John Myles White

"Who refuses to do arithmetic is doomed to talk nonsense."

Browse: *Home* / *2012* / *March* / *31* / *Julia, I Love You*

Julia, I Love You

By John Myles White on 3.31.2012

Julia is a new language for scientific computing that is winning praise from a slew of very smart people, including Harlan Harris, Chris Fonnesbeck, Douglas Bates, Vince Buffalo and Shane Conway. As a language, it has lofty design goals, which, if attained, will make it noticeably superior to Matlab, R and Python for scientific programming. In the core development team's own words:

> *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?)*

Remarkably, Julia seems to be on its way to meeting those goals. Last night, I decided to see for myself whether Julia would live up to the hype. So I taught myself just enough of the language to write an implementation of the slowest R code I've ever written: the Metropolis algorithm-style sampler Drew and I use in Chapter 7 of Machine Learning for Hackers to show off randomized, iterative optimization algorithms. You can find both the original R code and my new Julia code on GitHub in two files name `cipher.R` and `cipher.jl`, respectively.

In my opinion, the new code in Julia is easier to read than the R code because Julia has fewer syntactic quirks than R. More importantly, the Julia code runs much faster than the R code without any real effort put into speed optimization. For the sample text I tried to decipher, the Julia code completes 50,000 iterations of the sampler in 51 seconds, while the R code completes the same 50,000 iterations in 67 minutes — making the R code more than 75 slower than the Julia code.

Having seen that example alone, I would be convinced Julia is a real contender for the future of scientific computing. But this iterative sampling algorithm is not close to being the harshest comparison between Julia and R on my machine. For a more powerful example (lifted straight from the Julia docs), we can compare Julia and R code for computing the 25th Fibonacci number recursively.

First, the Julia code:

```julia
fib(n) = n < 2 ? n : fib(n - 1) + fib(n - 2)
@elapsed fib(25)
```

Second, the R code:

```r
fib <- function(n)
{
  ifelse(n < 2, n, fib(n - 1) + fib(n - 2))
}

start <- Sys.time()
fib(25)
```

```
end <- Sys.time()
end - start
```

The Julia code takes around 8 milliseconds to complete, whereas the R code takes around 4000 milliseconds. In this case, R is 500 times slower than Julia. To me, that's sufficient reason to want to start focusing my time on implementing the algorithms I care about in Julia. I hope others will consider doing the same.

Posted in Statistics | 50 Responses

50 responses to "Julia, I Love You"

David J. Harris 3.31.2012 at 6:50 pm | *Permalink*

This doesn't invalidate your point at all, but I was able to shave almost a factor of ten off your R Code's time by changing ifelse() to if() and else(). The problem with ifelse is that it calculates outcomes for both TRUE and FALSE, which adds up to a lot if you call fib() recursively.

So Julia might only be 50 times faster :-p

"`R
fib2 <- function(n)
{
if(n < 2){n}else{fib2(n – 1) + fib2(n – 2)}
}"`

David J. Harris 3.31.2012 at 6:54 pm | *Permalink*

By the way, I just checked, and Julia has updated their R script to use if() and else() as well:
https://github.com/JuliaLang/julia/blob/master/test/perf/perf.R

John Myles White 3.31.2012 at 7:01 pm | *Permalink*

That's a really good point, David. I was very fast and loose with my implementations of everything. And I didn't realize using if() and else() would make such a difference in R. On the other hand, I wonder how much the Julia version can be improved by using typing.

On another note, the comparison between iterative solutions for the Fibonacci number seems to suggest quite different things about Julia and R's relative performance. I'm cleaning up my implementations right now, but Julia doesn't seem to beat R by much when you're not solving the problem recursively. (Of course, being a language that's bad at recursion isn't something R can be proud of.)

Stefan Karpinski 3.31.2012 at 7:07 pm | *Permalink*

Thanks for the lovely writeup, John. I'm very pleased that you're enjoying Julia — and its performance :-)

@David: The "official" benchmark R code does use the if control-flow construct rather than the ifelse function. However, if I'm not mistaken, R has lazy evaluation — yes, this is a little-known fact about R — which means that the if and else expressions passed to the ifelse function are actually *not* both evaluated. For some reason that I don't at all comprehend, however, ifelse does seem to evaluate either the if or the else expression *twice* rather than only once. Check it out:

```
> ifelse(TRUE, print("hello"), print("world"))
[1] "hello"
[1] "hello"
> ifelse(FALSE, print("hello"), print("world"))
[1] "world"
[1] "world"
```

I have no idea why ifelse would do this. The upshot of this double evaluation behavior is that the fib benchmark in R may actually only be 250x slower than Julia on John's system.

Stefan Karpinski 3.31.2012 at 7:18 pm | *Permalink*

Adding type annotations wouldn't actually help performance at all. The way Julia works is that for every specific type signature a method is called with, a version specifically for those types of arguments is just-in-time compiled. So when you call fib(25), what's executed is compiled knowing that the argument is an Int64, and using type inference to figure out exactly what method to call recursively — if n is an Int64 then so are n-1 and n-2. (That's deduced by type inference from the definition of the minus operation, which is probably also inlined, not some pre-defined magic knowledge.)

Because of this way of working, you really can just write typeless Julia code with no performance penalties. The main reason for writing type annotations on function arguments is so that you can use multiple dispatch to specialize generic behaviors for specific argument types. Other than that, there are pretty limited reasons to need to put type annotations on anything.

The Julia fib benchmark is still about 2x slower than C/C++. That's largely because Julia function calls are still a bit slower than C function calls. However, I believe Jeff has some optimizations up his sleeve that may bring that overhead down once he gets a chance to implement them. We're still aiming to close the gap with C all the way.

David J. Harris 3.31.2012 at 7:38 pm | *Permalink*

@Stefan: I played around with ifelse() a bit, and it looks like we were both wrong. The first "hello" you got was from print(), but the second one was actually just a character vector (i.e. the output of ifelse). Check this out:

```
> ifelse(
+ TRUE,
+ {print("hello"); TRUE}, # print "hello" and then return TRUE
+ {print("world"); FALSE}
+ )
[1] "hello"
[1] TRUE
```

ifelse() *sometimes* does more evaluation than it needs to, roughly like I said, but only when there's a mix of TRUEs and FALSEs in the test vector.

I haven't profiled the code, but I suspect that most of the overhead from ifelse in this case is actually coming from vector manipulation.

Stefan Karpinski 3.31.2012 at 7:43 pm | *Permalink*

Ah, that makes sense. I has a suspicion that might have been what was going on, but didn't dig any deeper. In any case, lazy evaluation in a very impure, imperative programming language is completely weird.

Douglas Bates 3.31.2012 at 8:16 pm | *Permalink*

Actually you don't even need the else in the R function. It could be written as

```
fib3 <- function(n) {
if (n < 2) return(n)
```

Douglas Bates 3.31.2012 at 8:19 pm | *Permalink*

I hit the wrong key and submitted an incomplete function. The function in my posting should have been

```
fib3 <- function(n) {
if (n < 2) return(n)
fib(n – 1) + fib(n – 2)
}
```

Clint 4.1.2012 at 12:24 am | *Permalink*

I'm running the new R code against the Julia code for fib(250) and seeing that R is running 7 threads and using 125mb of memory where julia is using 3 threads and 68mb of memory on an old (2007 core duo) iMac.

I'll hold off on doing fib(250) in Perl; otherwise I wouldn't get an answer for a week.

Paolo 4.1.2012 at 3:43 am | *Permalink*

Great discussion! I'll take a close look at julia development! As usual for simple (and non so simple) tasks the strength of R, its selling point, is the vast collection of packages optimized for all kind of task, e.g. for fibonacci numbers (on my VERY slow macbook 2007):

```
library(gmp)
start <- Sys.time()
fibnum(250)
Big Integer ('bigz') :
[1] 7896325826131730509282738943634332893686268675876375
end <- Sys.time()
end – start
Time difference of 0.00165987 secs
```

Furthermore, thanks to Rcpp & co. collection of packages it is relatively easy to write short c++ code to implement your algorithm in a very effective and optimized way. Just my 2 cents, all the ideas depicted in the post are really inspiring!

Nathaniel Daw 4.1.2012 at 8:37 am | *Permalink*

I had a similar experience where I just took a piece of slow matlab code (which I had previously used to determine that python/numpy were even 3x slower) and found it was 20x faster in julia.

Marc 4.1.2012 at 12:20 pm | *Permalink*

I am as enthousiastic as others, but I'm wondering how we are going to handle the transition, or the co-existence, of for instance R and Julia. As pointed out above, vast stock of routines available for R remains of utmost value.
How do you see the future ? Is there going to be a CRAN-like repository for Julia ? The relatively clean package contribution system for R is valuable – for Matlab it's complete mess, I'll never miss that, and their licensing system has become unbearable anyway. I feel that the research communities may be a bit different/broader for Julia : people from database/data management might be happy to use Julia, because it runs faster, because it seems designed with parallelism in mind, and because database people are more and more mixing with machine learning people.

Marc 4.1.2012 at 12:24 pm | *Permalink*

Wasn't there any possibility/point of being backward-compatible with R ?

G. Grothendieck 4.1.2012 at 2:09 pm | *Permalink*

Its well known that R is no good at recursion. Use a loop as shown below. On my machine fib2a(20) below runs 500x faster than the R code in fib2 and if we byte compile fib2a it runs 2000x fater than the R code in fib2 so its not R that is slow but its the use of a style of programming that does not match R's capabilities:

```
fib2a <- function(n) {
if (n < 3) 1
else {
y <- rep(1, n)
for(i in 3:n) y[i] <- y[i-1] + y[i-2]
y[n]
}
}

library(compiler)
fib2c <- compiler(fib2a)

library(rbenchmark)
benchmark(fib(20), fib2a(20), fib2c(20))
```

The result of the last line is (after trimming away some junk):

```
> benchmark(fib2(20), fib2a(20), fib2c(20), replications = 500)
test replications elapsed relative user.self sys.self
1 fib2(20) 500 58.68 1956.000000 58.39 0.09
2 fib2a(20) 500 0.11 3.666667 0.11 0.00
3 fib2c(20) 500 0.03 1.000000 0.03 0.00
```

Tim Salimans 4.1.2012 at 3:58 pm | *Permalink*

Great stuff! I myself just did a little comparison between Matlab and Julia for a simple Gibbs sampler (see my blog), also with very impressive results.

T. U. 4.1.2012 at 5:54 pm | *Permalink*

Hi there!

I re-implemented the metropolis-sampler in Python, and achieved around 560fold speedup from the R code you posted (which would put it at 10x speedup compared to the Julia-speedup you reported). You can find the code here: http://pastebin.com/AcWahEby

While I think Julia's goals are very awesome, it's not quite there yet. And with Python/SciPy and Matlab we already have two great options that could dethrone Julia. But I guess only time will tell… :)

David 4.1.2012 at 8:30 pm | *Permalink*

Great post but as mentioned recursive calls in R are very slow. This has been addressed on stackoverflow: http://stackoverflow.com/questions/6807068/why-is-my-recursive-function-so-slow-in-r

Here is my little comparison. Code can be found here: http://pastebin.com/3vnLHfcE

```
N <- 25
res <- benchmark(fibRcpp(N),
fib2(N),
fib2a(N),
fib2c(N),
columns=c("test", "replications", "elapsed",
"relative"),
order="relative",
replications=20)
print(res)
test replications elapsed relative
4 fib2c(N) 20 0.001 1
3 fib2a(N) 20 0.002 2
1 fibRcpp(N) 20 0.025 25
2 fib2(N) 20 9.307 9307
```

Abhijit 4.1.2012 at 11:02 pm | *Permalink*

Hi John,

Are there pipes being developed between R and Julia, so that we can leverage both the large library of R and the efficiencies of Julia in projects?

Vaidotas Zemlys 4.2.2012 at 7:05 am | *Permalink*

Well, how often you need to calculate Fibonacci numbers in your daily statistical work? Me personally, never. Call me spoiled, but I value code clarity over speed. The linked post of Douglas Bates for example suggests Julia favors devectorising, which for me indeed is heretical thought. If you start devectorising code what is Julia's benefit over C? I thought the point of languages such as R or Matlab is to save people time by not writing book-keeping code.

Dave 4.2.2012 at 9:08 am | *Permalink*

I think what's being done with Julia is commendable.

What I can't tell is – who are you pitching this to? To me, it feels like only full time coders will benefit for the immediate future.

To explain what I mean… I've been trying to get my head around R for about two years or so (not a full-tiume coder, so I dabble in my day job).

R definitely has hangups but the benefit is that there are a lot of people writing functions that reduce the need to code many tasks from sctrach. It's this off the shelf aspect is helpful to a user like me.

If I decide it's too slow – my question is: why learn Julia? Why not learn C?

While Julia can probably do all the stuff that R's currently doing in ready-made-packages, as a user deciding on switching, its hard to buy into it, until there's a body of code freely available or unless I'm already a code demon.

So, for me while it's an exciting concept, it's a case of waiting until braver souls scale the peaks and lay the path.

Bill Harris 4.2.2012 at 11:28 am | *Permalink*

From http://www.jsoftware.com/jwiki/Essays/Fibonacci%20Index:

```
fib=: 3 : 0 ” 0
mp=. +/ .*
{.{: mp/ mp~^:(I.|.#:y) 2 2$0 1 1 1x
)

6!:2 ‘fib 25’
0.000117899
6!:2 ‘fib 250’
0.000181834
```

i.e., fib(25) takes 0.12 msec, and fib(250) takes 0.18 msec.

That's on a machine that gave 3 seconds for the original R version:

```
> source(“juliabenchmark.R”)
Time difference of 3.108877 secs
```

J provides a connection to R, for when you need R's libraries (http://www.jsoftware.com/jwiki/Interfaces/R?action=show&redirect=Addons%2Fstats%2Fr).

What am I missing? J seems about 80 times faster than Julia in this example.

Michael Lachmann 4.2.2012 at 12:28 pm | *Permalink*

Cheating, I know….

```
fib=(function() {fibV=c(1,1);fib=function(n) {if( length(fibV)<n ) {fibV[n]<<-fib(n-1)+fib(n-2)};fibV[n]}; fib})()

start <- Sys.time(); print(fib(250)); end <- Sys.time(); end-start
[1] 7.896326e+51
Time difference of 0.004354954 secs
```

NJank 4.2.2012 at 1:26 pm | *Permalink*

Coming from a completely non-programmer background, I use Octave (GNU equivalent to Matlab) for most of my scientific problem solving. I've seen scientists and engineers completely balk at coding something in C/C++ (even though they took that one programming class in college), but dive right into Matlab. The question is whether the language/environment you're using is appropriate for the level of effort being done. Most Octave/Matlab users probably couldn't care less about shaving time off a

particular algorithm. They just need to implement the algorithm and be sure the code is correct.
Now, there obviously are exceptions, but again, for most non-programmer scientists, it comes down to: how easily can I put my algorithm into code. So, if you want to gain acceptance in the 'not worrying about how my ifelse works on the inside' community, what's the barrier to entry for simple algorithms? I don't quite want to get as pedantic as "how many lines to implement hello world". But, more importantly, how intuitive are those lines. In matlab/octave, there's no header or include declarations. no class definitions. I've introduced matlab as a matrix math calculator, then you show a script to repeat a set of calculations, then you show looping constructs, and whether or not you introduce functions and file i/o, you've covered 90% of what most undergrad engineers want to know to get the job done.

I've been tempted but never got around to more than peeking at Python, R, or I guess not Julia. What would they give me that Octave doesn't?

G. Grothendieck 4.2.2012 at 4:18 pm | *Permalink*

@Bill, If we use Algorithm 2B from:
http://cubbi.org/serious/fibonacci/j.html
which seems more comparable to the R and Julia code (than using the Fibonacci matrix algorithm which is a completely different approach) then we get:

```
fibj=: {:@($:&1x 0) : ((<:@[ $: +/\@|.@])^:(*@[))
6!:2 'fibj 250'
0.00157967
```

which is slower than the 0.00043 seconds on my machine for fib2c(250) in R (see my earlier post for fib2c).

Sergei Steshenko 4.2.2012 at 4:51 pm | *Permalink*

To NJank, regarding "What would they give me that Octave doesn't?" From my brief reading on Julia I see it's a much more functional (I mean programming paradigm) language – with nested functions, for example.

Also, macros are nice.

And _speed_ – you will sooner or later need speed.

I am using GNU Octave myself, but I am always looking around.

Alex K 4.3.2012 at 3:14 am | *Permalink*

I do not download anything that requires git because OS X does not support it and I do not want to add another layer of error to my machine.

Has anyone compared Julia to Java??

WT 2.11.2013 at 11:55 pm | *Permalink*

OS X 10.8 ships with git:

git version 1.7.9.6 (Apple Git-31.1)

Sergei Steshenko 4.3.2012 at 4:36 am | *Permalink*

To Alex K. GIT is trivially built from sources, and running it does not require root privileges.

Regarding Java – it's a poor language because of many missing features of functional paradigm. Though, IIRC, latest Java version finally has clsoures.

Alex K 4.3.2012 at 6:07 am | *Permalink*

I have used git. No one in my team liked it though we ended up able to use it.

This URL just about says all I want to say.

http://steveko.wordpress.com/2012/02/24/10-things-i-hate-about-git/

I came in one day to find someone in the US team had reset the remote repository to the state it was in three weeks before.

It feels like git is liked by people who love fiddling with system settings rather than doing something productive like coding. THe learning curve is too steep for those with work to do and who have a life
.

jayveesea 4.3.2012 at 7:03 am | *Permalink*

great write up! Julia looks very promising.

i am surprised i never see much talk about gsl-shell (http://www.nongnu.org/gsl-shell/index.html), its been around for a few years, and now uses luajit2, which is very fast. anyways, i ran a quick test on my pc…

```
–from http://en.literateprograms.org/Fibonacci_numbers_%28Lua%29
function fib(n) return n<2 and n or fib(n-1)+fib(n-2) end
local t=os.clock()
N=fib(25)
print(os.clock()-t)
```

elapsed time was 0.000999.. in gsl-shell. and your implementation in julia the elapsed time was 0.006000.

the nice thing about gsl-shell (besides using luajit2) is it has many of the gsl libraries built in which give alot more functionality then standard lua. i would love to see gsl-shell included in the some of these julia benchmarks.

Vaidotas Zemlys 4.3.2012 at 7:07 am | *Permalink*

Well I moved all my colleagues, which are not programmers onto git and github. It is been two years, already, and more than 10 statistical modelling projects with R. Everybody was (is) productive and was (is) happy. So do not generalise based on your own experience. For me git is a god given tool, which keeps some sanity in this crazy world :)

Sergei Steshenko 4.3.2012 at 2:16 pm | *Permalink*

To Alex K. AFAIR, GIT is used in Linux kernel development. So, I think, it is used for something productive – even though i don't use it myself.

To me GIT looks much more powerful than, say, Subversion (SVN).

Phil Hartfield 4.3.2012 at 3:18 pm | *Permalink*

It is difficult to say exactly how much slower Perl is, even using Math::BigInt::BMP. I get fib.pl 250 running faster than fib.pl 25 four times out of five. Fib.pl 10000 runs in under 200ms. Fib.pl 1000000 took less than 30 seconds. Time for perl to start and run hits 30ms to 40ms every time for fib.pl 10…

```
[root@titan ~]# time ./fib.pl 250; time ./fib.pl 10000
7896325826131730509282738943634332893686268675876375
real 0m0.034s
user 0m0.028s
sys 0m0.006s
336447648764317832666216120051075433103021484606800639065647699746800814421666623681555955136337340255820653326
real 0m0.186s
user 0m0.181s
sys 0m0.003s

[root@titan ~]# time ./fib.pl 1000000 | wc
0 1 208988

real 0m27.359s
user 0m27.295s
sys 0m0.047s
```

Rob 4.3.2012 at 3:32 pm | *Permalink*

Seems everyone expects R to be able to do everything well. Truth is, it's first and foremost a statistical programming language, and probably the best one there is. There are some features, like if() and else() statements, but that isn't what R is about. There are simply awesome packages which people have built on this cool framework which many of us find invaluable. 50 times slower? I would call that a trivial constant on the running time which I would accept gladly for all the stuff I don't have to code from scratch because it's R. If running time really is a problem for you, learn to vectorize your work in R and you'll find your programs run just as quick as cpp. Julia sounds cool, lets hope R & J can go on a date together somehow…

Michael Lachmann 4.4.2012 at 4:51 am | *Permalink*

Yes, I agree with Rob. We are comparing apples and oranges. R is mainly good for statistical analysis, aided mainly by its easy operation on vectors and huge library. (and it is free, both ways)

In my opinion — though I haven't worked much with matlab –in scientific computing matlab is superior to R because of speed and uniformity (i.e. language is more consistent).

Julia seems very nice. It seems to be aimed as a "competitor" to matlab. Though, it could be that with enough library support it could be used as a replacement to R. Depends on the libraries. gsl and octave see to be in the same bin.
Julia has an advantage with its built-in parallelization.

In general one shouldn't use C/C++ to analyze data. Even a simple task like reading a csv is complicated. (written by someone who in the other window is debugging C++ code to read csv....)

But, for me the documentation of Julia is too weak to be able to test Julia. I can't even figure out what the different libraries are called, and what they contain. What is the difference between LineIterator(stream) and each_line(stream or command) and how are they used? Did anyone find better documentation than http://julialang.org/manual/ ?

BobC 4.18.2012 at 11:40 pm | *Permalink*

Another speedy competitor in the wings is the PyPy rewrite of Python, which also approaches C/C++ speed. What PyPy does for pure Python code is nothing short of amazing.

For DVCS fans who are GIT-haters, consider giving Fossil SCM a try. Think of it as a distributed Trac. Or a local GitHub in a single executable and a single database file.

roger 6.8.2012 at 1:13 pm | *Permalink*

Julia has no named arguments? yikes.

Sergei Steshenko 6.8.2012 at 1:52 pm | *Permalink*

If Julia has associative arrays, absence of named arguments is a no issue in reality.

Stefan Karpinski 6.8.2012 at 3:30 pm | *Permalink*

Julia does have associative arrays (they're just a data type defined in the language, but they're as fast or faster than, e.g., Python's dicts), but that doesn't entirely make up for the convenience of keyword arguments. Keyword arguments are a planned feature: https://github.com/JuliaLang/julia/issues/485#issuecomment-4218541. There are some significant issues arising from combining keyword arguments and multiple dispatch, and there are also performance considerations. We want to make sure we get this feature right instead of rushing something half-baked out the door. The syntax f(name=val) is already reserved for keyword arguments.

gd047 6.11.2012 at 1:38 pm | *Permalink*

Recursion is a very inefficient way. Try this much faster way to get the fibonacci sequence

```
l1 <- (1+sqrt(5))/2 ; l2 <- (1-sqrt(5))/2
P <- matrix(c(0,1,1,0),nrow=2)
S <- matrix(c(l1,1,l2,1),nrow=2)
L <- matrix(c(l1,0,0,l2),nrow=2)
C <- c(-1/(l2-l1),1/(l2-l1))

start <- Sys.time()
n<-25
c(sapply(seq(1,n,2),function(k) P %*% S %*% L^k %*% C))[n]
end <- Sys.time()
end – start
```

zhanxw 2.16.2013 at 9:49 pm | *Permalink*

I think you missed the point…
Here people want to benchmark the same program, not to write the fastest code.

Olivier Lenoble 8.18.2012 at 8:32 am | *Permalink*

I'm always wary when people want to benchmark programming languages by testing an algorithm coded in exactly the same way across languages. Obviously each of them has its strong points and you end up not testing the speed of a language but rather you're simply asking the question "How fast is X at doing this calculation in that specific way"…. It's a bit of a lazy way and it's not the way professional programmer would approach a problem (or at least, it is not the way they should).
Something I noticed in particular is that people have a tendency to use plain procedural programming, which means that languages as C will always come on top of the benchmark.

About fibonacci, as somebody pointed out, you should use some linear algebra instead:

```
# F0 and F1 are the first two terms of the fibonacci sequence
fibon <- function(N, F0 = 1, F1 = 1){
vec0 <- c(F0,F1)
if ((N== 0) | (N==1)){
return(vec0[N+1])
}

A <- array(c(0,1,1,1), dim = c(2,2))
A0 <- array(c(0,1,1,1), dim = c(2,2))

for (i in 3:N){
A <- A%*%A0
}
return(sum(A[2,]*vec0))
}
```

Now testing it (I call the function fibon(25) 1000 times to get some relevant average value.

```
starttime <- Sys.time()
fib25 <- mapply(fibon, rep(25, 1000))
endtime <- Sys.time()
endtime – starttime
```

On my notebook (!!), this gives me 0.3s for 1000 iterations….

Kijysae 9.7.2012 at 3:19 am | *Permalink*

I do not know Julia. But looking at your file cipher.jl on
github, I see this:

————————————

```
function sample(a, n)
o = []

for i = 1:n
index = ceil(length(a) * rand())
o = [o, a[index]]
end

o
end
```

———————————

So in the above, the array ‘o’ is being resized each time in the loop? In Matlab, it is considered bad practice to do this and must first allocate the array size before.

How does Julia handle this? Would it not be better also in Julia to preallocate the size of ‘o’ in the above to get even better performace?

Stefan Karpinski 9.7.2012 at 10:42 am | *Permalink*

Hi Kijysae. It is true that pre-allocating arrays like this is *much* better practice (the difference is often shockingly large). Appending will work and it’s not fully O(n^2) because the array implementation uses an exponential allocation strategy to mitigate the performance in such cases, but still, pre-allocation is always better.

dude 2.12.2013 at 12:03 am | *Permalink*

Well I’ll be switching for the simple reason that “R” is a stupid fucking name which makes it hard to search for. At least “C” had the excuse that it was around before search engines existed. “Julia” is not much better (as the title for this article shows), but I’ll take what I can get.

Doug 2.13.2013 at 2:01 pm | *Permalink*

Hi John,

Just a suggestion that if you ever present Julia to an engineer what uses Matlab, don’t use a recursive formula as the example. I understand that this is a nice example in terms of showing off Julia’s conciseness and syntactic sugar, but to me (an engineer) recursive functions fall somewhere between irrelevant fansy pants and heretical. Also, > ? : is not exactly readable to me, but maybe I’m just out of touch.

I’d suggest:

```
tic
A = rand([1000,1000,1000]);
B = A.^2;
toc
Elapsed time is 16.870236 seconds.
```

Above is single threaded on a 3Ghz Xeon. Alternatively linear algebra, fminsearch or ffts. These are a Matlab user’s real world.

Sergei Steshenko 2.13.2013 at 3:55 pm | *Permalink*

Regarding “an engineer what uses Matlab, don’t use a recursive formula as the example. I understand that this is a nice example in terms of showing off Julia’s conciseness and syntactic sugar, but to me (an engineer) recursive functions fall somewhere between irrelevant fansy pants and heretical” -well, I am an engineer who uses GNU Octave (a Matlab clone, but it doesn’t matter), but I am also a programmer.

And, for example, I once wrote a partial Verilog parser with capability to parse and evaluate arbitrary precision expressions. And because of this recursion is my friend – in parsers – not in HW design.

So a recursive and a non-recursive example are always nice to have.

Doug 2.13.2013 at 4:11 pm | *Permalink*

To be fair, the Julia docs are much more focussed on 'core' computational stuff and it looks impressive. I'm familiar with Octave, I just couldn't wean myself off Matlab performance so Julia looks really attractive if I return from the dark side to the GPL. OTOH, it seems the Julia performance comparison with Matlab was not entirely fair: dynamic allocation?!

Currently I code mostly in *gasp* C# these days (doing a lot of LINQ exorcisms and parallelism).

Regards recursion, except in very special circumstances parsers aren't performance critical and I don't use fractals or complexity theory often in my analyses.

Sergei Steshenko 2.13.2013 at 4:34 pm | *Permalink*

The problem with GPL is not even the GPL itself, but hypocrisy of its proponents – I've had it over my head being a member of Octave users list.

As to other languages – I am trying to teach myself OCaml because I want to grasp functional paradigm.

Regarding Julia – maybe I'll build it from sources – building from sources is my hobby kind of, and also a way of not being trapped by vendor lock-in – even though I'm on Linux. My point is that I do not want to be forced to make an upgrade in order to get newer versions of stuff.

Recursion is often a concise way to express oneself, and functional languages support it pretty well performance-wise.

« Previous *Next »*