



[\(http://julialang.org/\)](http://julialang.org/)

[http://julialang.org/ \(http://julialang.org/\)](http://julialang.org/)

Daniel Molina Cabrera

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

Sobre mí



Profesor de Dpto Ciencias de la Computación e Inteligencia Artificial

Intereses

- Meta-heurísticas
- Machine Learning
- Desarrollo de Software

Sobre mí



Convencido del Software Libre

Intereses *frikis*

- Linuxero convencido.
- Aficionado a Emacs.
- Pythonero y últimamente Julianero

¿Qué es Julia?

Motivación de Julia en 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?)

Por tanto

- Es un Lenguaje Software Libre.
- De propósito general, pero hecho por y para científicos.
- Eficiente (no interpreta las funciones, las compila y ejecuta).
- Muy similar a Python.

Código en Python vs Julia

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

%time fib(40)
```

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

```
0.682273 seconds (3.96 k allocations: 228.594 KiB)
0.677145 seconds (5 allocations: 176 bytes)
```

```
Out[1]: 165580141
```

MENU ▾

nature

Subscribe



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

Evolución de Julia

Origen de Julia

- Desarrollado por varios estudiantes de Doctorado del MIT desde 2009, primera versión pública en 2012.
- Versión 1.0 en Agosto de 2018.
 - Más de 2 millones de descargas, estimado medio millón de usuarios habituales.
 - 750 han subido commit (yo incluído).
 - +2400 paquetes, algunos de mucha calidad.

Recientes Hitos de Julia

- Sistema de Paquetes a Final de 2018.
- Reescrito depurador en 2019.

Futuros cambios

- Versión 1.3 con mejor soporte de hebras (Release).
- Mejorar la carga de librerías en Versión 1.4.
- Permitir crear ejecutables.

Motivo de Julia

Evitar el problema del doble lenguaje

- Uno sencillo para flexibilidad, pero lento.
- Uno complejo pero rápido.

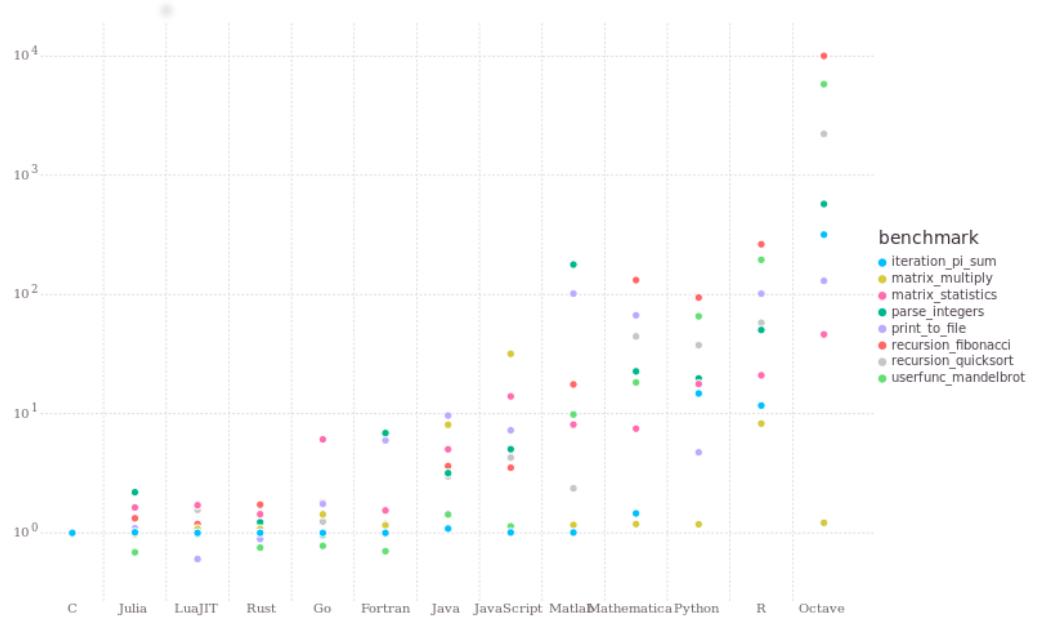


Ousterholt's dichotomy

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

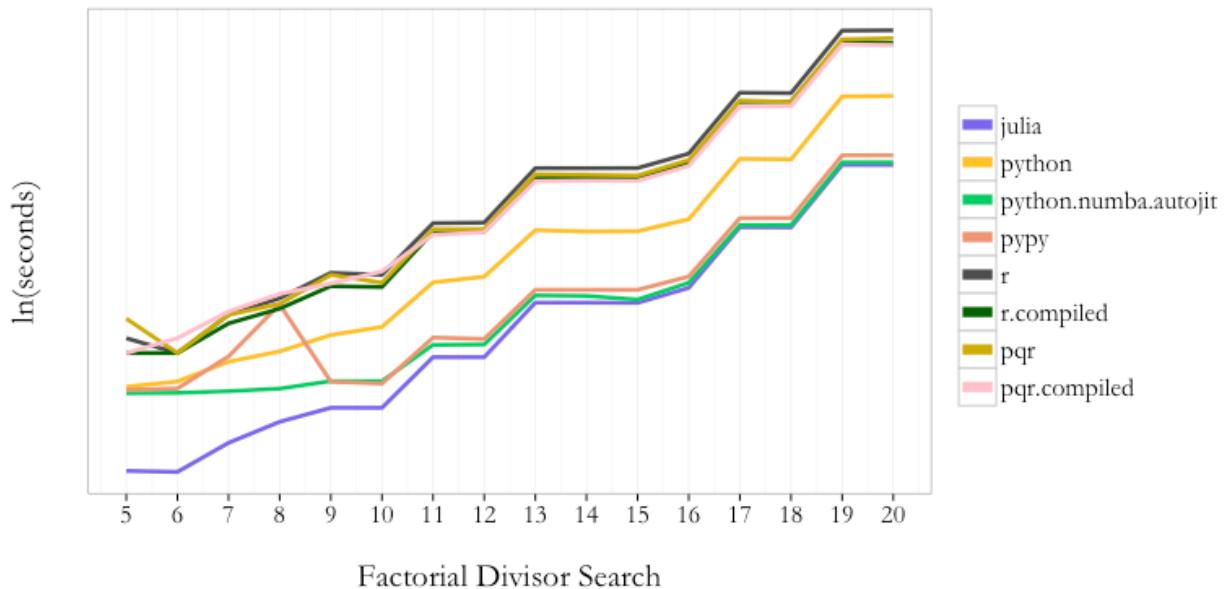
No tiene por qué ser así

Algunos Benchmarks



Resultados

Language Speed Comparison



Fuente: <https://randyzwitch.com/python-pypy-julia-r-pqr-jit-just-in-time-compiler/>
[\(https://randyzwitch.com/python-pypy-julia-r-pqr-jit-just-in-time-compiler/\)](https://randyzwitch.com/python-pypy-julia-r-pqr-jit-just-in-time-compiler/)

What makes **julia** great?

Δ Java (not concise)
✓ Python
Δ R (only R code.
not C or C++)

Clear, concise
code that can easily
be changed

When coded well, it
is very fast

✓ Java
Δ Python (Cython,etc)
Δ R (vectorized)

Great ability to mix
loop based &
matrix/vector
operations

Δ Java (not really)
✓ Python
Δ R (only vectorized)

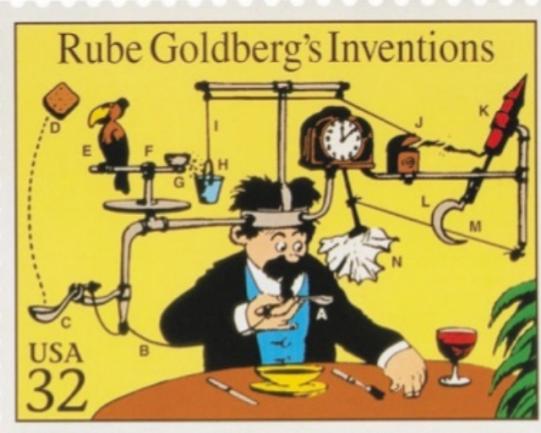
El objetivo es simplificar la vida

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





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

Mi caso es parecido

- Trabajo en Metaheurísticas en problemas complejos.
- La mayoría usa Matlab, yo Python.

Ejecutando código externo

- Pasado en Matlab, problemas de licencia.
- Ejecutar en Octave supone tardar mucho más.
- Pasarlo a Python es conflictivo:
 - Sintaxis distinta.
 - Posibles problemas: En Matlab índices desde 1, en Python por 0.
 - En Python es necesario usar numpy, sintaxis más alejada de Matlab.

Proceso de mis algoritmos

- Python para prototipado.
- Numpy para rendimiento.
- Evitar condiciones, usar operaciones vectorizadas.
- Identificar cuellos de botella.
- A veces C++ para reimplementar esas partes (Cython).

Usando Julia

- No es necesario Numpy ni cython.
- Librerías sólo disponibles en Python usando PyCall.
- Más fácil llamar código C/C++ que desde Python.

Comparando Julia con Python

Semejanzas

- Sintaxis muy similar.
- Entorno Interactivo (REPL) como IPython.

```
lobianco@Lobianco-dell-office:~/GitBook/Library/Import/juliatutorial$ julia
          _ _ _ | A fresh approach to technical computing
   _ _ _ | Documentation: http://docs.julialang.org
  / \  | Type "?help" for help.

julia> [3a + β for a in 1:3, β in 1:2]
3x2 Array{Int64,2}:
 4  5
 7  8
10 11

julia> █
```

- Modo Julia, por defecto.
- Modo Package [, para buscar y gestionar paquetes.
- Modo Shell \; para ejecutar comandos.

Vamos a verlo.

Repositorio Oficial de Paquetes

```
In [2]: using Pkg  
Pkg.add("OhMyREPL")  
  
    Updating registry at `~/.julia/registries/General`  
    Updating git-repo `https://github.com/JuliaRegistries/General.git`  
[1mFetching: [=====] 100.0 %.0 % Resolving  
g package versions...  
    Updating `~/.julia/environments/v1.2/Project.toml`  
[no changes]  
    Updating `~/.julia/environments/v1.2/Manifest.toml`  
[no changes]  
  
In [3]: using OhMyREPL  
  
hola 1  
hola 2  
hola 3  
hola 4  
hola 5  
hola 6  
hola 7  
hola 8  
hola 9  
hola 10
```

Más semejanzas con Python

- Uso de paquetes, importa con `import` o con `using`.
- Sistema de tests automáticos.

```
In [4]: import Distributions  
rand(Distributions.Uniform(-5, 5), 3)
```

```
Out[4]: 3-element Array{Float64,1}:  
-3.9072927729655538  
-4.103617780649939  
4.652310420511977
```

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

```
Out[5]: 3-element Array{Float64,1}:  
-1.7376596774291264  
2.277523569834181  
4.336852103441629
```

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

```
Out[6]: 3-element Array{Float64,1}:  
-0.5128874724794734  
1.861680261356879  
-1.7341530042303988
```

Semejanzas en la sintaxis

- No hace falta definir variables.
- Vectores son referencias.
- Manejo de iteradores (for, enumerate, zip, ...).
- Uso de funciones lambda.

```
In [4]: var = [1 2 3]
println(var)
var = 3
println(var)
filter(x->x[end]=='2', ["usuario$i" for i in 1:3])
```

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

```
Out[4]: 1-element Array{String,1}:
"usuario2"
```

Estructuras de Datos en el lenguaje

- Arrays, dentro del propio lenguaje.
- String, UTF-8.
- Diccionarios.
- Conjuntos.

```
In [8]: valores = ["hola", "adios"];
dict = Dict(i=>val for (i, val) in enumerate(valores))

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

```
2adios
1hola
```

```
In [20]: println(keys(dict))

[2, 3, 1]
```

```
In [21]: collect(Set([1, 2, 4, 2, 4]))
```



```
Out[21]: 3-element Array{Int64,1}:
 4
 2
 1
```

Diferencias en la sintaxis

- *def => function* (como Matlab).
- No usa tabulador para distinguir, usa **end**.
- Lo bueno es que no tiene begin, y el *end* es reducido (no antes de un *else*, por ejemplo).

```
In [10]: 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
0.012404 seconds (20.29 k allocations: 1.106 MiB)
```

```
Out[10]: 165580141
```

Tiposopcionales

- No es necesario, él deduce cuando se llama al método.
- Existen tipos genéricos, útiles para evitar errores (Int, Float, Real, Number, AbstractString, ...).

```
In [16]: sphere(x)= sum(x.^2)  
sphere(3)
```

```
Out[16]: 9
```

```
In [17]: sphere([3, 4, 5])
```

```
Out[17]: 50
```

Es eficiente porque se compila según el tipo concreto

```
In [11]: @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.Broadcasted{Base.Broadcast.DefaultArrayStyle{1},Nothing,typeof(Base.literal_pow),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
```

Es eficiente porque se compila según el tipo concreto

```
In [18]: fabsurda(x)=x^2+3;
```

```
In [19]: @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 [20]: @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 [15]: @code_llvm fabsurda(3)
```

```
; @ In[12]:1 within `fabsurda'
define i64 @julia fabsurda_17187(i64) {
top:
; ┌ @ intfuncs.jl:244 within `literal_pow'
; ┌ r @ int.jl:54 within `*'
;   %1 = mul i64 %0, %0
; LL
; ┌ r @ int.jl:53 within `+'
;   %2 = add i64 %1, 3
; L
;   ret i64 %2
}
```

Diferente inicio del índice

Empieza en 1, no en 0.

- No es tan problemático, se suele usar iterador y/o enumerate.

```
In [23]: valores = ["Uno", "Dos", "Tres"]  
  
for i = 1:length(valores)  
    println("$i: ", valores[i])  
end  
println("-"^8)  
for val in valores  
    println(val)  
end  
for (i, val) in enumerate(valores)  
    println("$i: $val")  
end
```

```
1: Uno  
2: Dos  
3: Tres  
-----  
Uno  
Dos  
Tres  
1: Uno  
2: Dos  
3: Tres
```

Modelo Funcional, no Orientado a Objetos

- Los paquetes definen estructuras y funciones sobre ellas.
- Usando jerarquías de tipos se puede reutilizar código.
- El Multiple Dispatch (redefinir métodos).
 - Interfaz Table usado por múltiples implementaciones (DataFrames, ...).
- Interfaz funcional.
- Operador |> Permite encadenar operaciones.

In [17]:

```
msg = "Hola"  
println(uppercase(msg))  
msg |> uppercase |> println
```

```
HOLA  
HOLA
```

```
In [14]: struct MyData
    n::Int64
    s::String
end

using Base: println

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

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

```
Mi println: 4: Hola
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[14]:9
[3] println(io::IO, xs...) in Base at strings/io.jl:75
[4] println(xs...) in Base at coreio.jl:4
methodswith(typeof(data)) = Method[println(io::IO, v::MyData)] in Main at In[14]:9
```

Out[14]: 1-element Array{Method,1}: • println(io::IO, v::MyData) in Main at In[14]:9

Operador Punto

- Permite vectorizar cualquier función.
- Permite operaciones más eficientes.

```
In [5]: 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]
```

Ejemplo más complejo

```
In [80]: A = rand(Uniform(-5, 5), 3000); B = rand(Uniform(-5, 5), 3000);
```

```
In [81]: C = similar(A);
function add1!(C, A, B)
    C .= A .+ B
end
@btime add1!(C, A, B);
```

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

```
In [82]: add2!(C, A, B) = @. C = A+B
@btime add2!(C, A, B);
```

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

```
In [83]: 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)
```

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

Uso de ficheros

```
In [22]: users = ["Estudiante$(elem)" for elem in 1:2:8]
```

```
Out[22]: 4-element Array{String,1}:
  "Estudiante1"
  "Estudiante3"
  "Estudiante5"
  "Estudiante7"
```

```
In [23]: open("usuarios.txt", "w") do file
    for user in users
        println(file, user)
    end
end
```

```
In [24]: open("usuarios.txt") do file
    for line in readlines(file)
        println(strip(line))
    end
end
```

```
Estudiante1
Estudiante3
Estudiante5
Estudiante7
```

Comunicación con otros lenguajes

- Comunicación con Python/R/C++.

```
// Código en libsaludo.so
#include <stdio.h>

char msg[80];

char *getline(int num) {
    sprintf(msg, "Hola a los %d asistentes", num);
    return msg;
}
```

Llamando al código C/C++

Se compila como librería .so, y luego se puede llamar desde Julia.

```
In [28]: using Libdl  
lib_ptr = dlopen("./libsaludo.so")  
get_name = dlsym(lib_ptr, "getline")  
result = ccall(get_name, Cstring, (Cint, ), 16)  
println(unsafe_string(result))
```

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

Llamando a código Python

```
In [29]: using PyCall  
math=pyimport("math")  
println(math.sin(3)) # Llama a función sin de math  
np=pyimport("numpy")  
np.random.rand(5) # Convierte al tipo array de Julia
```

0.1411200080598672

```
Out[29]: 5-element Array{Float64,1}:  
0.6645426015215873  
0.48156087221872945  
0.48267320440048245  
0.2579092002120934  
0.516599210934333
```

Modo de Ayuda

Uso de cadenas descriptivas para cada función.

In [25]:

```
"""
Esta función permite calculo el número de fibonacci.

n: Entero del cual calcular fibonacci.

"""

function fibo(n::Int)
    (a, b) = (1, 1)

    for i in 1:n-1
        (a, b) = (a+b, a)
    end
    a
end

@time @show fibo(40)
```

```
fibo(40) = 165580141
0.015480 seconds (30.12 k allocations: 1.558 MiB)
```

Out[25]: 165580141

In [26]: ?fibo

```
search: fibo failprob unsafe_pointer_to_objref
```

Out[26]:

Esta función permite calculo el número de fibonacci.

n: Entero del cual calcular fibonacci.

Flexible

Se puede personalizar

```
In [30]: "a" in "holá"  
use occursin(x, y) for string containment  
  
Stacktrace:  
[1] error(::String) at ./error.jl:33  
[2] in(::String, ::String) at ./strings/search.jl:455  
[3] top-level scope at In[30]:1  
  
In [31]: using Base: in  
  
Base.in(a::AbstractString, b::AbstractString) = occursin(a, b)  
  
"a" in "holá"  
  
Out[31]: true
```

Macros

- Empiezan por @.
- Permiten generar código.

```
In [16]: macro load(x) # using a veces tarda, creo una macro para sólo hacerlo si no est  
á ya en memoria  
    if !(x in names(Main, all=false, imported=true))  
        return :(using $x)  
    end  
end  
  
@time using OhMyREPL  
@time @load OhMyREPL
```

```
0.000317 seconds (634 allocations: 32.906 KiB)  
0.000001 seconds (4 allocations: 160 bytes)
```

Se puede construir fácilmente la documentación, Paquete [Documenter.jl](#) (<https://juliadocs.github.io/Documenter.jl/stable/>):

- Usa Markdown para describirlo para los usuarios.
- Permite añadir la documentación del API.
- Tests como documentación.

Comunidad Científica

Librerías/Paquetes populares en: <https://pkg.julialang.org/docs/> (<https://pkg.julialang.org/docs/>)

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

Comunidad científica

Librerías del Estado del Arte

- Librería de optimización: [JuMP \(<https://pkg.julialang.org/docs/JuMP/DmXqY/0.20.1/quickstart.html#Quick-Start-Guide-1>\)](https://pkg.julialang.org/docs/JuMP/DmXqY/0.20.1/quickstart.html#Quick-Start-Guide-1)
- Ecuaciones Diferenciales: [DifferentialEquations \(<https://docs.juliadiffeq.org/latest/>\)](https://docs.juliadiffeq.org/latest/)
- Librerías estadística: [Distributions \(<https://juliastats.org/Distributions.jl/latest/startng/>\)](https://juliastats.org/Distributions.jl/latest/startng/)

Más genéricas

- Páginas web: [Genie \(<https://genieframework.github.io/Genie.jl/>\)](https://genieframework.github.io/Genie.jl/)
- Base de Datos: [Octo \(<https://github.com/wookay/Octo.jl>\)](https://github.com/wookay/Octo.jl)

IDEs

- Juno: Atom con los paquetes de Julia pre-instalados.
- Visual Code.
- SublimeText.
- Emacs.

Juno

The screenshot displays the Juno IDE interface, which integrates several tools for Julia development:

- Project View:** Shows a tree of Julia packages and files.
- Code Editor:** An untitled file containing Julia code for performing a multidimensional type-II discrete cosine transform (DCT) using FFTW. The code includes imports for FFTW, defines a function `profile_test`, and uses various Julia and FFTW functions like `randn`, `maximum`, `mapslices`, and `dct`.
- Terminal:** Shows the output of running the code in the Julia REPL, including the result of `sum(ans)`.
- Workspace:** Displays variables `a` (a `Float64[5]`) and `ans` (3.75...).
- Plots:** A line plot titled "Timing" showing performance metrics over time.
- Profiler:** A separate window showing call graphs and performance data.
- Bottom Status Bar:** Shows file paths, line numbers, and other status information.

Visual Code

The screenshot shows a Visual Studio Code interface with the following components:

- Project Explorer:** Shows a tree view of files and folders, including "pharma_1.png", "Statistics of Complexity.jl", "JULIA", "INPUT", "Mamba", "StochMCMC", ".DS_Store", "BAYES-SGHMC Notebooks.ipynb", "BAYES ARDL.jl", "BAYES GROWTH AR.jl", "BAYESIAN PHL LEI.jl", "BAYESIAN ARDL.ipynb", "BAYESIAN GROWTH.ipynb", "BAYESIAN GROWTH.jl", "BAYESIAN GROWTHAR.ipynb", "Bayesian Linear Reg.ipynb", "CHAPTER 2.jl", "COMPARE DIFFERENT.ipynb", "COMPARE DIFFERENTAR.ipynb", "COMPARE DIFFERENTAR.ipynb", "Data for LEI Worksh.ipynb", "DIAGNOSTICS MCMC.ipynb", "lei_data", "Diagnostics.jl", "Gibbs.jl", "HMC.jl", "Iris.png", "MCMC Diagnostics.jl", "MCMC Methods.ipynb", and "plots.jl".
- Code Editor:** Displays a Julia script with code related to BAYES-SGHMC Notebooks, including imports for DataFrames, Distributions, Gadfly, and Gadfly.push_theme(:dark). It also includes code for reading data from CSV files and plotting.
- Terminal:** Shows the output of the script execution, including variable definitions and data frames.
- Output:** Shows the results of the plot command, displaying a time series plot of "lei_data" over time.
- File Explorer:** Shows the file system structure of the workspace.
- Search:** Shows search results for terms like "ans", "in_dir", "lei_data", "lei_df", "lei_growth", "ou_Rdir", "n", "i", "r", "c", "t", "HMC", "MH", "SGHMC", "ans", "in_dir", "lei_data", "lei_df", "lei_growth", "ou_Rdir", "r", "c", "t", "HMC", "MH", "SGHMC".
- Status Bar:** Shows the date and time (Main June 11th, Sunday, 2:34 pm) and update count (4 updates).

Otras ventajas de Julia

- Notación matemática.

```
In [1]: fun_mat(x; α=0.5)=3α+2x
```

```
fun_mat(3, α=0.1)
```

```
Out[1]: 6.3
```

- Más fácil de general paquete oficial que pip.
 - Creado desde pkg, recomendable usar PkgTemplate.jl
- Criterios razonables, los métodos que cambian atributos terminan con "!".

Consejos sobre Julia

- Cargar la librería es costoso.
 - Cargar una vez, y luego cargar con include las funciones.
 - Usar paquete Revise que permite recargar los ficheros modificados.
- Que las variables no cambien de tipo.
- No usar [], si no Tipo[].

In [11]:

```
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}
```

Consejos sobre Julia

- Leer documentación, adaptarse al lenguaje.
- Documentar funciones propias, y ser claro.
- Usar parámetros sólo por seguridad (tipos genéticos), no abusar.
- Usar paquetes, dependencias claras.
- Evita variables globales, sólo constantes (palabra **const**).
- Usar funciones, incluso una main().

Desventajas de Julia

- Gran latencia al compilar (como Tiempo para cargar librerías, y mostrar la primera gráfica).
- Falta de madurez en librerías (falta documentación).
- Falta compilación estática.
- Muy enfocado al entorno científico, faltan librerías genéricas.
- Union de tipos costoso.
- Malos mensajes de error.



Muchas gracias por la atención

danimolina@gmail.com

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

normal