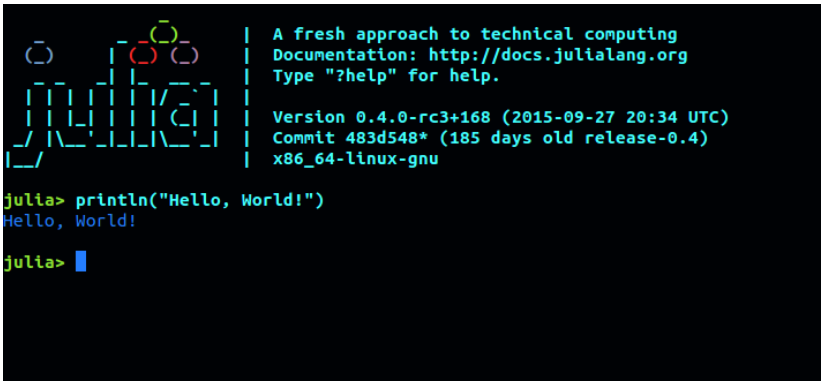
OCEANOGRAPHER'S CHOICE
nekton, plankton, pings, and backscatter

The Julia Language is the Way of the Future

Posted on 2016/03/31 by Sam



Recent surveys have shown that the R Language for statistical computing is undergoing an explosion in popularity among scientists, researchers, and other data-analyzing types. There are good reasons for this: R is Free, easy to use (as much as any programming language can be, anyway), can create beautiful graphics straight out of the box, and has a huge ecosystem of user-contributed libraries covering almost any statistical analysis you could dream of doing. It is becoming the lingua franca in many fields, especially in biology and ecology. For many students and scientists I know, "learning R" and "learning to program" are used almost interchangeably. There are more than 8,000 add-on packages available on CRAN, and some of the best, like those in the Hadleyverse, are changing the way people use the language for the better. We are currently living in the Golden Age of R.

But I don't think R is the way of the future. The future, I believe, belongs to a very new scientific programming language called Julia.

(If you think the future belongs to Matlab, you are probably beyond hope).

Because despite its strengths, R has a number of drawbacks and, er, eccentricities. Most of these drawbacks can be traced to the fact that R was created 20 years ago as an open-source implementation of a previous language, S, which was created twenty years before that. The package ecosystem on CRAN is

huge, but there is a lot of confusion and overlap: it's not uncommon to find three packages that each do 2/3 of what you need. And if you're writing your own code for something not available in a package, R can be very slow. If it can't be vectorized–that is, if you need to write it in a loop–your choices are to leave R and write that bit in C, or hit "Enter" and go get a coffee.

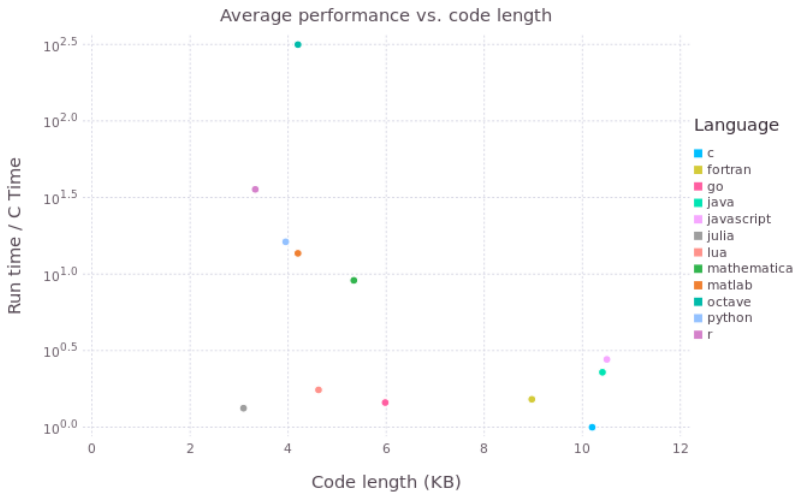
Enter Julia. I first heard of this language when someone posted this ridiculous manifesto by its creators to Reddit a few years ago. The kicker is this outrageous paragraph:

> *We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.*
>
> *(Did we mention it should be as fast as C?)*

They stop short of demanding a pony, but you get the idea. What kind of language could *possibly* come *close* to doing all this?

Well, as it turns out, the one they created.

Julia is a really well-thought-out language. While the syntax looks superficially Matlabby, that is about as far as the similarity goes. Like Matlab, R, and Python, Julia is interactive and dynamically typed, making it easy to get started programming. But Julia differs from those languages in a few major ways. Under the hood, it has a rigorous but infinitely flexible type system, and calls functions based on "multiple dispatch": different code is automatically chosen based on the types of the all arguments supplied to a function. When these features are combined with the built-in just-in-time (JIT) compiler, they let code–even scalar for-loops, which are famous performance killers in R–run as fast as C or Fortran. But the real killer is that you can do this with code as concise and expressive as Python.



This graph shows average run times of several small benchmark programs, plotted vs. the average length of their source files. The times are relative to C. Note the log scale on the y-axis. Ideally, we want concise programs with short run times–i.e., lower is better on both axes.

The graph shows the classic tradeoff in technical computing. Compiled languages like C and Fortran (lower right) are fast to run but slow to write, while "scripting" languages like R and Python (upper left) are fast to write but slow to execute. But wait, what's that grey dot way in the lower left-hand corner?

That's Julia.

Let's look at the Julia script I used to generate this plot to see a few more of the language's killer features. (The .csv data file can be downloaded here.)

```
using Gadfly # grammar-of-graphics plotting
using DataFrames, DataFramesMeta

benchmarks = readtable("julia_benchmarks.csv")
ctimes = @linq benchmarks |>
        where(:Language .== "c") |>
        select(:Benchmark, :Time) |>
        rename(:Time, :CTime)
benchmarks = join(benchmarks, ctimes, on=:Benchmark)

summaries = @by(benchmarks, :Language,
        Time = geomean(:Time ./ :CTime),
        KB = mean(:KB))

p = plot(summaries, x=:KB, y=:Time, color=:Language,
        Geom.point, Scale.y_log10,
        Guide.xlabel("Code length (KB)"), Guide.ylabel("Run time / C Time"),
        Guide.title("Average performance vs. code length"),
        Theme(background_color=colorant"white"))

draw(PNG("julia_benchmarks.png", 18cm, 12cm), p)
```

From top to bottom, kindly notice:

1. Libraries. Julia has a centralized package registry and built-in package manager, and while not anywhere near as big as CRAN, it already covers many of the day-to-day tasks a data analyst might want to do. You can read in tabular data, manipulate it, fit generalized linear models with R-like syntax, and plot them using readily available packages.One really nice feature (at least so far) of the Julia package ecosystem is that it is almost all on GitHub. A side effect of this is that there is a much greater tendency to collaborate and plan. Functionality tends to be split up into logical chunks, and similar efforts are combined.
2. Split-apply-combine data manipulation. The advent of the Hadleyverse has revolutionized the way many people work with data in R (myself included). Once you've gone dplyr, there's no going back. Julia also has the capability, via the DataFramesMeta package, to do these kinds of manipulations. While still in the experimental phase, you can already split, apply, and combine data frames, and pipe the output from one operation to the input of the next.
3. This brings us to another killer feature of Julia: metaprogramming. Julia is "homoiconic," a weird word that means Julia code can be represented as a data structure within the language. As a result, you can write programs that automatically generate other programs. See those @ symbols in the @linq and @by functions? That means they aren't actually functions, but rather "macros": basically, functions that write custom code based on their inputs. R is also homoiconic, but in my experience only the

bravest of hackers ever take advantage of that capability. While still not a beginner's technique, I find metaprogramming in Julia is significantly easier than in R. Many Julia packages take advantage of metaprogramming, and the results, for the user, can be downright magical.

4. Plotting. There is no default plotting library yet in Julia, but there are several very good options. Gadfly, used here, is a ggplot-inspired library. Also available is PyPlot, a wrapper around Python's mature Matplotlib library.

With features like this, there are a lot of intriguing possibilities. For example. Most R people doing Bayesian analysis will run the computationally-intensive part, MCMC, with an external package written in C or C++, such as BUGS, JAGS, or Stan. This works ok, until you write a bug into your BUGS or a jank into your JAGS. A cryptic error is then thrown from the Great Compiled Beyond, and good luck figuring it out. But what if you were working in Julia? The MCMC code could be written in Julia itself. It could run about as fast as C, but chasing down bugs wouldn't mean switching languages. You could write your probability model using any standard Julia functions. Julia's code introspection would let you use automatic differentiation to implement Hamiltonian Monte Carlo algorithms–the secret sauce of the Stan software. And in fact, this is an area of active development: see Mamba and Lora. (While neither of these is yet as polished or simple to use as JAGS or Stan, there are Julia wrappers for JAGS and Stan for the meantime.)

I wouldn't recommend everyone drop R and use Julia...yet. The language is less than five years old, and still very much under development. Some features are likely to change. Many packages are still in experimental mode, and not everything that is available in R is there for Julia (though a surprising amount is). Documentation is improving, but is not always complete or thorough. There is not yet an IDE comparable to RStudio. If you're happy and productive in R (or whatever you're using), by all means carry on.

But even if you are, it's worth downloading Julia and playing around with it for a couple of hours. I did this back in 2012, and have since fallen in deep: I now spend roughly a third of my coding time in Julia, and have written a couple of full-featured packages. Learning a second language, whether human or computer, is good for you and can help you see old problems in a new light. So why not make your next language Julia? It might even be the last one you need to learn...

***

Some further resources for the intrigued:

The Julia Manual. Work your way through the examples here, and you'll already have a pretty good grasp of how things work.

Julia for R programmers (PDF): a presentation introducing Julia to the R-using audience.

Differences from other languages: for Matlab, R, Python, and C.

Learning Julia: a large collection of links and videos on the JuliaLang website.

This entry was posted in Quantitative, Uncategorized and tagged Julia, programming, Python, R. Bookmark the permalink.

2 Responses to *The Julia Language is the Way of the Future*

Pingback: *Deconstructing the "Near" Perfect Deal – Our Investment in Julia Computing - Amicki's Tech Store*

Pingback: *7 Things You May Not Know About Julia - Skillenza Blog*