Introduction to

(http://julialang.org/)

http://julialang.org/ (http://julialang.org/)

Daniel Molina Cabrera

dmolina@decsai.ugr.es (mailto:dmolina@decsai.ugr.es)
https://github.com/dmolina/julia_presentacion (https://github.com/dmolina/julia_presentacion)

About me

Assistant professor at Computer Science and Artificial Intelligence, University of Granada (Spain)

Interests

- Metaheuristics
- Machine Learning
- Software Development

About me

Free Software supporter.

Several *frikis* interests

- Linux user (not Windows).
- Emacs.
- *Pythonic* and lastly *Julian*

What is Julia?

Julia Motivation in 2009

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?)

So

- It is a Free Software Language (not as Matlab).
- For general proposed, but done by and for researchers.
- Very efficient (functions are compiled and later not, not interpreted).
- Very similar to Python (and to Matlab) in the good things.

Python Code *vs* Julia Code

```
def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n-1)+fib(n-2)

%time fib(40)
```

In [2]:

```
function fib(n)
    if n ≤ 1
        1
    else
        fib(n-1)+fib(n-2)
    end
end
@time fib(40)
@time fib(40)
```

```
  0.663471 seconds (2.53 k allocations: 137.844 KiB)
  0.664287 seconds (5 allocations: 176 bytes)
```

Out[2]:

```
165580141
```

TOOLBOX · 30 JULY 2019

Julia: come for the syntax, stay for the speed

Researchers often find themselves coding algorithms in one programming language, only to have to rewrite them in a faster one. An up-and-coming language could be the answer.

Jeffrey M. Perkel

Julia Evolution

Julia Origin

- Developed by several Phd students in MIT since 2009, first public version in 2012.
- Version 1.0 in August 2018.
    - More than 2 millions of downloads.
    - 750 have committed (me included).
    - +2400 libraries, some of them with a high quality.

Recent Julia events

- New package system at the end of 2018.
- New debugger in 2019.

Expected changes

- Version 1.3 with better support of threads.
- Improve the time loading libraries in version 1.4.
- In a future to create binaries.

Julia Reason of existence

Avoid the two-languages problem

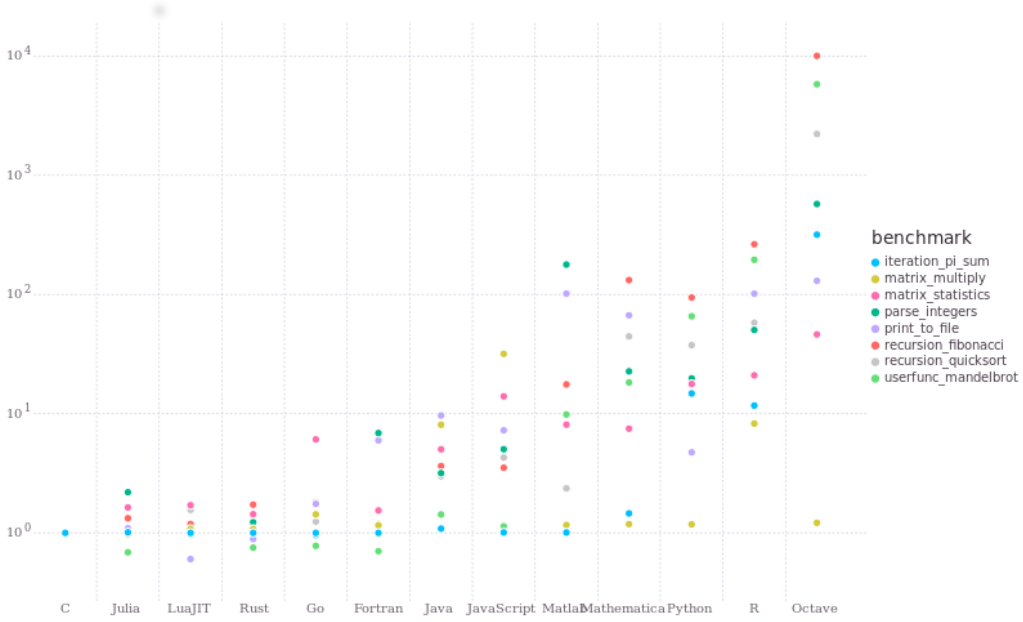
- One simple for flexibility but slow (as Python).
- One complex but quickly.

Ousterholt's dichotomy

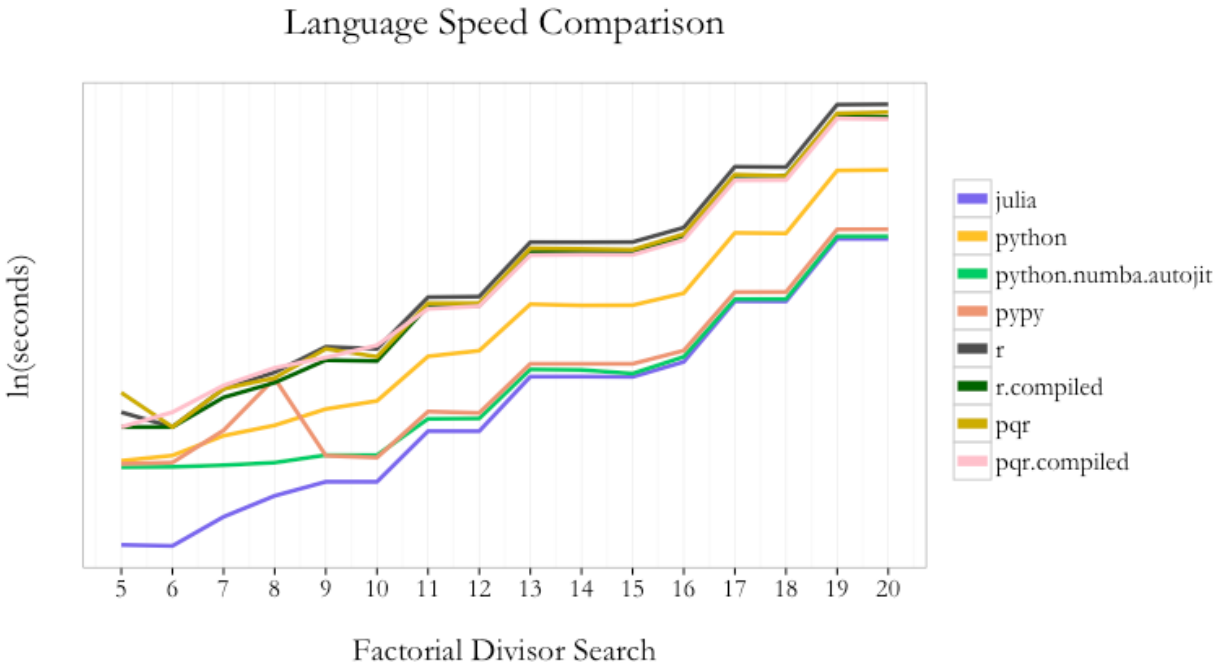
static	dynamic
compiled	interpreted
user types	standard types
fast	slow
hard	easy

It can be different

Some Benchmarks

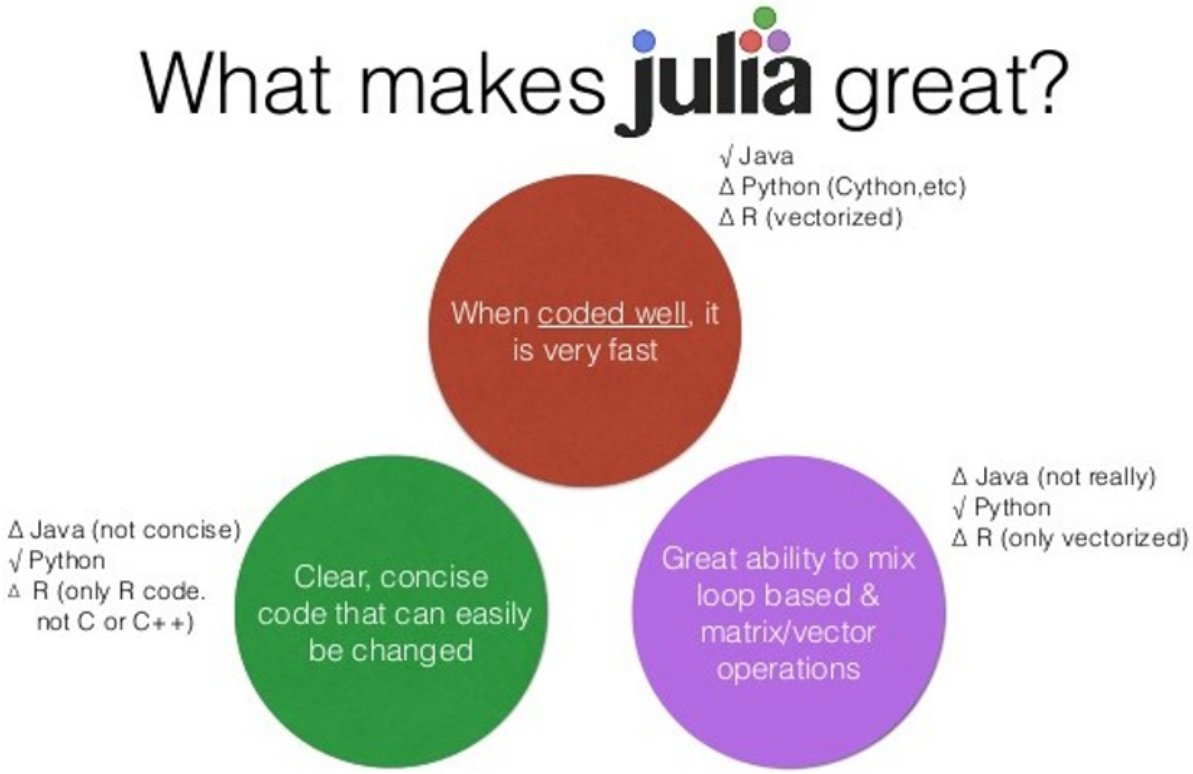


Results



Source: https://randyzwitch.com/python-pypy-julia-r-pqr-jit-just-in-time-compiler/ (https://randyzwitch.com/python-pypy-julia-r-pqr-jit-just-in-time-compiler/)

What makes julia great?

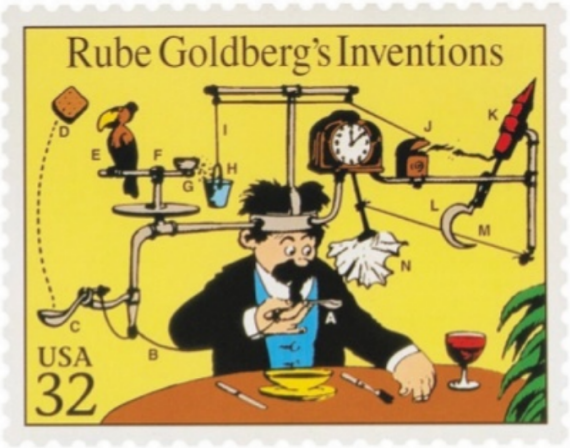


The aim is to simplify our life

My Data Science Stack circa 2009

In a single project, I was using:

- **Matlab** for linear algebra
- **R** for stats & visualization
- **C** for the fast stuff
- **Ruby** to tie it all together



Rube Goldberg Revised

Here's my data science stack today:

- ~~Matlab~~ **Julia** for linear algebra
- ~~R~~ **Julia** for stats & visualization
- ~~C~~ **Julia** for the fast stuff
- ~~Ruby~~ **Julia** to tie it all together

My case is similar

- I work in metaheuristics in complex problems.
- The majority uses Matlab, I use Python.

Running shared code

- Usually I receive code of other algorithms in Matlab, license problems to run them.
- Run them in Octave means to take a lot of time running.
- Traslate the code to Python is not easy:
    - Different Syntax.
    - Sources of problems: In Matlab indexes start with 1, while in Python with 0.
    - In Python is needed to use numpy, syntax even further of original Matlab.

For my algorithms

- Python for prototype.
- Numpy for performance.
- Avoid conditionals, try to use vectored operations, even when there are more complex to understand.
- Identify bottlenecks.
- Sometimes use C++ for re-implementing the slowest components (Cython).

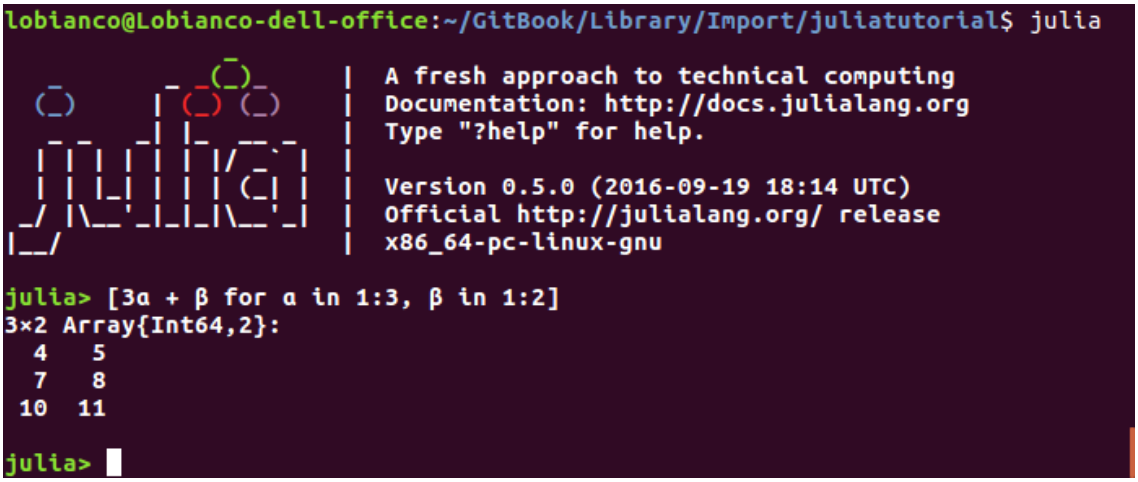
Using Julia

- It is not needed neither Numpy neither cython.
- Libraries only availables are Python using PyCall.
- Easier to call C/C++ code than from Python.

Comparing Julia with Python

Similarities

- Very similar syntax.
- Interactive mode (REPL) similar to IPython.



- Julia mode, by default.
- Package mode [, for searching and managing packages.
- Shell Mode \; for running commands.

Let's go to see them.

Official Packages Repository

In [3]:
```
using Pkg
Pkg.add("OhMyREPL")
```

```
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
[1mFetching: [========================================>]  100.0 %.0 %
[============>                             ]  27.6 %]  54.9 %>   ]  91.1 % Reso
lving package versions...
 Installed DocumentFormat ── v1.1.1
 Installed OffsetArrays ──── v0.11.4
 Installed PlotThemes ─────── v1.0.1
 Installed BangBang ───────── v0.3.9
 Installed JuliaInterpreter ─ v0.7.6
  Updating `~/.julia/environments/v1.3/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.3/Manifest.toml`
  [198e06fe] ↑ BangBang v0.3.8 ⇒ v0.3.9
  [ffa9a821] ↑ DocumentFormat v1.1.0 ⇒ v1.1.1
  [aa1ae85d] ↑ JuliaInterpreter v0.7.5 ⇒ v0.7.6
  [6fe1bfb0] ↑ OffsetArrays v0.11.3 ⇒ v0.11.4
  [ccf2f8ad] ↑ PlotThemes v1.0.0 ⇒ v1.0.1
```

In [4]:
```
using OhMyREPL
```

```
┌ Info: Precompiling OhMyREPL [5fb14364-9ced-5910-84b2-373655c76a03]
└ @ Base loading.jl:1273
```

More similarities

- Packages usage, import with **import** o with **using**.
- Automatic tests system.

In [5]:
```
import Distributions
rand(Distributions.Uniform(-5, 5), 3)
```

```
┌ Info: Precompiling Distributions [31c24e10-a181-5473-b8eb-7969acd0382f]
└ @ Base loading.jl:1273
```

Out[5]:
```
3-element Array{Float64,1}:
 -4.002472679130271
 -4.299497402698087
 -4.219969996661906
```

In [6]:
```
using Distributions: Uniform
rand(Uniform(-5, 5), 3)
```

Out[6]:
```
3-element Array{Float64,1}:
 -4.50963886640892
 -0.3465556865137609
  4.597014906348424
```

In [7]:
```
using Distributions
rand(Uniform(-5, 5), 3)
```

Out[7]:
```
3-element Array{Float64,1}:
  2.4261227535053926
 -4.2103109320071
  2.0735008203413052
```

More similarities

- It is not need to declare variables.
- Vectors.
- Iterators (for, enumerate, zip, ...).
- Lambda functions.

In [8]:
```
var = [1 2 3]
println(var)
var = 3
println(var)
filter(x->x[end]=='2', ["user$i" for i in 1:3])
```

```
[1 2 3]
3
```

Out[8]:
```
1-element Array{String,1}:
 "user2"
```

Data Structures in the language

- Arrays, inside the lenguaje.
- String, UTF-8.
- Dictionaries.
- Sets

In [9]:
```
valores = ["hello", "bye!"];
dict = Dict(i=>val for (i, val) in enumerate(valores))

for (k,v) in dict
    println(k, v)
end
```

```
2bye!
1hello
```

In [10]:
```
println(keys(dict))
```

```
[2, 1]
```

In [11]:
```
collect(Set([1, 2, 4, 2, 4]))
```

Out[11]:
```
3-element Array{Int64,1}:
 4
 2
 1
```

Differences in the syntax

- *def* => *function* (as Matlab).
- No use tab for separate, it uses **end**.
- The good thing is has not not begin, and *end* is limited (not before an else, for instance).

In [12]:
```
function fibo(n::Int)
    (a, b) = (1, 1)

    for _ in 1:n-1
        (a, b) = (a+b, a)
        end
    a
end
@time @show fibo(40)
```

```
fibo(40) = 165580141
  0.039249 seconds (61.35 k allocations: 3.026 MiB)
```

Out[12]:
```
165580141
```

Optional typing

- It is not needed, Julia deduces theme when you call the function.
- There are generic types, usefuls for avoid errors (Int, Float, Real, Number, AbstractString, ...).

In [13]:
```
sphere(x)= sum(x.^2)
sphere(3)
```

Out[13]:
```
9
```

In [14]:
```
sphere([3, 4, 5])
```

Out[14]:
```
50
```

It is efficient because it can compile for each concrete type

In [15]:
```
@code_warntype sphere([3, 4, 5])
```

```
Variables
  #self#::Core.Compiler.Const(sphere, false)
  x::Array{Int64,1}

Body::Int64
1 ─ %1 = Core.apply_type(Base.Val, 2)::Core.Compiler.Const(Val{2}, false)
│   %2 = (%1)()::Core.Compiler.Const(Val{2}(), false)
│   %3 = Base.broadcasted(Base.literal_pow, Main.:^, x, %2)::Base.Broadcast.Br
oadcasted{Base.Broadcast.DefaultArrayStyle{1},Nothing,typeof(Base.literal_po
w),Tuple{Base.RefValue{typeof(^)},Array{Int64,1},Base.RefValue{Val{2}}}}
│   %4 = Base.materialize(%3)::Array{Int64,1}
│   %5 = Main.sum(%4)::Int64
└──      return %5
```

It is efficient because it can compile for each concrete type

In [16]:
```
fabsurda(x)=x^2+3;
```

In [17]:
```
@code_warntype fabsurda(3)
```

```
Variables
  #self#::Core.Compiler.Const(fabsurda, false)
  x::Int64

Body::Int64
1 ─ %1 = Core.apply_type(Base.Val, 2)::Core.Compiler.Const(Val{2}, false)
│   %2 = (%1)()::Core.Compiler.Const(Val{2}(), false)
│   %3 = Base.literal_pow(Main.:^, x, %2)::Int64
│   %4 = (%3 + 3)::Int64
└──      return %4
```

In [18]:
```
@code_warntype fabsurda(1//3)
```

```
Variables
  #self#::Core.Compiler.Const(fabsurda, false)
  x::Rational{Int64}

Body::Rational{Int64}
1 ─ %1 = Core.apply_type(Base.Val, 2)::Core.Compiler.Const(Val{2}, false)
│   %2 = (%1)()::Core.Compiler.Const(Val{2}(), false)
│   %3 = Base.literal_pow(Main.:^, x, %2)::Rational{Int64}
│   %4 = (%3 + 3)::Rational{Int64}
└──      return %4
```

In [19]:
```
@code_llvm fabsurda(3)
```

```
;  @ In[16]:1 within `fabsurda'
define i64 @julia_fabsurda_18630(i64) {
top:
; ┌ @ intfuncs.jl:244 within `literal_pow'
; │┌ @ int.jl:54 within `*'
    %1 = mul i64 %0, %0
; └└
; ┌ @ int.jl:53 within `+'
   %2 = add i64 %1, 3
; └
  ret i64 %2
}
```

Working with indexes

- Index starts with 1, not 0.
- It is not so probelmatic, anyway it is very common to use iterator and/or enumerate:

In [20]:
```
values = ["One", "Two", "Three"]

for i = 1:length(values)
    println("$i: ", values[i])
end
println("-"^8)
for val in values
    println(val)
end
for (i, val) in enumerate(values)
    println("$i: $val")
end
```

```
1: One
2: Two
3: Three
--------
One
Two
Three
1: One
2: Two
3: Three
```

Functional, not Object Oriented

- Packages define structs and functions using them.
- Multiple Dispatch:
    - Not cost.
    - Example: Table interface used by multiple implementations (DataFrames, ...).
- Functional interface.
- Operator |> to concat operations.

In [21]:
```
msg = "Hello"
println(uppercase(msg))
msg |> uppercase |> println
```

```
HELLO
HELLO
```

In [22]:

```
struct MyData
    n::Int64
    s::String
end

using Base: println

function Base.println(io::IO, v::MyData)
    println(io, "My println: ", string(v.n), ": ", v.s)
end

data = MyData(4, "Hello")
println(data)
@show methods(println)
@show methodswith(typeof(data))
```

```
My println: 4: Hello
methods(println) = # 4 methods for generic function "println":
[1] println(io::IO) in Base at coreio.jl:5
[2] println(io::IO, v::MyData) in Main at In[22]:9
[3] println(io::IO, xs...) in Base at strings/io.jl:73
[4] println(xs...) in Base at coreio.jl:4
methodswith(typeof(data)) = Method[println(io::IO, v::MyData) in Main at In[2
2]:9]
```

Out[22]: 1-element Array{Method,1}:

- println(io::**IO**, v::**MyData**) in Main at In[22]:9

Dot operator

- It allows us to vectorise any function.
- It produces more efficient operations.

In [23]:

```
op(x) = x*x-5
@show op.([1,2,3, 4, 5, 6, 7])
# println.(op.([1, 2, 4]));
```

```
op.([1, 2, 3, 4, 5, 6, 7]) = [-4, -1, 4, 11, 20, 31, 44]
```

More complex example

In [24]:
```
A = rand(Uniform(-5, 5), 3000); B = rand(Uniform(-5, 5), 3000);
```

In [25]:
```
C = similar(A);
function add1!(C, A, B)
    C .= A .+ B
end
@btime add1!(C, A, B);
```

```
  782.811 ns (0 allocations: 0 bytes)
```

In [26]:
```
add2!(C, A, B) = @. C = A+B
@btime add2!(C, A, B);
```

```
  772.692 ns (0 allocations: 0 bytes)
```

In [27]:
```
function add3!(C, A, B)
@inbounds @simd for i in 1:length(A)
    C[i] = A[i]+B[i]
    end
end

@btime add3!(C, A, B)
```

```
  773.067 ns (0 allocations: 0 bytes)
```

Files usage

In [28]:
```
users = ["Students$(elem)" for elem in 1:2:8]
```

Out[28]:
```
4-element Array{String,1}:
 "Students1"
 "Students3"
 "Students5"
 "Students7"
```

In [29]:
```
open("users.txt", "w") do file
    for user in users
        println(file, user)
    end
end
```

In [30]:
```
open("users.txt") do file
    for line in readlines(file)
        println(strip(line))
    end
end
```

```
Students1
Students3
Students5
Students7
```

Communication with other languages

- Calling Python/R/C++ code.

```c
// Code in libsaludo.so
#include <stdio.h>

char msg[80];

char *getname(int num) {
  sprintf(msg, "Hello to the %d asistants", num);
  return msg;
}
```

Calling the code

It is compiled as .so library, and later it can be called from Julia.

In [31]:
```
using Libdl
lib_ptr = dlopen("./libsaludo.so")
get_name = dlsym(lib_ptr, "getname")
result = ccall(get_name, Cstring, (Cint, ), 16)
println(unsafe_string(result))
```

```
Hola a todos los 16 asistentes
```

Calling Python code

In [32]:
```
using PyCall
math=pyimport("math")
println(math.sin(3)) # Call to sin function from Python math
np=pyimport("numpy")
np.random.rand(5) # It converts to Julia array
```

```
0.1411200080598672
```

Out[32]:
```
5-element Array{Float64,1}:
 0.46084979674501225
 0.7373994577304266
 0.9538972752780361
 0.06493803706344503
 0.19926060636955512
```

Help mode

It encourages the addition of descriptions for each function.

In [33]:
```
"""
This function calculates the fibunacci number.

n: Idel cual calcular fibonacci.
"""
function fibo(n::Int)
    (a, b) = (1, 1)

    for _ in 1:n-1
        (a, b) = (a+b, a)
    end
    a
end
@time @show fibo(40)
```

```
fibo(40) = 165580141
  0.009521 seconds (16.93 k allocations: 925.431 KiB)
```

Out[33]:
```
165580141
```

In [34]:
```
?fibo
```

```
search: fibo fib find_library failprob unsafe_pointer_to_objref
```

Out[34]:

This function calculates the fibunacci number.

n: Idel cual calcular fibonacci.

Flexible

It can be customized

In [35]:
```
"a" in "hola"
```

```
use occursin(x, y) for string containment

Stacktrace:
 [1] error(::String) at ./error.jl:33
 [2] in(::String, ::String) at ./strings/search.jl:533
 [3] top-level scope at In[35]:1
```

In [36]:
```
using Base: in

Base.in(a::AbstractString, b::AbstractString) = occursin(a, b)

"a" in "hola"
```

Out[36]:
```
true
```

Macros

- They start with @.
- They can generate code.

In [37]:
```
macro load(x) # using sometimes is slow, this macro only does it if it was not previously loaded
    if !(x in names(Main, all=false, imported=true))
        return :(using $x)
    end
end

@time using OhMyREPL
@time @load OhMyREPL
```

```
  0.729349 seconds (963.56 k allocations: 45.481 MiB, 1.63% gc time)
  0.000001 seconds (3 allocations: 144 bytes)
```

It can be build very easily the documentation, Package [Documenter.jl (https://juliadocs.github.io/Documenter.jl/stable/)](https://juliadocs.github.io/Documenter.jl/stable/):

- Markdown to describe the documentation.
- It can insert the API documentation.
- Tests as documentation.

Scientific Community

Popular packages/libraries in: https://pkg.julialang.org/docs/ (https://pkg.julialang.org/docs/)

- Notebook: IJulia (https://pkg.julialang.org/docs/IJulia/nfu7T/1.20.2/)
- Scientific: QueryVerse (https://www.queryverse.org/)
    - DataFrames: DataFrames.jl (https://github.com/JuliaData/DataFrames.jl)
    - Visualization: VegaLite (https://www.queryverse.org/VegaLite.jl/stable/)
- Visualization: Gadfly (https://gadflyjl.org/stable/tutorial/), Plots (https://docs.juliaplots.org/latest/tutorial/#tutorial-1), StatsPlots (https://github.com/JuliaPlots/StatsPlots.jl)
- Deep Learning Libraries: Flux.jl (https://fluxml.ai/), KNet.jl (https://github.com/denizyuret/Knet.jl).
- Machine Learning: ScikitLearn.jl (https://github.com/cstjean/ScikitLearn.jl), MLJ (https://pkg.julialang.org/docs/MLJ/rAU56/0.5.4/)

Scientific Community

State-of-the-art libraries

- Optimization library: JuMP (https://pkg.julialang.org/docs/JuMP/DmXqY/0.20.1/quickstart.html#Quick-Start-Guide-1)
- Differential Equations: DifferentialEquations (https://docs.juliadiffeq.org/latest/)
- Statistical Distributions: Distributions (https://juliastats.org/Distributions.jl/latest/starting/)

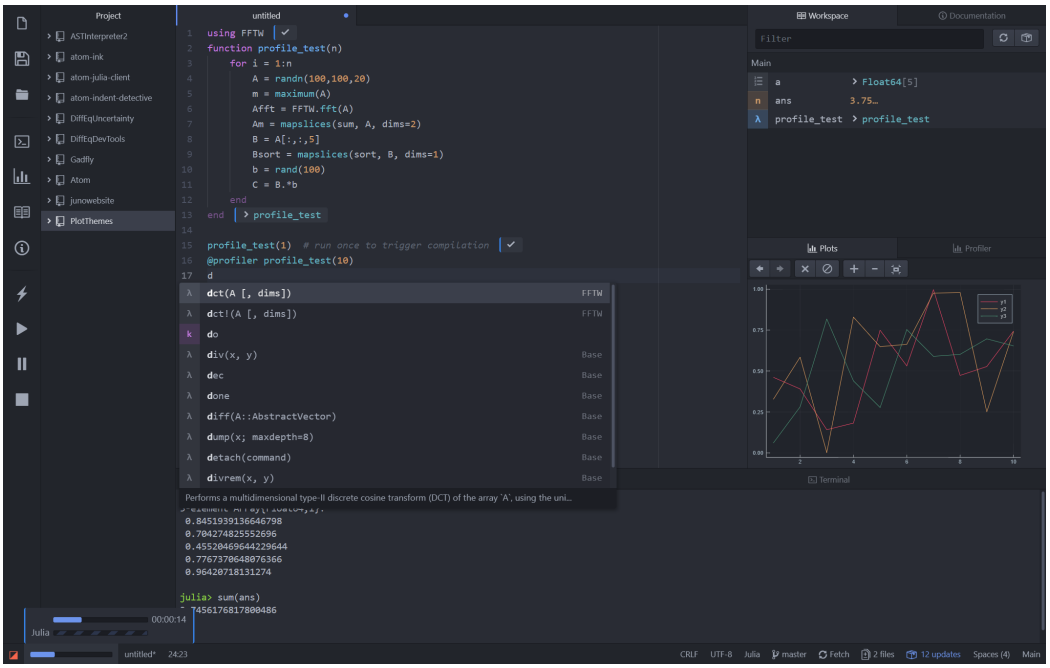
More generic

- web pages: Genie (https://genieframework.github.io/Genie.jl/)
- Databases: Octo (https://github.com/wookay/Octo.jl)

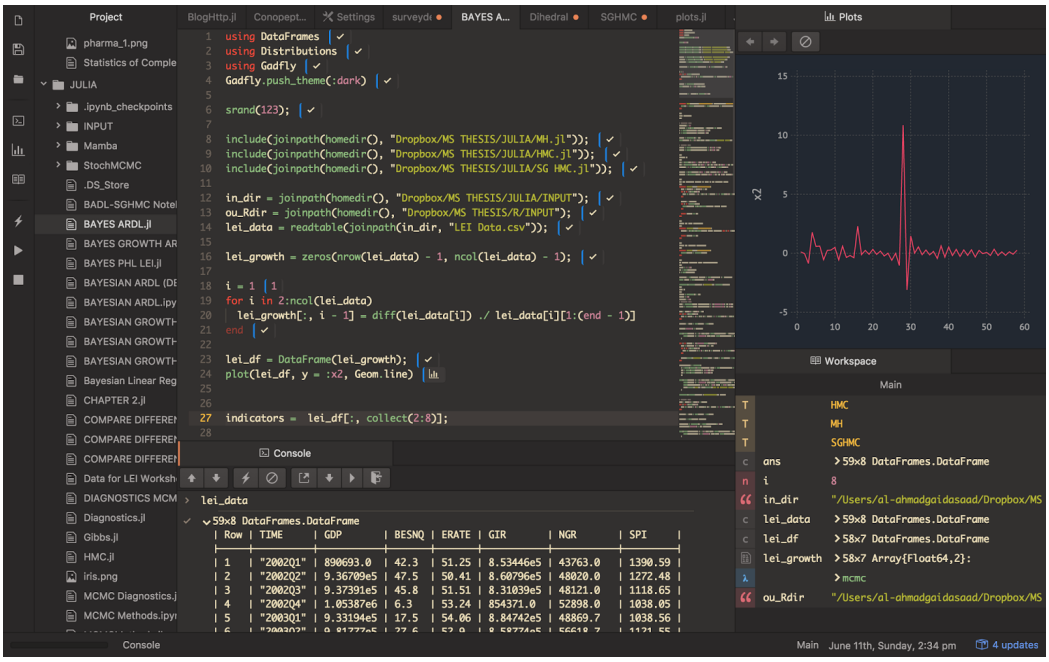
IDEs

- Juno: Atom with Julia packages pre-installed.
- Visual Code.
- SublimeText.
- Emacs.

Juno


Project
ASTInterpreter2
atom-ink
atom-julia-client
atom-indent-detective
DiffEqUncertainty
DiffEqDevTools
Gadfly
Atom
junowebsite
PlotThemes
untitled
using FFTW
function profile_test(n)
for i = 1:n
A = randn(100,100,20)
m = maximum(A)
Afft = FFTW.fft(A)
Am = mapslices(sum, A, dims=2)
B = A[:,:,5]
Bsort = mapslices(sort, B, dims=1)
b = rand(100)
C = B.*b
end
end
profile_test(1) # run once to trigger compilation
@profiler profile_test(10)
d
dct(A [, dims])
dct!(A [, dims])
do
div(x, y)
dec
done
diff(A::AbstractVector)
dump(x; maxdepth=8)
detach(command)
divrem(x, y)
FFTW
Base
Performs a multidimensional type-II discrete cosine transform (DCT) of the array `A`, using the uni...
Workspace
Documentation
Filter
Main
a
Float64[5]
ans
3.75...
profile_test
Plots
Profiler
Terminal
0.8451939136646798
0.704274825552696
0.45520469644229644
0.7767370648076366
0.96420718131274
julia> sum(ans)
3.745617617800486
00:00:14
Julia
untitled*
24:23
CRLF
UTF-8
Julia
master
Fetch
2 files
12 updates
Spaces (4)
Main

Visual Code


Project
pharma_1.png
Statistics of Comple
JULIA
.ipynb_checkpoints
INPUT
Mamba
StochMCMC
.DS_Store
BADL-SGHMC Notel
BAYES ARDL.jl
BAYES GROWTH AR
BAYES PHL LEI.jl
BAYESIAN ARDL (DE
BAYESIAN ARDL.ipy
BAYESIAN GROWTH-
BAYESIAN GROWTH-
BAYESIAN GROWTH-
Bayesian Linear Reg
CHAPTER 2.jl
COMPARE DIFFERE
COMPARE DIFFERE
COMPARE DIFFERE
Data for LEI Worksh
DIAGNOSTICS MCM
Diagnostics.jl
Gibbs.jl
HMC.jl
iris.png
MCMC Diagnostics.j
MCMC Methods.ipy
BlogHttp.jl
Conopept...
Settings
surveyd
BAYES A...
Dihedral
SGHMC
plots.jl
using DataFrames
using Distributions
using Gadfly
Gadfly.push_theme(:dark)
srand(123);
include(joinpath(homedir(), "Dropbox/MS THESIS/JULIA/MH.jl"));
include(joinpath(homedir(), "Dropbox/MS THESIS/JULIA/HMC.jl"));
include(joinpath(homedir(), "Dropbox/MS THESIS/JULIA/SG HMC.jl"));
in_dir = joinpath(homedir(), "Dropbox/MS THESIS/JULIA/INPUT");
ou_Rdir = joinpath(homedir(), "Dropbox/MS THESIS/R/INPUT");
lei_data = readtable(joinpath(in_dir, "LEI Data.csv"));
lei_growth = zeros(nrow(lei_data) - 1, ncol(lei_data) - 1);
i = 1
for i in 2:ncol(lei_data)
lei_growth[:, i - 1] = diff(lei_data[i]) ./ lei_data[i][1:(end - 1)]
end
lei_df = DataFrame(lei_growth);
plot(lei_df, y = :x2, Geom.line)
indicators = lei_df[:, collect(2:8)];
Console
lei_data
59x8 DataFrames.DataFrame
Row | TIME | GDP | BESNQ | ERATE | GIR | NGR | SPI
1 | "2002Q1" | 890693.0 | 42.3 | 51.25 | 8.53446e5 | 43763.0 | 1390.59
2 | "2002Q2" | 9.36709e5 | 47.5 | 50.41 | 8.60796e5 | 48020.0 | 1272.48
3 | "2002Q3" | 9.37391e5 | 45.8 | 51.51 | 8.31039e5 | 48121.0 | 1118.65
4 | "2002Q4" | 1.05387e6 | 6.3 | 53.24 | 854371.0 | 52898.0 | 1038.05
5 | "2003Q1" | 9.33194e5 | 17.5 | 54.06 | 8.84742e5 | 48869.7 | 1038.56
Plots
x2
Workspace
Main
HMC
MH
SGHMC
ans > 59x8 DataFrames.DataFrame
i 8
in_dir "/Users/al-ahmadgaidasaad/Dropbox/MS
lei_data > 59x8 DataFrames.DataFrame
lei_df > 58x7 DataFrames.DataFrame
lei_growth > 58x7 Array{Float64,2}:
> mcmc
ou_Rdir "/Users/al-ahmadgaidasaad/Dropbox/MS
Console
Main June 11th, Sunday, 2:34 pm
4 updates

Other Julia advantages

- Mathematical notation.

In [38]:
```
fun_mat(x; α=0.5)=3α+2x

fun_mat(3, α=0.1)
```

Out[38]:
```
6.3
```

- Easier to create an official package than using pip.
    - Created from pkg, recommendable to use PkgTemplate.jl
- Reasonable notation, functions which change the parameters end with "!".

Suggestions about Julia

- Loading libraries can be costly.
    - Load it once, and later use include to load the files.
    - Use Revise that is able to reload automatically the files when they change.
- For performance, the variable should not change its type.
- Not use [], but Type[].

In [39]:
```
a = []
push!(a, 3)
@show typeof(a)
b = Int[]
push!(b, 3)
@show typeof(b);
```

```
typeof(a) = Array{Any,1}
typeof(b) = Array{Int64,1}
```

Suggestions about Julia

- Read documentation, to adapt the style to the language.
- To document own functions, be clear.
- Define types in parameters only for security, not abuse.
- Use packages, simple and it maintains clear the dependencies of your code.
- Avoid global variables, only use global constants (keywork **const**).
- Put all the code inside a function, as main().

Julia main Disadvantages

- Great latency compiling (the time of the first plot).
- Several libraries are not very mature (many of them should have better documentation).
- It lacks static compilation.
- Very focussed arounf scientific environment, it lacks more general libraries.
- Costly type union.
- It could be better error messages.

Thanks you for the attention

danimolina@gmail.com

https://github.com/dmolina/julia_presentacion (https://github.com/dmolina/julia_presentacion)

normal