

Julia for Scientists

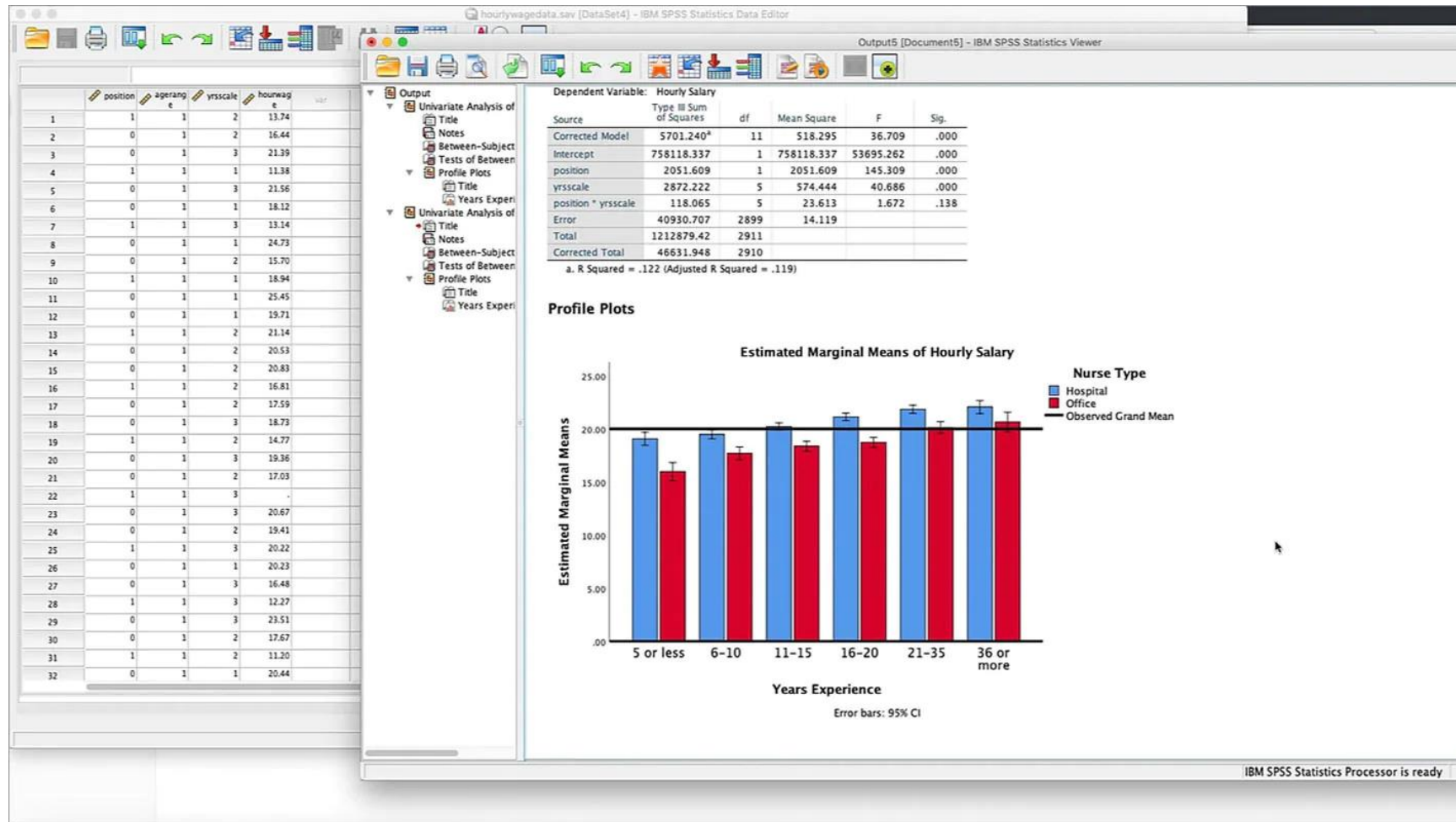
Tomas Fiers



Originally presented at the Lunchtime Data Club,
School of Psychology, University of Nottingham
Dec 7, 2022

If you're using Excel, SPSS, Stata, JASP, ...

Why learn a programming language?



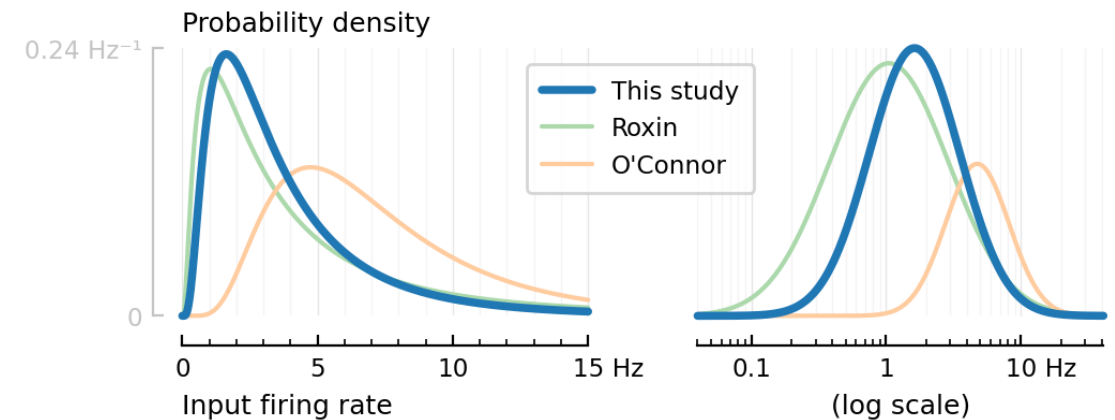
Why program?

- Automate analyses
→ less error-prone

..which goes hand in hand with:

- **Reproducibility**
- Customize:
 - Special plots
 - Tweak analyses
- Run simulations
- For fun

An example custom plot:



Choosing a programming language

as a researcher

	First released (The new builds on & learns from the old)	Free & open? (Hackable, “own your code-running environment”)	Online community (→ Learning resources & documentation iterations)
• Main choices:			
R	1995 / 1976 (S)	Yes	Huge
Python	1991	Yes	Huge
• Others			
Julia	2012	Yes	Medium
Matlab	1979	No	Large

- ..is also choosing a community



🤍❤️🤍🇺🇦 Victor
@vzverovich

Replying to @lefticus

Julia is Matlab without users

3:28 AM · Feb 21, 2022



Guillaume Dalle
@giomdal



Matlab is Julia without open source contributors 🧑



🤍❤️🤍🇺🇦 Victor @vzverovich · Feb 21

Replying to @lefticus

Julia is Matlab without users

8:59 AM · Mar 20, 2022

Julia syntax

```
using Unitful: MΩ, pF, mV           # Import names from a package

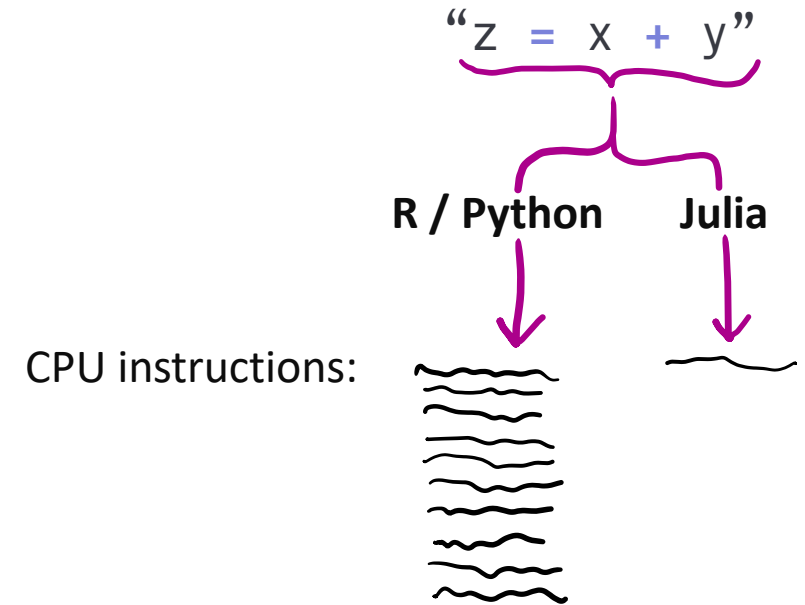
"""
Simulate a simple leaky-integrate-and-fire (LIF) neuron, given
input current `I` and a timestep `Δt`.
Return when the neuron fires its first spike.

The neuron's input resistance `R` and time constant `τ` can be
customized by keyword argument.
"""
function first_spike(I, Δt; R = 100MΩ, τ = 200pF)
    N = length(I)                     # Number of samples
    v = -70mV                         # Resting membrane potential
    for i in 1:N
        dv = -v + R*I[i]              # Leaky current integration
        v += dv/τ * Δt                 # Euler integration of ODE
        if v > -55mV                  # Spike!
            return time = i * Δt
        end
    end
    return nothing                    # Never spiked
end
```

In Python / R / Matlab:
“Avoid for-loops”
“Write vectorized code”

Compilation: Your code \rightsquigarrow the CPU

- If **one line** of Julia code corresponds to just a few CPU instructions
- ..then the same line in base Python / R / Matlab will often correspond to an order of magnitude **more CPU instructions** *
 - ..That's why the code that does the 'real' numeric work in these languages is actually written in C / C++
NumPy, PyTorch, Tensorflow, dplyr, ...: all have their core written in a different language
 - ..That's why, to have your code run fast, you're discouraged from writing `for`-loops for numeric code ..
 - .. and instead use the provided library functions
e.g. `np.where(...)`
 - Python is often used as "glue-code" (see next slide)
 - If you want a custom numeric algorithm that's not provided by the libraries, you need to learn C / C++
The "two languages-problem"



* Matlab added JIT compilation [in 2015](#) (but it's rather opaque)
Python can have JIT compilation via the fantastic **Numba** package. (But you can only use base Python with Numba, not arbitrary other packages).

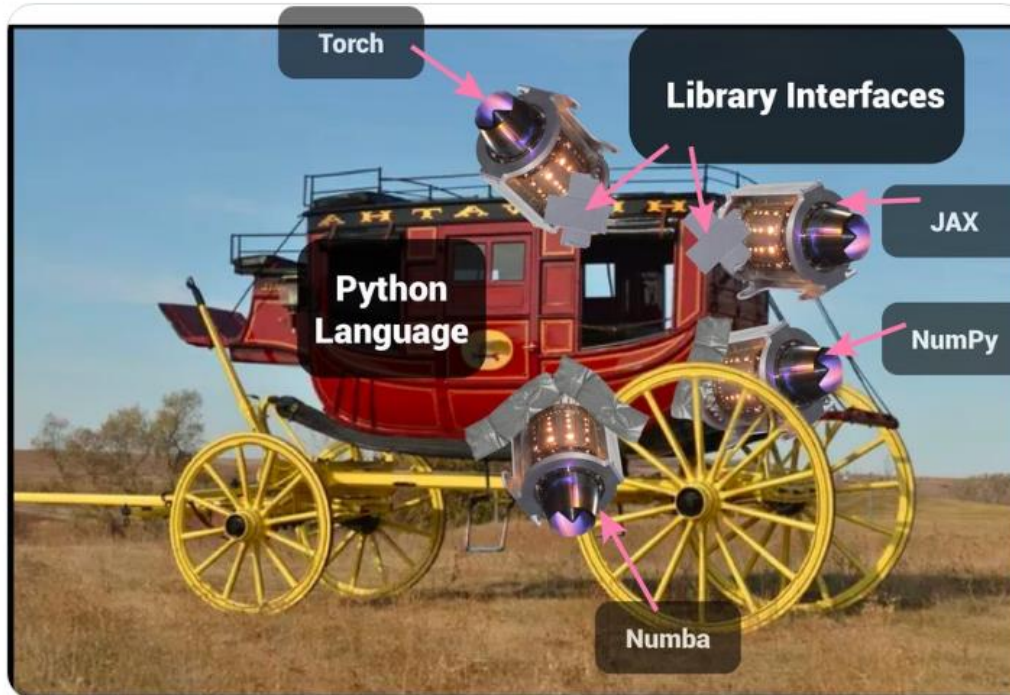


Miles Cranmer
@MilesCranmer



The more I use Julia, the more Python and its numeric libraries look like a Victorian-era stagecoach with jet engines duct-taped to it, each pointing a different direction (=mutually incompatible).

It's such a weird ecosystem, and makes it so much harder for users to contribute.



5:50 PM · Nov 7, 2022

JIT compilation

- If **one line** of Julia code corresponds to just a few CPU instructions
- ..then the same line in base Python / R / Matlab* will often correspond to an order of magnitude **more CPU instructions**
- Why is this ↑ ?
 - The same line of code (say, `z = x + y`) does different things, based on the **type** of `x` and `y`
 - If they're integers (`8 + 3`), use the ``leaq`` CPU instruction
 - If one is a float (`8 + 3.3`), call ``convert`` and use the floating point processor unit
 - If they're both *plots*, call subroutines, to compose the plots together into a bigger figure
 - ...
 - Python, R, and Matlab need to check the types of `x` and `y` *every time the line is run*, and then call the appropriate subroutines
 - Hence all these extra CPU instructions
 - Julia will ***infer*** the types of `x` and `y`
 - When? The first time the function that contains our line of code is called
 - It does this type inference based on the arguments that the function was called with (more specifically, their types), and by analyzing the function's source code you wrote
 - I then compile a fast version of the function
This is just-in-time (JIT) compilation

Data analysis in Julia

- DataFrames.jl
 - Tidyverse's dplyr & Python's Pandas equivalent
 - Better API than Pandas, imho
 - In the very capable hands of Bogumił Kamiński
 - Check out his tutorials:
github.com/bkamins/Julia-DataFrames-Tutorial
- Work in Jupyter notebooks
 - Via IJulia.jl
 - Ju stands for Julia (r for R).
- `missing` datatype is built-in in Julia
 - distinct from `nothing`
- I plot using Python's matplotlib 😊
 - Via PyPlot.jl
 - There's also Makie.jl
 - ..and Gadfly.jl, which is ggplot-inspired

Example of working with a DataFrame containing missing values, in a Jupyter notebook (loading data from Arrow, which is useful for data interchange with R or Python):

In [30]:

```
y = Arrow.Table("x.arrow") |> DataFrame
```

Out[30]: 3×4 DataFrame

Row	A	B	C	D
	Bool	Int64?	String?	Char?
1	true	1	missing	a
2	false	2	b	missing
3	true	missing	c	c

Source: https://github.com/bkamins/Julia-DataFrames-Tutorial/blob/master/04_loadsave.ipynb

Julia likes

- Unicode variable names & operators

- Easy input of LaTeX & Unicode names: ----->

- Plus reverse lookup: `help?> δ`
"δ" can be typed by `\delta<tab>`

```
julia> \partial<tab>
```

```
julia> ∂
```

- Some code is read much more than it is written. There, readability counts!
- For throwaway / exploratory code, not worth the slower input though
- Real-life example from my own code:

```
izh() = begin
    # Conductance-based synaptic current
    I_syn = g_e*(v-E_e) + g_i*(v-E_i)
    # Izhikevich 2D system
    Δ.v = (k*(v-v_l)*(v-v_t) - u - I_syn) / C # Membrane potential
    Δ.u = a*(b*(v-v_r) - u)                  # Adaptation current
    # Synaptic conductance decay
    Δ.g_e = -g_e / τ # (g_e is sum over all exc synapses)
    Δ.g_i = -g_i / τ
end
has_spiked() = (v ≥ v_s) ←----- Compact operator :)
on_self_spike() = begin
    v = v_r
    u += Δu
end
```

Julia likes

- Community
 - Discourse forum & Slack
 - Scientists
 - Contribute to ecosystem (open source, build upon others)
- As close-to-the-metal as you like
 - Look under the hood. Understand why something is slow/fast, and how it works
- “data structures + functions” design style
 - Decoupling is good
 - Versus: when you’re designing software in Python, you’re often pushed towards a coupled OOP design, with inheritance
- Keyword argument syntax sugar:
 - ```
options = [some object]
simulate(x, options = options) # Python
simulate(x; options) # Julia
```
- Inspectability
  - `@edit` to jump to source code of anything... amazing
  - `@code_native` to see cpu instructions
  - `?` for documentation
- Dependency management
  - Single, ergonomic tool ( $\leftrightarrow$  Python)
    - Pkg.jl, with `]` REPL mode
  - Easy **reproducibility** via thin environments
    - Project.toml & Manifest.toml
  - Not just for Julia code, for e.g. data too!
    - Artifacts.jl, DataDeps.jl
    - And for binaries: Yggdrasil & BinaryBuilder.jl
- Macro’s
  - Lisp-like. ‘Code as data’


# Julia annoyances

- **Package startup time** 🦴 (“time-to-first-plot”)
  - Language developers are working hard this year to improve this
- No winning plotting package yet
- ``name.<tab>`` autocompletion (API discovery) not as good as Python
  - “Power of the dot” in OOP languages
- Getting floats to print with lower precision is way more difficult than it should be for new users
- Traits / interfaces (lack of)
- Error handling is underdeveloped / under-practiced (“→ silent fails & crashes”)
- See also:
  - [yuri.is/not-julia](https://yuri.is/not-julia)
  - [danluu.com/julialang](https://danluu.com/julialang)
  - [viralinstruction.com/posts/badjulia](https://viralinstruction.com/posts/badjulia)

# “Julia has a correctness problem”

- (i.e. *there's nasty hidden bugs everywhere*)
- Not true for Base Julia:
  - every line there is pored over by many language developers
  - automatic test coverage is very comprehensive
- For other people's packages:
  - Not a problem in my experience.
  - But you have to inspect the packages that you use, if they're not in Julia Base; and make a value judgement about their quality
  - A lot of Julia packages are of *very* high quality in my experience
    - Except for the lack of error checking (of inputs and outputs)
      - Julia doesn't hold your hand:  
you gotta know what you're doing mathematically / numerically / statistically

# Why did I switch to Julia?

-  [Advent of Code](#) :) (2021)
- Physical units in neuron simulations:
- I could keep using:
  - my Jupyter notebook workflow
  - my Matplotlib experience

```
parameters = (
 # Izhikevich neuron
 C = 100 * pF
 k = 0.7 * (nS/mV)
 v_l = -60 * mV
 v_t = -40 * mV
 a = 0.03 / ms
 b = -2 * nS
 v_s = 35 * mV
 v_r = -50 * mV
 Δu = 100 * pA
 # Synapses
 E_e = 0 * mV
 E_i = -80 * mV
 τ = 7 * ms
 # Inputs
 N_e = 40
 N_i = 10
 N = N_e + N_i
 Δg_e = 60nS / N_e
 Δg_i = 60nS / N_i
 # Integration
 Δt = 0.1ms
 T = 10seconds
)
```

# julia Tips

- Code must be type-inferable (“type-stable”)
  - Put everything in (small) functions
  - If using globals: `const`, or typed
- Read the manual
  - Especially the “Performance tips” section, if you’re wondering why your code is not as fast as promised. Also:
- **Ask questions on the forum**
  - [discourse.julialang.org](https://discourse.julialang.org)
  - People are very eager to help, and the community managers do a great job
- Use `Revise.jl` (Use all of Tim Holy’s packages actually).
  - This minizes nr. of times you have to restart the Julia session (re: time-to-first-X problem)
  - Plus:
  - If using VS Code, there’s a plugin for Julia. Also: [the JuliaMono font](#) :) Example: ----->
  - On Windows, use the Julia REPL in the Windows Terminal
  - Put commonly used snippets in your `startup.jl`
- Don’t load unnecessary packages
  - Julia *Base* has no real latency (time-to-first-X) problem. It’s loading many packages that gets you
    - Especially packages that have many dependencies themselves (looking at you SciML ecosystem :P)
  - Do you really need this package?  
Can you just implement it yourself / copy the relevant part?
- Learn by doing
  - Like by doing some Advent of Code puzzles!

```
Code excerpt from the
JuliaMono homepage.
Original by Zygmunt Szpak
⊗ = kron
N = length(D[1])
 $\mathcal{M}, \mathcal{M}' = \mathcal{D}$
 $\Lambda_1, \Lambda_2 = \mathcal{C}$
 $\mathbf{e}_1 = \text{@SMatrix} [1.0; 0.0; 0.0]$
 $\mathbf{e}_2 = \text{@SMatrix} [0.0; 1.0; 0.0]$
for n = 1:N
 index = SVector(1,2)
 $\Lambda_n[1:2,1:2] = \Lambda_1[n][\text{index},:]$
 $\Lambda_n[3:4,3:4] = \Lambda_2[n][\text{index},:]$
 $\mathbf{m} = \text{hom}(\mathcal{M}[n])$
 $\mathbf{m}' = \text{hom}(\mathcal{M}'[n])$
 $\mathbf{U}_n = (\mathbf{m} \otimes \mathbf{m}')$
 $\partial_x \mathbf{u}_n = [(\mathbf{e}_1 \otimes \mathbf{m}') \ (\mathbf{e}_2 \otimes \mathbf{m}')$
 $\mathbf{B}_n = \partial_x \mathbf{u}_n * \Lambda_n * \partial_x \mathbf{u}_n'$
 $\Sigma_n = \boldsymbol{\theta}' * \mathbf{B}_n * \boldsymbol{\theta}$
 $\Sigma_n^{-1} = \text{inv}(\Sigma_n)$
 ...
end
```



# Should you use Julia?

- Do you ‘just’ need data analysis, automation, and pretty, customized plots?
  - Then, no
- Or do you also write custom numeric algorithms / simulations?
  - Then, yes :)
  - ..Unless you already know Matlab and don’t have the time
  - ..Plus, Python and R have huge ecosystems of packages that might already do your custom thing
    - A concrete example in computational neuroscience: **Brian** Python package for spiking neural network simulations (core written in C++)
    - Also, Python has **Numba** for JIT-optimization of hot inner loops ([numba.pydata.org](https://numba.pydata.org)). That might be enough for your use case

# Links

- [“Seven Lines of Julia”](#): examples of Julia, in different applications.
  - “What cool thing can you do in seven lines of code?”
- [tfiers.github.io/phd](https://tfiers.github.io/phd)
  - made with [JupyterBook](#)
  - auto-built and -published [with GitHub Actions on GitHub Pages](#)
- [github.com/schluppeck/ng-data-club](https://github.com/schluppeck/ng-data-club)
  - Repo of the Lunchtime data club
- [Discussion of these slides on Julia Discourse](#)
  - (woah meta)