Julia: Bridging the Gap in Technical Computing for Data Science & Beyond

Nick Uhorchak University of Arkansas

June 11, 2025

Outline

The Language Problem

- **A familiar pain point:** But C++ is faster than ... *(R, Python...)*.
- When performance is critical, often re-implementing in lower-level languages (C++, Fortran).
- This creates a development overhead and potential for bugs.
- Besides, who likes to write code twice!

What is Julia?

- High-level, high-performance **dynamic** programming language.
- Designed for numerical and scientific computing.
- **"Walks like Python, Runs like C."**
- Open-source and free.

Why Julia Now? Key Interests

- **Performance:** JIT compilation to native machine code (LLVM). Often competitive with C/Fortran.
- **Solves the "Language Problem":** Write high-performance code directly in Julia.
- **Mathematical Syntax:** Intuitive for mathematicians and engineers. Similar to python.
- **Dynamic & Flexible:** Interactivity of dynamic languages with performance of compiled ones.
- **Growing Ecosystem:** Rich set of packages for data science, ML, scientific computing.
- **General Purpose:** Beyond numerical computing, capable of web dev, scripting.

Julia vs Python/R: What Really Matters for Data Science

What do you want as a data scientist?

- Fast iteration: load, transform, visualize, model — quickly.
- Simple syntax and interactive experience (like Python or R).
- Seamless scaling: small prototype to big data or production.

Where Julia Fits:

Julia

- Feels like Python or R in the REPL.
- You get real speed without changing your code. (unless you code poorly anyways!)
- Same package handles exploration & production.
- DataFrames.jl, CSV.jl, MLJ.jl, Plots.jl — clean and expressive.

Python / R

- Excellent for exploration and prototyping.
- Performance usually depends on compiled extensions (NumPy, data.table).
- Scaling up often means changing tools.
- Language boundaries can complicate deployment.

Julia offers the simplicity of scripting with the performance of systems code — in a single, consistent toolchain.

Numbers and Arrays: Julia vs Python

- **Numbers:** Native support for various types (Integers, Floats, Complex, Rationals).
- **Arrays/Matrices:** Core for numerical computing.
- **One-based indexing:** (Similar to R/MATLAB, different from Python/C++).
- **Broadcasting ('.'):** Element-wise operations (similar to NumPy).

Julia

```julia
# Vector
v = [1, 2, 3, 4, 5]

# Matrix
M = [1 2; 3 4]

# Array comprehension
A = [i*j for i in 1:3, j in 1:3]
```

Python (NumPy)

```python
import numpy as np

v = np.array([1, 2, 3, 4, 5])
M = np.array([[1, 2], [3, 4]])
A = np.array([[i*j for j in range(1, 4)]
              for i in range(1, 4)])
```

Tuples and Dictionaries: Julia vs Python

- **Tuples:** Immutable, ordered collections.
- **Dictionaries (Dict):** Key-value pairs.

Julia

```julia
# Tuple
t = (1, "hello", 3.14)

# Dictionary
d = Dict("name" => "Julia",
         "version" => 1.9)
```

Python

```python
# Tuple
t = (1, "hello", 3.14)

# Dictionary
d = {"name": "Julia",
     "version": 1.9}
```

Tabular Data: Julia vs Python

- Equivalent to R's `data.frame` or Python's Pandas DataFrame.
- Efficient for manipulation, supports missing data.

Julia (DataFrames.jl)

```julia
using DataFrames

df = DataFrame(Name=["Alice", "Bob"],
               Age=[25, 30])

select(df, :Name)
filter(:Age => >(26), df)
```

Python (pandas)

```python
import pandas as pd

df = pd.DataFrame({
    "Name": ["Alice", "Bob"],
    "Age": [25, 30]
})

df[["Name"]]
df[df["Age"] > 26]
```

Control Flow Syntax: The Role of `end`

Julia requires explicit `end` **statements** to close control blocks like `for`, `if`, and `function`.

Julia

```julia
for i in 1:5
    println(i)
end
```

Python

```python
for i in range(1, 6):
    print(i)
```

R

```r
for (i in 1:5) {
  print(i)
}
```

- Julia uses `end` to clearly mark block boundaries.
- Python relies on indentation.
- R uses braces {} to define scope.

Statistics Libraries

- Statistics.jl, StatsBase.jl, Distributions.jl, HypothesisTests.jl

```julia
using Statistics, Distributions

data = randn(100)
println("Mean: $(mean(data))")
println("Std Dev: $(std(data))")

d = Normal(0, 1)
println("PDF at 0: $(pdf(d, 0.0))")
```

Machine Learning with MLJ.jl

- **MLJ.jl:** Unified interface to many ML models.
- Common API for EvoTrees, DecisionTree, Flux, etc.

```julia
using MLJ, EvoTrees, DataFrames
using MLJBase
@load EvoTreeClassifier pkg=EvoTrees

X = DataFrame(f1 = rand(100), f2 = rand(100))
y = categorical(rand(Bool, 100))

train, test = partition(eachindex(y), 0.7, shuffle=true)
Xtrain, Xtest = X[train,:], X[test,:]
ytrain, ytest = y[train], y[test]

model = EvoTreesClassifier()
mach = machine(model, Xtrain, ytrain)
fit!(mach, verbosity=0)

yhat = predict(mach, Xtest)
```

Visualization Libraries

- **Plots.jl**, **Makie.jl**, **StatsPlots.jl**

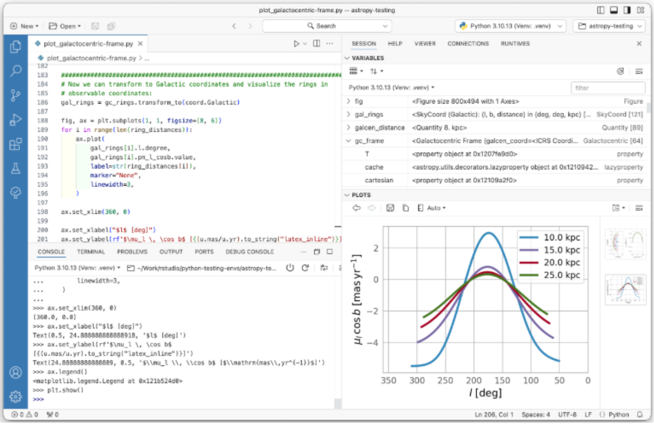
```julia
using Plots, StatsPlots

x = 1:10; y = rand(10)
plot(x, y, seriestype = :scatter,
     title = "My Scatter Plot",
     xlabel = "X-axis", ylabel = "Y-axis")

histogram(randn(1000), bins=50, title="Normal Distribution
    Histogram")
```

IDEs for Julia Development

- **VS Code with Julia Extension**
- **Jupyter Notebooks via IJulia.jl**
- **Julia REPL**
- *Positron (my favorite)*



Package Management: Julia vs Python vs R

How do I install and manage packages?

Feature	Julia	Python	R
Built-in environment management	`Pkg`	`venv`, `virtualenv`	`renv`, `packrat`
Third-party tools	–	`conda`, `pipenv`, `poetry`	`conda`, `checkpoint`
Dependency file	`Project.toml`	`requirements.txt`, `pyproject.toml`	`renv.lock`
Version pinning	`Manifest.toml`	`pip freeze`, `lock files`	`renv.lock`
Project isolation	✓	✓	✓
Ease of use	High	Medium–High	Medium

Table: Comparison of environment and package management across Julia, Python, and R

Best Practices: Writing Performant Julia Code

- Prefer functions over global scope.
- Ensure type-stable functions.
- Use `BenchmarkTools.jl` to profile performance.
- Embrace multiple dispatch.

Use Cases: Statistical & ML Analyses

- MCMC, simulation, ML pipelines.
- No need to rewrite in C++ for speed.
- Optimization

Use Cases: Plotting & Data Manipulation

- High-fidelity plots with Makie.
- Big-data manipulation with DataFrames.jl

Use Cases: Delivering Analytical Products

- Web apps with Genie.jl, dashboards, REST APIs.
- Compile to native executables.

Why Julia?

- **Data Scientists:** Faster iteration, production-ready.
- **Engineers:** Simulation, modeling.
- **Mathematicians:** *optimization*, linear algebra.
- **Computer Scientists:** Metaprogramming, compiler tools.

Live Code Demonstration

Live demonstration will show Julia in comparison to pyhton and R.

Questions & Discussion

Q&A

Resources

- julialang.org
- Julia Discourse
- JuliaHub
- JuliaAcademy, RCall.jl, PyCall.jl

Thank You!

Thank You! Questions? Comments?

nicholas.m.uhorchak.mil@army.mil