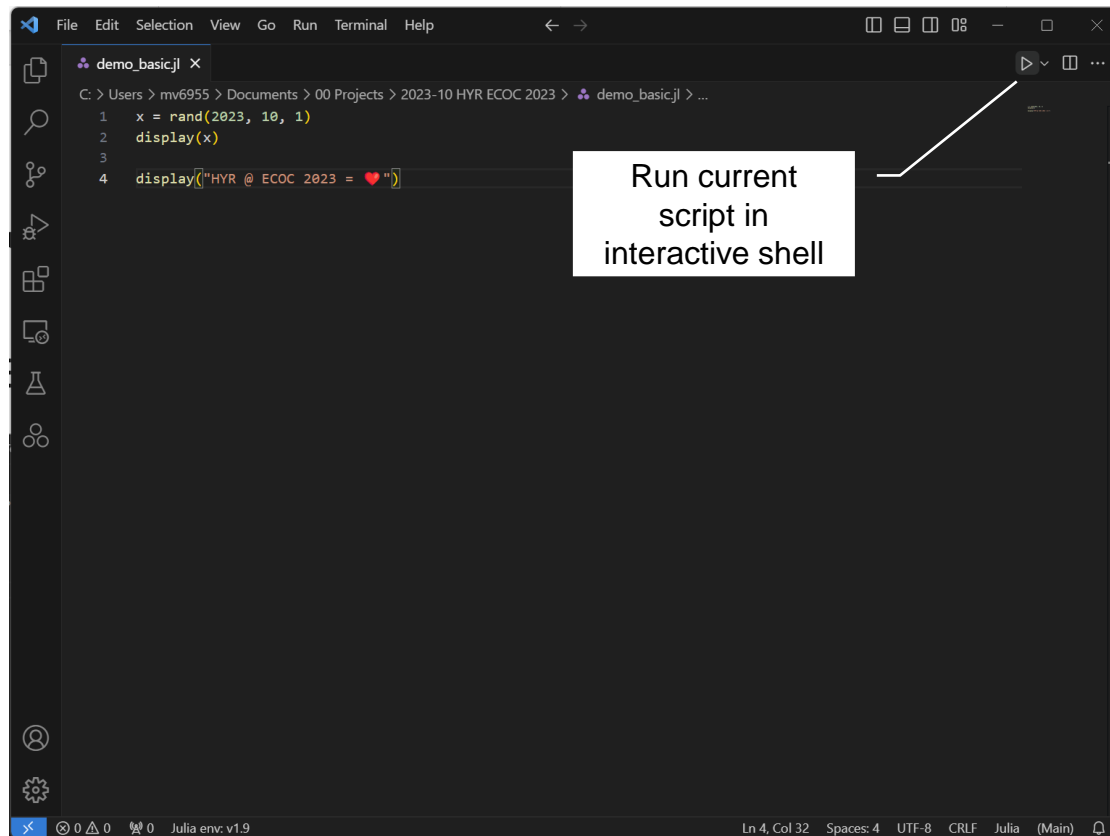
# Getting Started: Development Environment (I)

Recommended IDE: Visual Studio Code with „VS Code Julia extension“



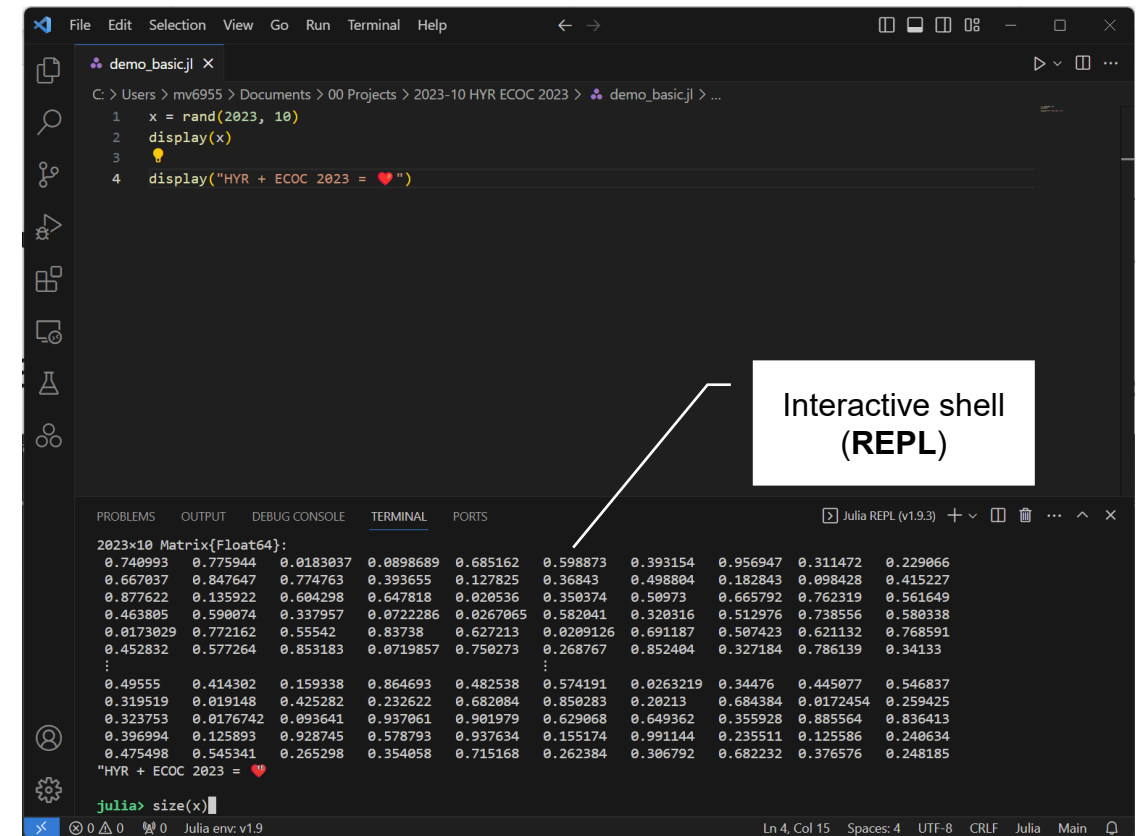
# Getting Started: Development Environment (II)

- Recommended IDE: [Visual Studio Code](#) with „[VS Code Julia extension](#)“



Run current script in interactive shell

```
demo_basic.jl
1 x = rand(2023, 10, 1)
2 display(x)
3
4 display("HYR @ ECOC 2023 = ❤️")
```



Interactive shell (REPL)

```
demo_basic.jl
1 x = rand(2023, 10)
2 display(x)
3
4 display("HYR + ECOC 2023 = ❤️")
```

PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL	PORTS
	2023x10 Matrix{Float64}:			
	0.740993	0.775944	0.0183037	0.0898689
	0.667037	0.847647	0.774763	0.393655
	0.877622	0.135922	0.604298	0.647818
	0.463805	0.590074	0.337957	0.0722286
	0.0173029	0.772162	0.55542	0.83738
	0.452832	0.577264	0.853183	0.0719857
	...			0.750273
	0.49555	0.414302	0.159338	0.864693
	0.319519	0.019148	0.425282	0.232622
	0.323753	0.0176742	0.093641	0.937061
	0.396994	0.125893	0.928745	0.578793
	0.475498	0.545341	0.265298	0.354058
				0.715168
				0.262384
				0.306792
				0.682232
				0.376576
				0.248185

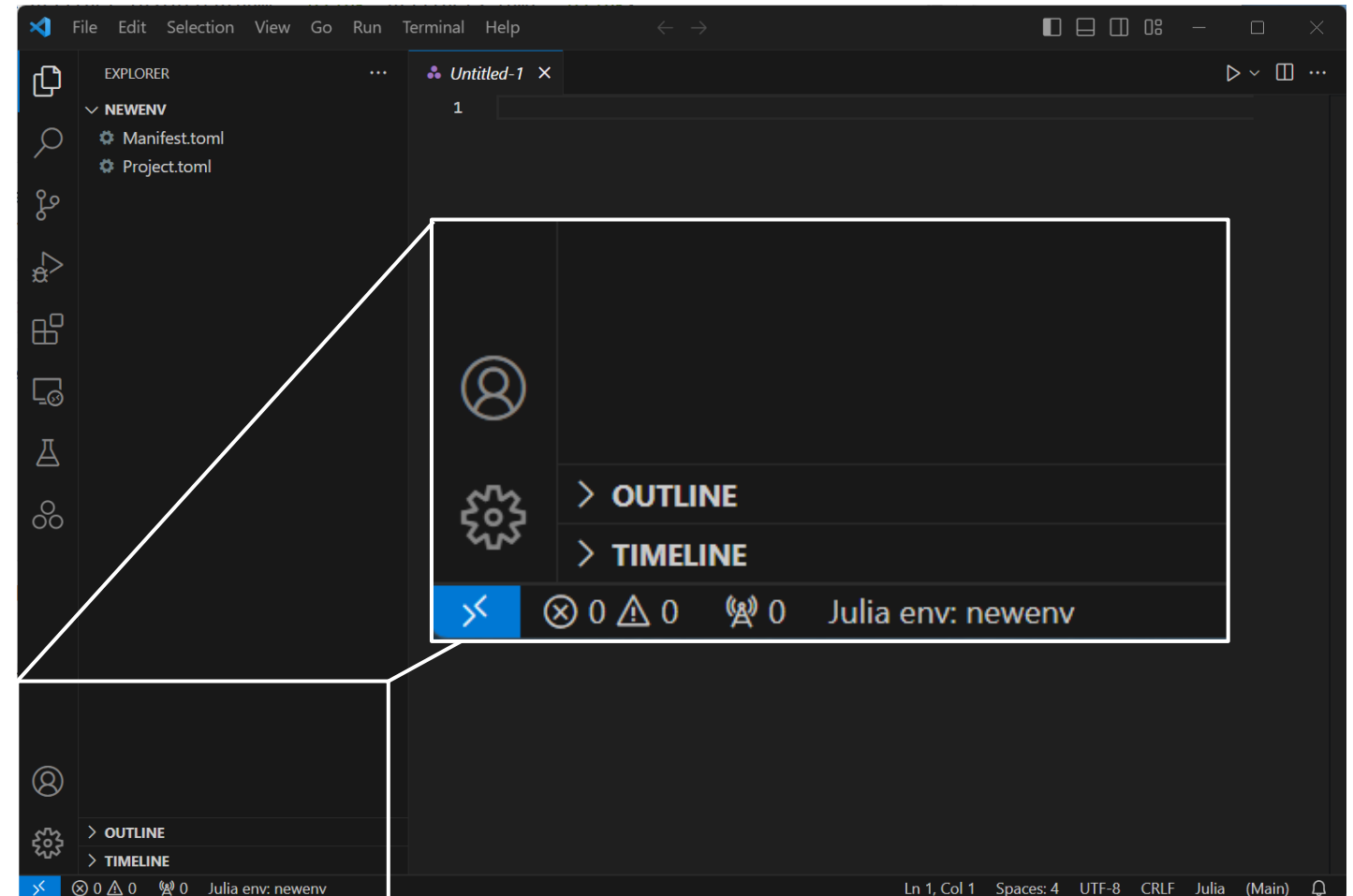
```
julia> size(x)
```





# Getting Started: Using Virtual Environments

- How can we use an environment in VS Code? 🤔
  1. „Open Folder“ in VS Code and choose the environment's directory
  2. Select the „Julia env“ from the VS Code bottom bar
- From the command line, you can activate any non-default environment <env> with `julia --project=<env>`



# Getting Started: Standard Library

- Among others, the rich standard library features the packages
  - **Dates**: Working with dates/time
  - **Distributed**: Tools for distributed parallel processing
  - **LinearAlgebra**: Linear algebra routines interfacing BLAS/LAPACK
  - **Random**: Advanced tools for random number generation
  - **SparseArrays**: Support for sparse vectors & matrices
- To use the functions provided by these libraries, load them into your workspace, e.g., with `using LinearAlgebra`
- If you prefer to keep these functions in a separate module, use `import LinearAlgebra` instead

# Getting Started: Using Packages

- To use the functions provided by these libraries, load them into your workspace, e.g., with

```
julia> using LinearAlgebra  
  
julia> det(ones(3,3))  
0.0
```

- If you prefer to keep these functions in a separate module, use `import LinearAlgebra` instead

```
julia> import LinearAlgebra  
  
julia> LinearAlgebra.det(ones(3,3))  
0.0
```

# Getting Started: Installing Packages

- The standard library does not provide FFT implementations or plotting...
- ... why we install [FFTW.jl](#) & [Plots.jl](#).
- On the REPL, activate the package manager with ]
- Input add FFTW, Plots and confirm with Enter
- Let's write an example script: `example_script_hyr.jl`

# Getting Started: Example Script

```
using FFTW
using Plots

# sampling frequency
R = 1e3
# observation time
T = 1
t = 0:1/R:T # or equivalently t = range(0, T, R+1)
ω0 = 2π * 200
# the dot-syntax is needed for broadcasting operations
x = sin.(ω0 .* t)
# number of samples
Ns = length(x)

x_hat = fft(x)
f = fftfreq(Ns, R)
plot(fftshift(f), fftshift(abs2.(x_hat)), label="|x̂|²")
```

