March 01, 2016

**Julia: A Fast Language for Numerical Computing**

By Alan Edelman

The Environmental Protection Agency’s discovery that Volkswagen installed emissions-cheating software in over 11 million vehicles last September garnered much attention. According to reports, the software made it seem like the vehicles were releasing significantly less nitrogen oxides than in reality, allowing them to discharge up to 40 times the allowable amount of smog-producing pollution. This prompted *The New York Times* to call for an openness in software that goes beyond the scientific need for transparency. Software openness is becoming a matter of safety and accountability for compliance purposes. This need applies to more than just cars, as the Internet of Things requires communication of data and instructions among multitudes of devices. Open source is not just a checkbox; easy-to-read common code is increasingly becoming paramount.

With its speed, openness, and transparency, Julia is emerging as that “must-have” easy-to-read common language. And in terms of safety and accountability, Julia is being used as the specification language for the next-generation airplane collision avoiding system (see Figure 1).

Why use Julia over languages such as R or Python? The differences lie in the programs’ intricacies. These are all very good languages, but when programming in R or Python the two-language problem lurks: the need to prototype with one slow dynamic language and rewrite with a fast static language to obtain the final product. Having been around longer than Julia, R and Python have naturally established the kind of popularity that takes some years to acquire. Nevertheless, the two-language problem is very serious. Prototyping in R or Python necessitates a rewrite for speed and deployment. If emissions software is written in Python, one must quickly drill down to C code to understand the details of what is happening. Julia’s advantage is that it solves the two-language problem.

While exact usage statistics are difficult to generate, Julia’s popularity is doubling every nine months, Moore’s law style. There are currently an estimated 100,000 Julia users, nearly 1,000 Julia packages, and more than 400 GitHub contributors. Dozens of universities are also using Julia in classes in engineering, big data, optimization, and numerical analysis, as well as in undergraduate computer science, linear algebra, and statistics. Anecdotally, many engineering and financial companies are using Julia at the “grassroots” level.
Nick Trefethen, former SIAM President and my Ph.D. advisor, first got me thinking about how to best articulate Julia’s speed and advantages. I spoke about Julia at the “New Directions in Numerical Computation” linear algebra conference in honor of Trefethen’s 60th birthday this past August. The following commentary includes a few key points from that talk.

**Placing Emphasis on Performance**

Julia demonstrates that if designed correctly, the very features that help the computer can also appeal to the human. Those unfamiliar with the transformations of code tend to replace complete understandings with magic accelerators based on some partial truths. Users hear that compiling makes code run faster. Jitting makes code run faster. Vectorizing makes code run faster. Declaring types makes code run faster. Interestingly, what makes an algorithm run faster seems less mysterious than what makes code run fast. Algorithms run quickly when they do not waste time computing unnecessary things. This efficiency can mean taking advantage of structure (as in not multiplying by or adding zeros in sparse matrix computations), or computing only as many digits as are necessary (in iterative methods, or by truncating to take advantage of hardware).

Ultimately, code runs faster when it is not beleaguered by unnecessary computational baggage. That the computer language is thought of separately from the computer algorithm is unfortunate. When considering numerical error, every computation must work just right. If not, the entire computation goes wrong. The same can be said of performance; a language must be designed just right for speed.

The earliest computer languages were designed primarily to meet the needs of the computer, and were close enough to the hardware to be relatively fast. Then came interpreted languages, which were designed primarily to meet the needs of the human. This facilitated productivity and was thus the better choice for most people. Julia finds a new place by striking a reasonable balance between human and computer.

People initially thought, almost by some kind of conservation law of nature, that a dynamic mathematical language was required to be slow. Julia shows that a carefully designed language that is comfortable for the programmer can still run quickly. It is important to understand that to date, many interpreted computer languages were designed with use cases in mind, while performance and acceleration came grafted in as potential “add-ons.” Placing more emphasis on performance enhances the cooperation between human and computer.

**A First Julia Experience: the Batman Curve**

The famous Batman curve is pictured in Figure 2, which users can experiment with on juliabox.org. The expressions in Figure 2 are equivalent to others that may be found online; I believe this version is much easier for people to read (See interactive code in Lecture 1).

**Julia is Fast and Flexible**

Computer programs run faster when they transform to a tight set consisting of a small number of quick execution steps. Consider some uses of the plus character “+.” We ask so very much from plus, and know...
Julia users rarely look at assembly, but they readily can. Julia’s tight assembly is a clue as to why it is so fast.

Figure 3. Julia users rarely look at assembly, but they readily can. Julia’s tight assembly is a clue as to why it is so fast.

Of course, “+” can refer to so much more, including dense matrix addition, sparse matrix addition, structured matrix addition, and any type of combination. The point is that Julia produces tight code in all instances.

How does Julia do it? For those desiring a quick answer, “multiple dispatch” is the explanation. The Julia language is built on careful principles of type stability that allow for type inference, which enables Julia to perform proper code selection through the multiple dispatch mechanism. Julia reduces the uncertainty in a computation that can waste time.

One can see multiple dispatch in action with the “@which” command in Figure 4, which allows users to drill into Julia code if they wish to see how it implements various “+” symbols. Without the user declaring types, the Julia program knows which “+” is intended and uses the correct assembly code without wasting time.

Julia is very much Julia all the way down. (There are bits of other software, and LLVM at the very bottom.) While Julia uses and plays nicely with packages from Python and other computing languages, it does not need to for speed. Even packages such as LAPACK would likely run as fast and could be more flexible if rewritten in Julia. Other numerical languages are front ends to C, C++, or Fortran.

Figure 4. Multiple dispatch in action with the “@which” command.

Tropical Numbers
Tropical algebra offers a “x” that distributes over a “+,” but the two symbols can function as arbitrary operations otherwise, as Françoise Tisseur explained at the Householder Symposium XIX[1].

Here is the famous MAX+PLUS example in Julia:

```julia
# Define your favorite plus, and times
⊕(x,y) = max(x,y) # Put your favoring semiring plus here
⊗(x,y) = x+y # Put your favorite semiring times here
# Best if ⊕ distributes over ⊗

import Base
immutable TNum{T} <: Number
  val::T
end

# immutable means if you execute this twice, it will warn you it's not cool
TNum(1)
TNum{Int64}(1)
TNum(2.0)
TNum{Float64}(2.0)

TNum(1/2)
inv([1/(i+j) for i=1:3, j=1:3])

3x3 Array{Rational{Int64},2}:
  72/1 -240/1  180/1
-240/1  900/1 -720/1
 180/1  -720/1  600/1

import Base
Base.zero{T}{{::Type{TNum{T}}}} = TNum(zero(T))
Base.one{T}{{::Type{TNum{T}}}} = TNum(one(T))
+(a::TNum, b::TNum) = TNum(a.val ⊕ b.val)
*(a::TNum, b::TNum) = TNum(a.val ⊗ b.val)
Base.show(io::IO, a::TNum) = print(io, a.val)

x=TNum(5)
y=TNum(6)
@show x + y
@show x^10
x + y => 5
x^10 => 50
```

One can then define tropical numbers of all types:

A few more basics allow for general work with tropical numbers:

Notice that now powers work, matrix multiply works, kronecker products work, etc. This demonstrates Julia's speed. Julia's abstractions allow the type information to flow directly through unhindered. For example, a user might think there is quite a lot going on upon defining a fancy tropical number type, but the assembly shows that the code is remarkably very tight:
Indeed, users can see the "compare" command in the plus and the "add" command in the times.

**Open Source Research**

In reality, Julia is fast because the community cares about performance. Because it is an open project, self-described "performance obsessive types" have flocked to the project, thus boosting Julia's performance. I personally want to see Julia emerge as the language of high performance computing, blurring the distinction between "Silicon Valley"-style big data and artificial intelligence (AI) computations, Wall Street style number crunching, and National Laboratory style scientific computing.

Numerical software research enjoys the benefit of its fruits being quite practical. Those with decades of experience in the business know that users have been sharing code development long before GitHub or the use of the term "open source."

It has been a pleasure to nurture the Massachusetts Institute of Technology's component of this worldwide effort and to experience the brilliant contributions from so many first-class researchers. Experience Julia for yourself by searching "Juliacon2015" online and watching the videos, trying [juliabox.org](https://juliabox.org), or visiting [julialang.org](http://julialang.org).

**Acknowledgments:** I'd like to dedicate this article to my Ph.D. adviser and former SIAM President, Nick Trefethen, on the occasion of his 60th birthday.

---

*Alan Edelman is a professor of applied mathematics at Massachusetts Institute of Technology. He has been working in the area of high performance computing systems, networks, software, and algorithms for 30 years and has won many prizes for his work including the prestigious Gordan Bell Prize, a Householder Prize, and a Babbage Prize. With Julia he sees a fresh approach to high performance technical computing.*