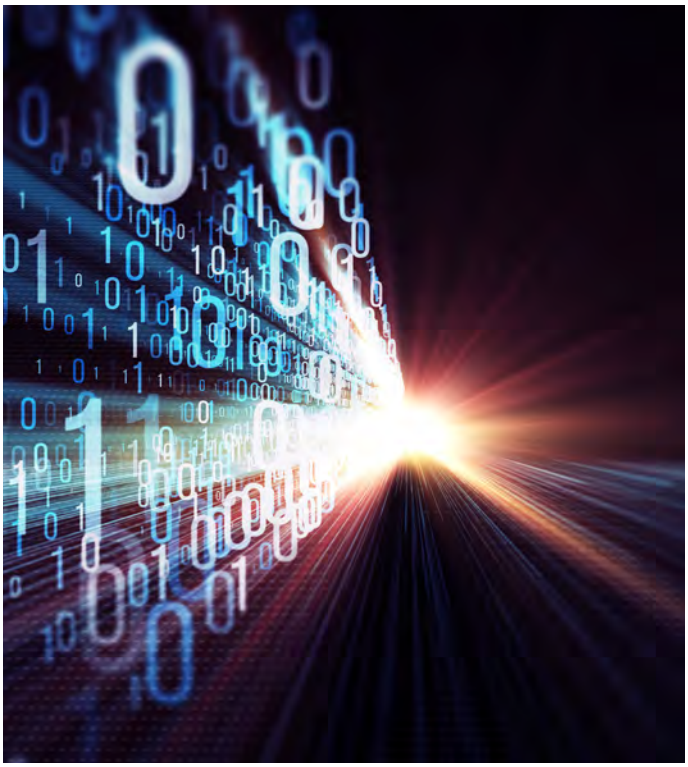
Article from

Actuary of the Future

October 2020

Julia for Actuaries

By Alec Loudenback



I have suggested that actuaries who are competent coders will differentiate both themselves and the companies they work for. Coding ability will be useful no matter what tools you utilize every day (e.g., Python/R/C++/etc. and associated packages) and all of those tools and communities contribute to moving actuarial processes out of the "Spreadsheet Age."

There's a newer programming language called Julia, and in this article, I'd like to state why Julia is worth considering for actuarial work.

JULIA OVERVIEW

Julia is relatively new[1], and **it shows**. It is evident in its pragmatic, productivity-focused design choices, pleasant syntax, rich ecosystem, thriving communities, and its ability to be both very general purpose and power cutting edge computing.

With Julia: math-heavy code **looks like math**; it's easy to pick up, and quick-to-prototype. Packages are well-integrated, with excellent visualization libraries and pragmatic design choices.

Julia's popularity continues to grow across many fields and there's a growing body of online references and tutorials, videos, and print media to learn from.

Large financial services companies have already started realizing gains: BlackRock's Aladdin portfolio modeling, the Federal Reserve's economic simulations, and Aviva's Solvency II-compliant modeling. The last of these has a great talk on YouTube by Aviva's Tim Thornham, which showcases an on-the-ground view of what difference the right choice of technology and programming language can make. Moving from their vendor-supplied modeling solution was **1000x faster, took 1/10 the amount of code, and was implemented 10x faster**.[2]

The language is not just great for data science—but also modeling, ETL, visualizations, package control/version management, machine learning, string manipulation, web-backends, and many other use cases.

FOR THE ACTUARY

Julia is well suited for actuarial work: easy to read and write and very performant for large amounts of data/modeling.

Expressiveness and Syntax

Expressiveness is the **manner in which** and **scope of** ideas and concepts that can be represented in a programming language. **Syntax** refers to how the code **looks** on the screen and its readability.

In a language with high expressiveness and pleasant syntax, you:

- Go from idea in your head to final product faster.
- Encapsulate concepts naturally and write concise functions.
- Compose functions and data naturally.
- Focus on the end-goal instead of fighting the tools.

Expressiveness can be hard to explain, but perhaps two short examples will illustrate.

Example: Retention Analysis

This is a really simple example relating Cessions, Policys, and Lives to do simple retention analysis.

First, let's define our data:

```
# Define our data structures
struct Life
  policies
end

struct Policy
  face
  cessions
 end

struct Cession
  ceded
end
```

Now to calculate amounts retained. First, let's say what retention means for a `Policy`:

```
# define retention
function retained(pol::Policy)
  pol.face - sum(cession.ceded for
  cession in pol.cessions)
end
```

And then what retention means for a `Life`:

```
function retained(l::Life)
  sum(retained(policy) for policy in
life.policies)
end
```

It's almost exactly how you'd specify it in English. No joins, no boilerplate, no fiddling with complicated syntax. You can express ideas and concepts the way that you think of them, not, for example, as a series of dataframe joins or as row/column coordinates on a spreadsheet.

We defined `retained` and adapted it to mean related, but different things depending on the specific context. That is, we didn't have to define `retained_life(...)` and `retained_pol(...)` because Julia can be **dispatched** based on what you give it. This is, as some would call it, unreasonably effective.

Let's use the above code in practice then.

The `julia>` syntax indicates that we've moved into Julia's interactive mode (REPL mode):

```
# create two policies with two and one
cessions respectively
julia> pol_1 = Policy( 1000, [
Cession(100), Cession(500)] )

julia> pol_2 = Policy( 2500, [
Cession(1000) ] )

# create a life, which has the two
policies
julia> life = Life([pol_1, pol_2])
```

```
julia> retained(pol_1)
400
```

```
julia> retained(life)
1900
```

And for the last trick, something called "broadcasting," which automatically vectorizes any function you write, no need to write loops or create if statements to handle a single versus repeated case:

```
julia> retained.(life.policies) # re-
tained amount for each policy
[400 ,  1500]
```

Example: Random Sampling

As another motivating example showcasing multiple dispatch, here's random sampling in Julia, R, and Python.

We generate 100:

- Uniform random numbers
- standard normal random numbers
- Bernoulli random number
- Random samples with a given set

Julia	R	Python

`using Distributions`
`rand(100)`
`rand(Normal(), 100)`
`rand(Bernoulli(0.5), 100)`
`rand(["Preferred", "Standard"],100)`
	`runif(100)`
	`rnorm(100)`
	`rbern(100,0.5)`
	`sample(c("Preferred", "Standard"), 100, replace=TRUE)`
		`import scipy.stats as sps`
		`import numpy as np`
		
		`sps.uniform.rvs(size=100)`
		`sps.norm.rvs(size=100)`
		`sps.bernoulli.rvs(p=0.5,size=100)`
		`np.random.choice (["Preferred","Standard"],size=100)`

By understanding the different types of things passed to `rand()`, it maintains the same syntax across a variety of different scenarios. We could define `rand(Cession)` and have it generate a random `Cession` like we used above.

The Speed

Julia is also fast. As the journal *Nature* said, "Come for the Syntax, Stay for the Speed."

Recall the Solvency II compliance that ran 1000x faster than the prior vendor solution mentioned earlier: what does it mean to be 1000x faster at something? It's the difference between something taking **10 seconds** instead of **three hours**—or **one hour** instead of **42 days**.

What analysis would you like to do if it took less time? A stochastic analysis of life-level claims? Machine learning with your experience data? Daily valuation instead of quarterly?

Speaking from experience, speed is not just great for production time improvements. During development, it's really helpful too. When building something, I can see that I messed something up in a couple of seconds instead of 20 minutes. The build, test, fix, iteration cycle goes faster this way.

Admittedly, most workflows don't see a 1000x speedup, but 10x to 1000x is a very common range of speed differences versus R or Python or MATLAB.

Sometimes you will see less of a speed difference; R and Python have already circumvented this and written much core code in low-level languages. This is an example of what's called the "two-language" problem where the language productive to write in isn't very fast. For example, more than half of R packages use C/C++/Fortran and core packages in Python like Pandas, PyTorch, NumPy, SciPy, etc., do this too.

Within the bounds of the optimized R/Python libraries, you can leverage this work. Extending it can be difficult: what if you have a custom retention management system running on millions of policies every night?

Julia packages you are using are almost always written in pure Julia: you can see what's going on, learn from them, or even contribute a package of your own!

More of Julia's Benefits

Julia is easy to write, learn, and be productive in:

- It's free and open-source
  - » Very permissive licenses, facilitating the use in commercial environments (same with most packages)
- Large and growing set of available packages
- Write how you like because it's multi-paradigm: vectorizable (R), object-oriented (Python), functional (Lisp), or detail-oriented (C)
- Built-in package manager, documentation, and testing-library
- Jupyter Notebook support (it's in the name! **Ju**lia-**Pyt**hon-**R**)
- Many small, nice things that add up:
  - » Unicode characters like α or β
  - » Nice display of arrays
  - » Simple anonymous function syntax
  - » Wide range of text editor support
  - » First-class support for missing values across the entire language
  - » Literate programming support (like R-Markdown)
- Built-in Dates package that makes working with dates pleasant
- Ability to directly call and use R and Python code/packages with the `PyCall` and `RCall` packages
- Error messages are helpful and tell you what line the error came from, not just the type of error
- Debugger functionality so you can step through your code line by line

For power-users, advanced features are easily accessible: parallel programming, broadcasting, types, interfaces, metaprogramming, and more.

These are some of the things that make Julia one of the world's most loved languages on the StackOverflow Developer Survey.

For those who are enterprise-minded: in addition to the liberal licensing mentioned above, there are professional products from organizations like Julia Computing that provide hands-on support, training, IT governance solutions, behind-the-firewall package management, and deployment/scaling assistance.

The Tradeoff

Julia is fast because it's compiled, unlike R and Python where (loosely speaking) the computer just reads one line at a time. Julia compiles code "just-in-time": right before you use a function for the first time, it will take a moment to pre-process the code section for the machine. Subsequent calls don't need to be re-compiled and are very fast.

A hypothetical example: running 10,000 stochastic projections where Julia needs to precompile but then runs each 10x faster:

- Julia runs in two minutes: the first projection takes one second to compile and run, but each 9,999 remaining projections only take 10ms.
- Python runs in 17 minutes: 100ms of a second for each computation.

Typically, the compilation is very fast (milliseconds), but in the most complicated cases it can be several seconds. One of these is the "time-to-first-plot" issue because it's the most common one users encounter: super-flexible plotting libraries have a lot of things to pre-compile. So, in the case of plotting, it can take several seconds to display the first plot after starting Julia, but then it's remarkably quick and easy to create an animation of your model results. The time-to-first plot is a solvable problem that's receiving a lot of attention from the core developers and will get better with future Julia releases.

For users working with a lot of data or complex calculations (like actuaries!), the runtime speedup is worth a few seconds at the start.

Package Ecosystem

Using packages as dependencies in your project is assisted by Julia's bundled package manager.

For each project, you can track the **exact** set of dependencies and replicate the code/process on another machine or another time. In R or Python, dependency management is notoriously difficult and it's one of the things that the Julia creators wanted to fix from the start.

Packages can be one of the thousands of publicly available, or private packages hosted internally behind a firewall.

Another powerful aspect of the package ecosystem is that due to the language design, packages can be combined/extended in ways that are difficult for other common languages. This means that Julia packages often interop without any additional coordination.

For example, packages that operate on data tables work without issue together in Julia. In R/Python, many features tend to come bundled in a giant singular package like Python's Pandas which has Input/Output, date manipulation, plotting, resampling, and more. There's a new Consortium for Python Data API Standards that seeks to harmonize the different packages in Python to make them more consistent (R's Tidyverse plays a similar role in coordinating their subset of the package ecosystem).

In Julia, packages tend to be more plug-and-play. For example, every time you want to load a CSV you might not want to transform the data into a dataframe (maybe you want a matrix or a plot instead). To load data into a dataframe, in Julia the practice is to use both the CSV and DataFrames packages, which help separate concerns. Some users may prefer the Python/R approach of less modular but more all-inclusive packages.

Some highlighted/recommended packages:

- Actuarial Specific (part of the *JuliaActuary.org* umbrella[3])
  - **MortalityTables**—Common tables and parametric models with survivorship calculations
  - **ActuaryUtilities**—Robust and fast calculations for common functions
  - **LifeContingencies**—Insurance, annuity, premium, and reserve maths.
- Data Science and Statistics
  - **DataFrames**—Work with datasets; similar to R's data.table and able to handle much larger datasets than Python's Pandas[4]
  - **Distributions**—Common and exotic statistical distributions
  - **GLM**—Generalized Linear Models
  - **Turing**—Bayesian statistics like STAN
  - **Gen**—Probabilistic programming
  - **OnlineStats**—Single-pass algorithms for real-time/large data
  - **CSV**—The fastest CSV reader
  - **ODBC**—Database Connections
  - **Dates**—Robust date types and functions
- Machine Learning
  - **Flux**—Elegant, GPU-powered ML
  - **Knet**—Deep learning framework
  - **MLJ**—ML models
- Notebooks
  - **IJulia**—the Julia kernel for Jupyter notebooks
  - **Pluto**—reactive/interactive notebooks that address some of the biggest complaints with Jupyter
- Visualization
  - **Plots**—Powerful but user-friendly plots and animations
  - **Queryverse**—Tidyverse-like data manipulation and plotting

- Dashboards
  - **Plot.ly Dash**
- Miscellaneous
  - **Optim**—Uni/Multivariate function optimization
  - **LinearAlgebra**—Built-in library for working with arrays/matrices
  - **JuMP**—Linear, Nonlinear, and other advanced optimization
  - **CUDA**—GPU programming made easier
  - **Revise**—Edit code while you work on it
- Interoperability
  - **PyCall**—use existing Python code/libraries inside Julia
  - **RCall**—use existing R code/libraries inside Julia
- Web
  - **HTTP**—Core web utilities
  - **Genie**—Full application framework
  - **Franklin**—Flexible Static Site Generator
- Documentation
  - **Weave/Literate**—Literate programming like RMarkdown
  - **Documenter**—Write your documentation as comments to your code and produce full docpages

And finally, some general resources to get started:

- JuliaLang.org, the home site with the downloads to get started and links to learning resources.
- JuliaHub indexes open-source Julia packages and makes the entire ecosystem and documentation searchable from one place.
- JuliaAcademy, which has free short courses in Data Science, Introduction to Julia, DataFrames.jl, Machine Learning, and more.
- Data Wrangling with DataFrames Cheat Sheet
- Learn Julia in Y minutes, a great quick-start if you are already comfortable with coding.
- Think Julia, a free e-book (or paid print edition) that introduces programming from the start and teaches you valuable ways of thinking.
- Design Patterns and Best Practices, a book that will help you as you transition from smaller, one-off scripts to designing larger packages and projects.

CONCLUSION

Looking at other great tools like R and Python, it can be difficult to summarize a single reason to motivate a switch to Julia, but hopefully this article piqued an interest to try it for your next project.

In an earlier article, I talked about becoming a **10x Actuary** which meant being proficient in the language of computers so that you could build and implement great things. In a large way, the choice of tools and paradigms shape your focus. Productivity is one aspect, expressiveness is another, speed one more. There are many reasons to think about what tools you use and trying out different ones is probably the best way to find what works best for you.

It is said that you cannot fully conceptualize something unless your language has a word for it. Similar to spoken language, you may find that breaking out of spreadsheet coordinates (and even a dataframe-centric view of the world) reveals different questions to ask and enables innovated ways to solve problems. In this way, you reward your intellect while building more meaningful and relevant models and analysis. ■

Alec Loudenback, FSA, MAAA, is an actuary for the Kuvare family of companies. He can be reached at *alec.loudenback@lbl.com.*

ENDNOTES

1 Python first appeared in 1990. R is an implementation of S, which was created in 1976, though depending on when you want to place the start of an independent R project varies (1993, 1995, and 2000 are alternate dates). The history of these languages is long and substantial changes have occurred since these dates.

2 Julia Computing. (2017). Aviva Solvency II Compliance. Retrieved June 04, 2020, from *https://juliacomputing.com/case-studies/aviva.html*

3 The author of this article contributes to JuliaActuary.

4 H20 AI. (2020). Database-like ops benchmark. Retrieved October 04, 2020, from *https://h2oai.github.io/db-benchmark/*