The Julia Manifesto

Jonathon LeFaive
CSG Tech Talk
June 30, 2016

Why They Created Julia

… 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?) …

Why They Created Julia

… 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?) …

MIT Licensed

Free and Open Source

Why They Created Julia

... 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?) ...

LLVM-based JIT Compiler

	Fortran gcc 5.1.1	Julia 0.4.0	Python 3.4.3	R 3.2.2	Matlab R2015b	Octave 4.0.0	Mathe-matica 10.2.0	JavaScript V8 3.28.71.19	Go go1.5	LuaJIT gsl-shell 2.3.1	Java 1.8.0_45
fib	0.70	2.11	77.76	533.52	26.89	9324.35	118.53	3.36	1.86	1.71	1.21
parse_int	5.05	1.45	17.02	45.73	802.52	9581.44	15.02	6.06	1.20	5.77	3.35
quicksort	1.31	1.15	32.89	264.54	4.92	1866.01	43.23	2.70	1.29	2.03	2.60
mandel	0.81	0.79	15.32	53.16	7.58	451.81	5.13	0.66	1.11	0.67	1.35
pi_sum	1.00	1.00	21.99	9.56	1.00	299.31	1.69	1.01	1.00	1.00	1.00
rand_mat_stat	1.45	1.66	17.93	14.56	14.52	30.93	5.95	2.30	2.96	3.27	3.92
rand_mat_mul	3.48	1.02	1.14	1.57	1.12	1.12	1.30	15.07	1.42	1.16	2.36

Figure: benchmark times relative to C (smaller is better, C performance = 1.0).

Why They Created Julia

… 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?) …

M E T A P R O G R A M M I N G

```
julia> e = parse("1 + 2 * 3")
:(1 + 2 * 3)
julia> dump(e)
Expr
  head: Symbol call
  args: Array(Any,(3,))
    1: Symbol +
    2: Int64 1
    3: Expr
      head: Symbol call
      args: Array(Any,(3,))
        1: Symbol *
        2: Int64 2
        3: Int64 3
      typ: Any
  typ: Any
julia> Meta.show_sexpr(e)
(:call, :+, 1, (:call, :*, 2, 3))
julia> e2 = Expr(:call, :+, 1, Expr(:call, :*, 2, 3))
:(1 + 2 * 3)
julia> e == e2
true
```

Why They Created Julia

… 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?) …

Operators

```
1 + 2 # => 3
11 - 1 # => 10
10 * 3 # => 30
35 / 5 # => 7.0
5 \ 2 # => 2.5
5 // 2 # => 2.5
2 ^ 2 # => 4 # power not xor
13 % 10 # => 3
2 < 3 < 2 # => false
2 * π # => 6.283185307179586
1 ∈ [1, 3, 4] # => true
1 ∉ [1, 3, 4] # => false
[1, 2] ∪ [3, 4, 5] # => [1, 2, 3, 4, 5]
```

```
# Custom unicode function
∑(x,y) = x + y # => ∑ (generic function with 1 method)
∑(1,2) # => 3

# Unicode alias
∑ = + # => + (generic function with 171 methods)
∑(1,2,3,4) # => 10

# Fraction
typeof(2//5) # => Rational{Int64}
```

Why They Created Julia

… 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?) …

Control Flow

```
# create a variable
foo = 2

# if statement
if foo < 5
    println("Is less than 5.")
elseif foo > 5
    println("Is greater than 5.")
else
    println("Is 5.")
end

# iterate over an array
for color in ["red", "green", "blue"]
    println(color)
end
```

Accessing Arrays

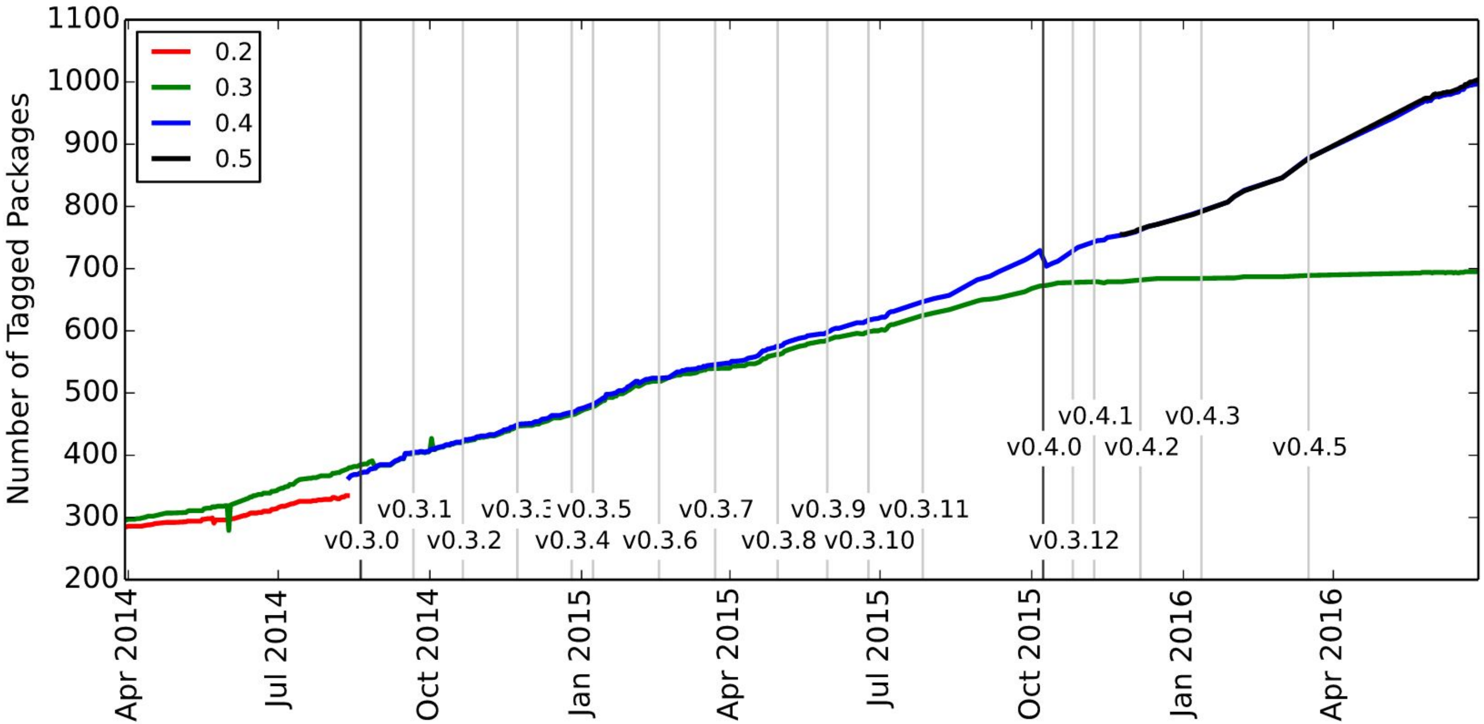
```julia
arr = [1, 2, 3, 4, 5]

arr[1] # => 1

arr[end] # => 5

arr[2:4] # => [2, 3, 4]

arr[2:end] # => [2, 3, 4, 5]

for i = 1:length(arr)
    println(arr[i])
end
```

Why They Created Julia

… 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?) …

Number of Packages by Julia Version



Working with Data

```
julia> using GLM, RDatasets

julia> form = dataset("datasets","Formaldehyde")
6x2 DataFrame
|-------|------|--------|
| Row # | Carb | OptDen |
| 1     | 0.1  | 0.086  |
| 2     | 0.3  | 0.269  |
| 3     | 0.5  | 0.446  |
| 4     | 0.6  | 0.538  |
| 5     | 0.7  | 0.626  |
| 6     | 0.9  | 0.782  |

julia> lm1 = fit(LinearModel, OptDen ~ Carb, form)
Formula: OptDen ~ Carb

Coefficients:
               Estimate  Std.Error  t value Pr(>|t|)
(Intercept)  0.00508571 0.00783368 0.649211   0.5516
Carb           0.876286  0.0135345  64.7444   3.4e-7


julia> confint(lm1)
2x2 Array{Float64,2}:
 -0.0166641  0.0268355
  0.838708   0.913864
```

Why They Created Julia

... 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?) ...

Strings

```
c = '\u2200'
typeof(c) # => Char
Int(c) # => 8704

str = "foobar"
typeof(str) # => ASCIIString
ustr = UTF8String("foobar")
typeof(ustr) # => UTF8String
ustr2 = "foobar \u2200"
typeof(ustr2) # => UTF8String


s = "1 + 2 = $(1 + 2)" # => "1 + 2 = 3"
world = "Earth"
msg = "Hello $(world)!" #=> "Hello Earth!"

m = match(r"(a|b)(c)?(d)", "acd") # => RegexMatch("acd", 1="a", 2="c", 3="d")
m[1] # => "a"
```

Why They Created Julia

… 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?) …

Built-in Linear Algebra

`Cholesky`	Cholesky factorization
`CholeskyPivoted`	Pivoted Cholesky factorization
`LU`	LU factorization
`LUTridiagonal`	LU factorization for Tridiagonal matrices
`UmfpackLU`	LU factorization for sparse matrices (computed by UMFPack)
`QR`	QR factorization
`QRCompactWY`	Compact WY form of the QR factorization
`QRPivoted`	Pivoted QR factorization
`Hessenberg`	Hessenberg decomposition
`Eigen`	Spectral decomposition
`SVD`	Singular value decomposition
`GeneralizedSVD`	Generalized SVD

Matrix type	`+`	`-`	`*`	`\`	**Other functions with optimized methods**
`Hermitian`				MV	`inv()`, `sqrtm()`, `expm()`
`UpperTriangular`			MV	MV	`inv()`, `det()`
`LowerTriangular`			MV	MV	`inv()`, `det()`
`SymTridiagonal`	M	M	MS	MV	`eigmax()`, `eigmin()`
`Tridiagonal`	M	M	MS	MV	
`Bidiagonal`	M	M	MS	MV	
`Diagonal`	M	M	MV	MV	`inv()`, `det()`, `logdet()`, `/()`
`UniformScaling`	M	M	MVS	MVS	`/()`

M (matrix)	An optimized method for matrix-matrix operations is available
V (vector)	An optimized method for matrix-vector operations is available
S (scalar)	An optimized method for matrix-scalar operations is available

Why They Created Julia

… 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?) …

Running External Programs

```
julia> run(pipeline(`cut -d: -f3 /etc/passwd`, `sort -n`, `tail -n5`))
210
211
212
213
214

julia> names = ["foo","bar","baz"]
3-element Array{ASCIIString,1}:
 "foo"
 "bar"
 "baz"

julia> exts = ["aux","log"]
2-element Array{ASCIIString,1}:
 "aux"
 "log"

julia> `rm -f $names.$exts`
`rm -f foo.aux foo.log bar.aux bar.log baz.aux baz.log`
```

Why They Created Julia

… 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?) …

Multiple Dispatch and Parametric Types

```
my_func(x,y) = 2x + y

my_func(1,2) # => 4

my_func('a','b') # => ERROR

function my_func(x::Char, y::Char)
  return 2 * Int64(x) + Int64(y)
End

my_func('a','b') # => 292
```

```
type Point
  x
  y
  z
end
Point(1, 3.0, "foobar")

type Point{T}
  x::T
  y::T
  z::T
end
Point(1, 2, 3)
```

```
type Point{T, T2}
  x::T
  y::T
  z::T2
end
Point(1, 2, 3.0)

type Point
  x::Float64
  y::Float64
  z::Float64
end
Point(1.0, 2.0, 3.0)
```

Why They Created Julia

… 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?) …

Why They Created Julia

... 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?) ...

loop_sum.jl

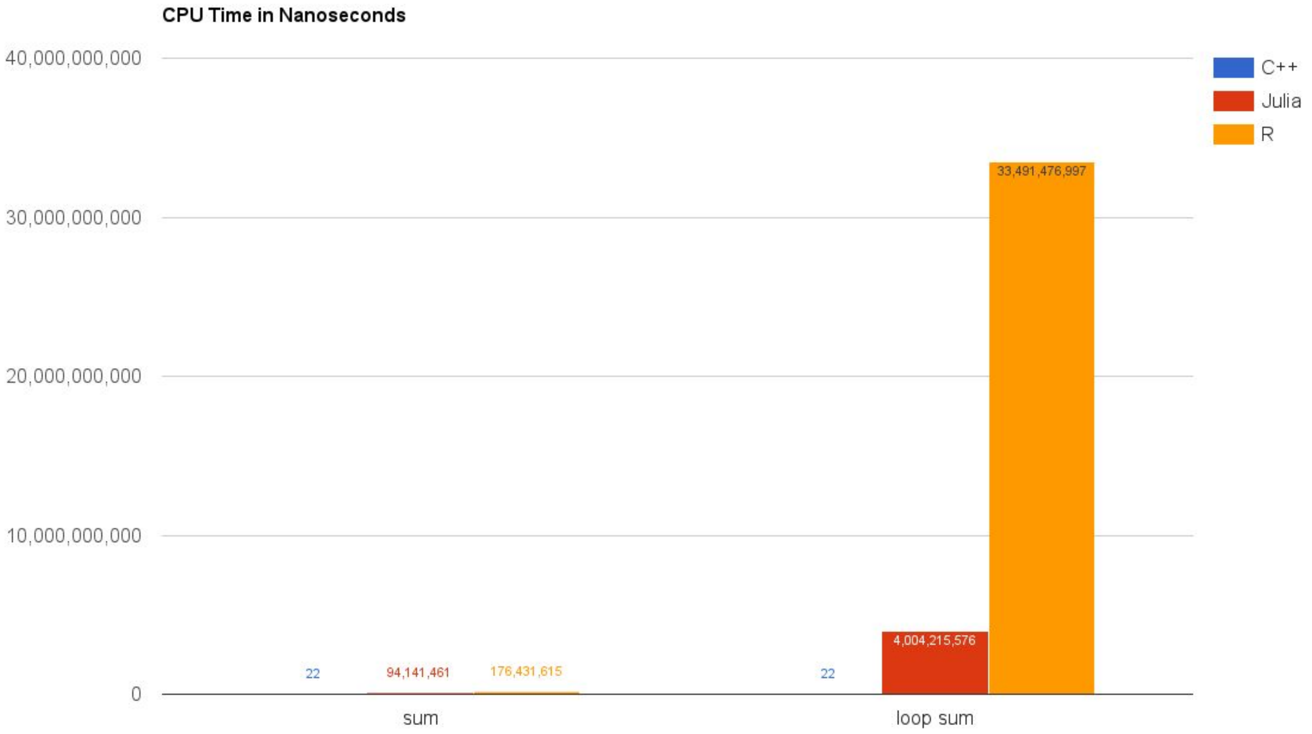
```julia
function loop_sum(arr)
  ret = 0.0
  for val in arr
    ret += val
  end
  return ret
end
```

loop_sum.R

```
loop_sum <- function(arr) {
  ret <- 0.0
  for (val in arr) {
    ret <- ret + val
  }
  return(ret)
}
```

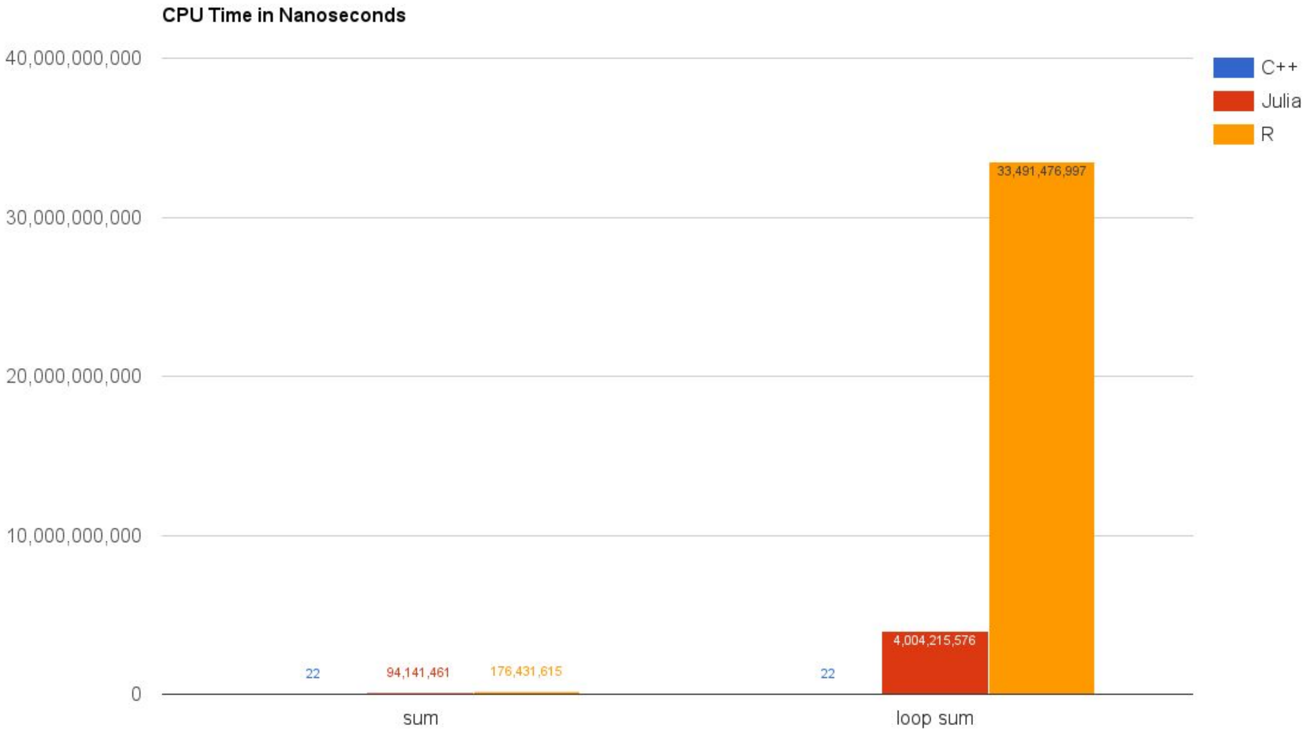
loop_sum.cpp

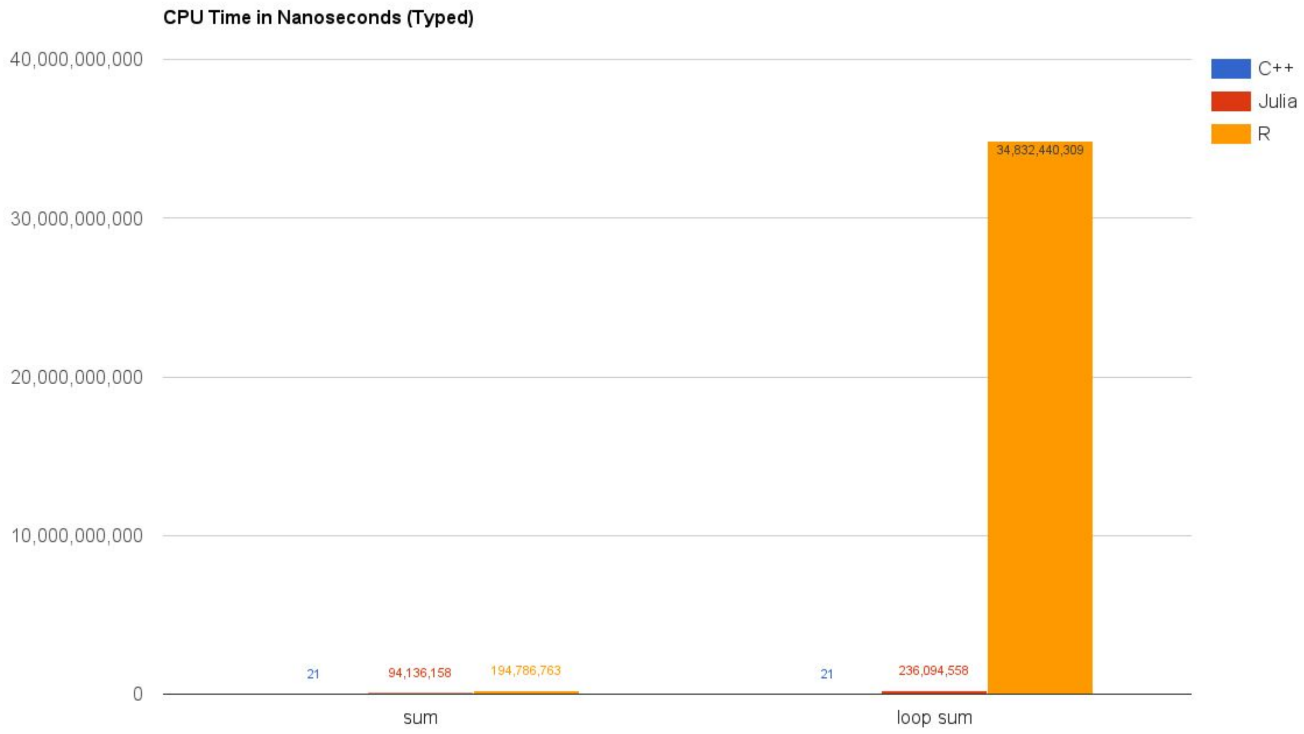
```
double loop_sum(const std::vector<double>& arr)
{
  double ret = 0.0;
  const std::size_t arr_size = arr.size();
  for (std::size_t i = 0; i < arr_size; ++i)
  {
    ret += arr[i];
  }
  return ret;
}
```

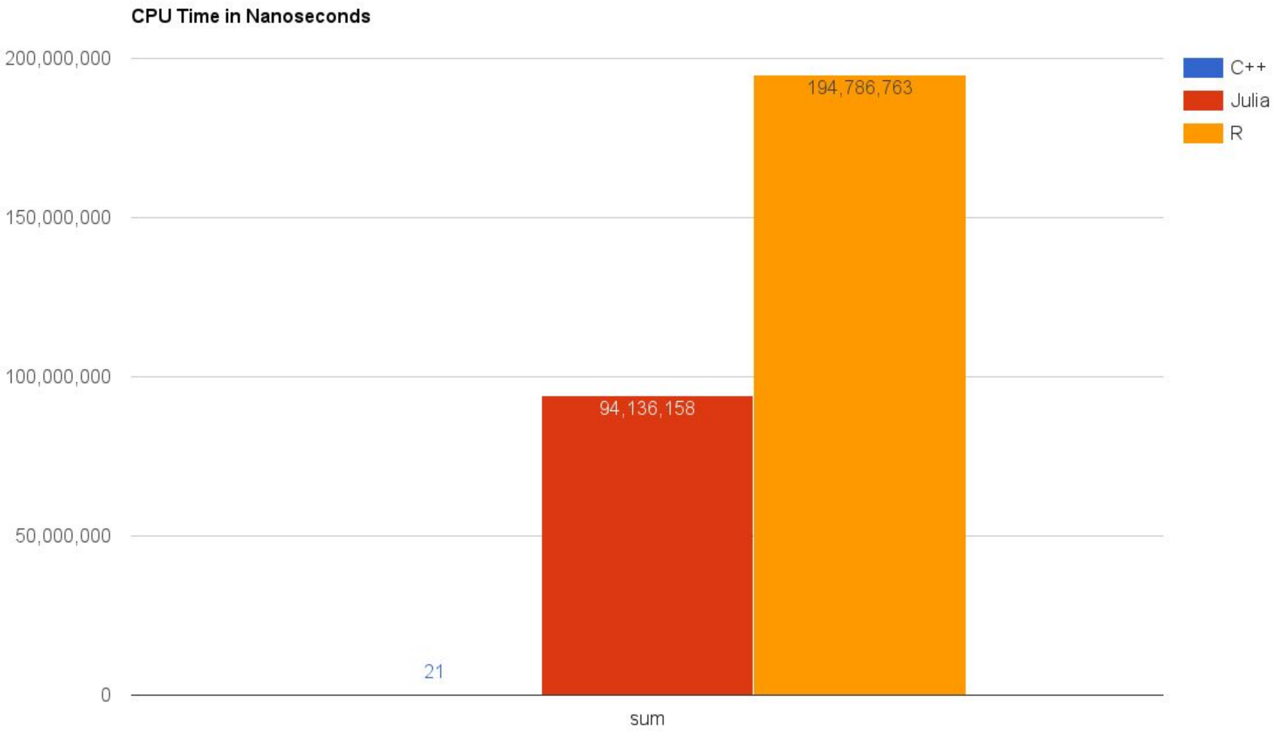

CPU Time in Nanoseconds
40,000,000,000
30,000,000,000
20,000,000,000
10,000,000,000
0
C++
Julia
R
22
94,141,461
176,431,615
22
4,004,215,576
33,491,476,997
sum
loop sum

loop_sum.jl (updated)

```
function loop_sum(arr::Array{Float64,1})
  ret::Float64 = 0.0
  for i = 1:length(arr)
    ret += arr[i]
  end
  return ret
end
```


CPU Time in Nanoseconds
40,000,000,000
30,000,000,000
20,000,000,000
10,000,000,000
0
C++
Julia
R
22
94,141,461
176,431,615
22
4,004,215,576
33,491,476,997
sum
loop sum


CPU Time in Nanoseconds (Typed)
40,000,000,000
30,000,000,000
20,000,000,000
10,000,000,000
0
C++
Julia
R
21
94,136,158
194,786,763
21
236,094,558
34,832,440,309
sum
loop sum


CPU Time in Nanoseconds
200,000,000
150,000,000
100,000,000
50,000,000
0
194,786,763
94,136,158
21
sum
C++
Julia
R

Further Reading

https://learnxinyminutes.com/docs/julia/

http://docs.julialang.org/en/release-0.4/stdlib/c/

http://docs.julialang.org/en/release-0.4/stdlib/parallel/

http://docs.julialang.org/en/release-0.4/manual/modules/