Seminario de Informática

Lenguaje Julia y Conjuntos de Julia

L. J. Hernández Paricio

Contents

1 Introducción **4**

1.1 Página web 4
1.2 Algunas características 5
1.3 Rapidez de ejecución 6
1.4 Primeros creadores y expansión del lenguaje 9
1.5 Otras entidades relacionadas con Julia 14
1.6 Alojado en GitHug 21
1.7 Instalación y documentación 22

2 IDE (Entornos de desarrollo integrados) y Editores **23**

2.1 Atom: Entorno de Desarrollo para varios Lenguajes 24
2.2 Jupyter 26

3 Multiple Dispatch (Tipos, Métodos y Funciones) **29**

3.1 Tipos 29
3.2 Métodos 34
3.3 Múltiple dispatch 36
3.4 Funciones 37
3.4.1 Cómo declarar una función 38
3.4.2 Tipo de argumentos admitidos 40
3.4.3 Difusión (Broadcasting) 42

3.4.4 Añadiendo un nuevo método 44

4 Extensión del Lenguaje Julia mediante paquetes **45**
4.1 Administrador de paquetes 45
4.2 Programación en paralelo y distribuida 48
4.3 Relaciones con otros lenguajes 51
4.3.1 Relaciones con Python: Paquete PyCall.jl (Páll Haraldsson+ 133 contributions) 51
4.3.2 SymPy.jl (john verzani) 51
4.4 Julia and Big Data 53

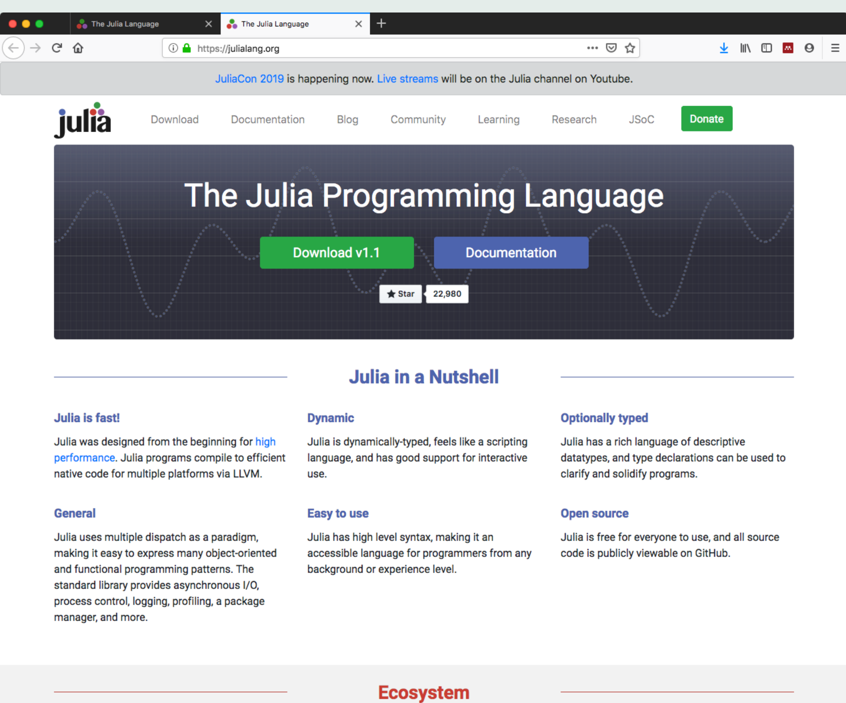
5 Un paquete para visualizar conjuntos de Julia **55**

1. Introducción

Julia: **Parece como Python, siente como Lisp y se ejecuta como Fortran**

1.1. Página web

https://julialang.org/



1.2. Algunas características

- Rapidez: Julia fue diseñado desde el principio para tener **alto rendimiento y rápida ejecución**.
- Generales: Los diseñadores dicen que el lenguaje utliliza **multiple dispacth** como paradigma y ello permite expresar patrones de la programación orientada a objetos o de la programación funcional.
- Información sobre la ejecución: **logging** (cuenta lo que ha sucedido), **profiles** (cuantifica el modo que se ha ejecutado el código).
- Técnicas: Sobresale en **cálculo numérico**. **Sintaxis excelente para las matemáticas**. Amplio soporte para **muchos tipos de datos** (opcionalmente tipado, tipos dinámicos). Paquetes para la programación en **paralelo y distribuida**.

1.3. Rapidez de ejecución

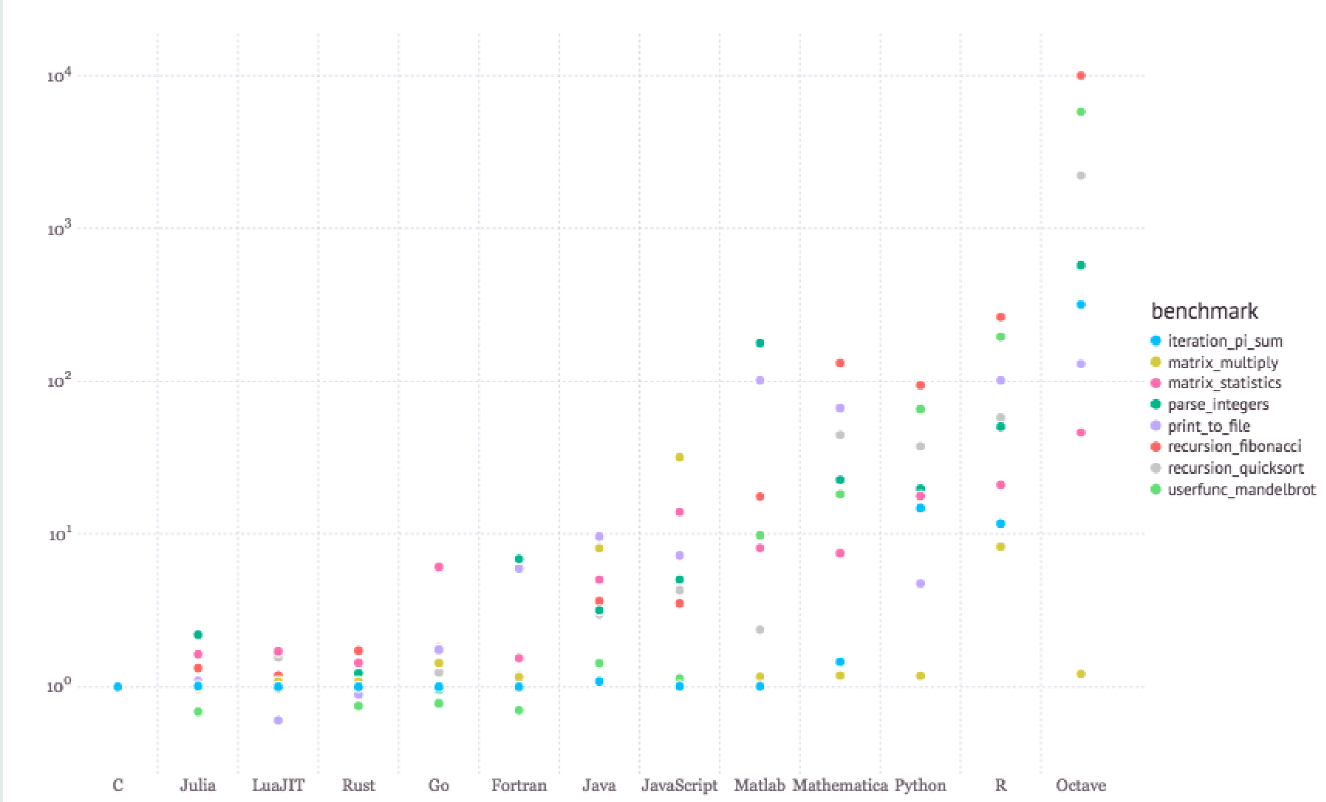
Para comparar el rendimiento se utilizan algoritmos idénticos y patrones de código implementados en diversos lenguajes.

Lenguajes: con **Julia** v1.0.0, **SciLua** v1.0.0-b12, **Rust** 1.27.0, **Go** 1.9, **Java** 1.8.0_17,**Javascript** V8 6.2.414.54, **Matlab** R2018a, Anaconda **Python** 3.6.3, **R** 3.5.0, and **Octave** 4.2.2.

Algoritmos: Iteration_π_sum, mutliplication_matrix, ...

Gráfico de la página web https://julialang.org/benchmarks/

Los tiempos calculados se comparan con los del el lenguaje C que mediante este criterio se le asigna el valor $10^0 = 1$



10^4
10^3
10^2
10^1
10^0
C
Julia
LuaJIT
Rust
Go
Fortran
Java
JavaScript
Matlab
Mathematica
Python
R
Octave
benchmark
iteration_pi_sum
matrix_multiply
matrix_statistics
parse_integers
print_to_file
recursion_fibonacci
recursion_quicksort
userfunc_mandelbrot

Los programas escritos en Julia son muy rápidos debido a su compilador **JIT** (Just-In-Time) basado en **LLVM** (Low Level Virtual Machine.

Está diseñado para **computación en la nube** y dispone de diversos paquetes para la **programación distribuida y en paralelo**.

Admite varios estilos de paralelismo y permite a los usuarios el diseño de otros estilos nuevos.

1.4. Primeros creadores y expansión del lenguaje

Hay una serie de entidades que rodean el lenguaje de programación Julia.

El **proyecto de Julia** fue fundado por **Jeff Bezanson, Alan Edelman, Viral Shah y Stefan Karpinski**.

El proyecto consiste en un código y una comunidad de personas que trabajan en ese código.

Actualmente hay **67 investigadores que tienen acceso comprometido a la organización JuliaLang.**

Otros muchos desarrolladores que son contribuyentes prolíficos al ecosistema de Julia pero que no tienen bit de compromiso.

La naturaleza comunitaria del código abierto hace que sea difícil definir con precisión dónde finaliza y comienza la comunidad en general.

El lenguaje fue creado por **Stefan Karpinski**, estudiante graduado de la Universidad de California. **Estaba involucrado en el desarrollo de una herramienta simulación de redes que requería el uso de lenguajes de programación diferentes. Ninguno de los lenguajes usados podía completar todo el proceso**. Por ello, Karpinski, junto con su compañero de universidad Viral Shah y Jeff Bezanson del MIT, decidieron resolver el problema diseñando un nuevo lenguaje que fuera compatible con las diferentes tareas.

Stefan Karpinski

Viral B Shah es un Ingeniero informático Indio graduado en "Padmabhushan Vasantdada Patil Pratishthan's College of Engineering" y doctor por la Universidad Santa Bárbara de California. **Sigue siendo una de los desarrolladores mas activo del lenguaje Julia.**

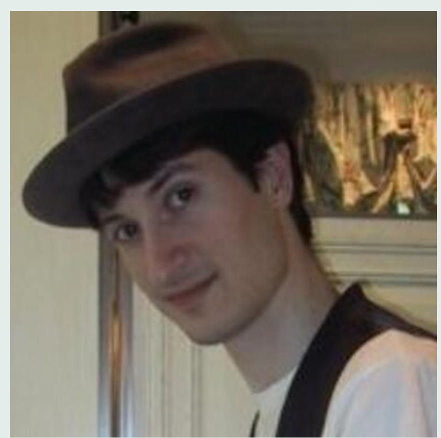
Está involucrado en un proyecto del Gobierno India para dar una identidad única a residentes en la India basado en datos demográficos y biométricos.

Viral B Shah

Jeff Bezanson obtuvo su graduado en informática en la Universidad de Harvard (2000-2004). Realizo su máster (septembre 2010-2012). **Defendió su tesis (junio 2015) en el Massachusetts Institute of Technology dirigida por Alan Edelman**.

Es uno de los fundadores del projecto Julia (finananciado en el MIT 2009).

También es cofundador del la empresa Julia Computing, Inc que expande Julia hacia objetivos comerciales.



Jeff Bezanson

Alan Edelman fue alumno del "Hampshire College Summer Studies in Mathematics", Edelman obtuvo el B.S. y M.S. grados en Matemáticas en la Universidad de Yale en 1984 y el Ph.D. en matemática aplicada en el MIT en 1989. Edelman estuvo en U.C. Berkeley de 1990-93 y **se incorporó a la Facultad de Matemáticas de MIT en 1993**.

Alan Edelman

1.5. Otras entidades relacionadas con Julia

Julia Stewards (Revisores)

Existe un grupo oficial de personas que representan al proyecto Julia: Julia Stewards.

Este grupo existe para manejar la resolución de conflictos y los informes de comportamiento inapropiado o problemático en la comunidad Julia.

Los administradores actuales son: Milan Bouchet-Valat, Simon Byrne, Tim Holy, Katharine Hyatt, Steven Johnson, Stefan Karpinski y Viral Shah.

El laboratorio de julia

Gran parte del trabajo inicial sobre el desarrollo del núcleo Julia se realizó en el MIT en lo que ahora se conoce como The Julia Lab bajo la dirección del profesor Alan Edelman.

Todos los co-creadores de Julia han sido parte del laboratorio en algún momento.

Jeremy Kepner fundador del Lincoln Laboratory Supercomputing Center (MIT) fue uno de los primeros financiadores del trabajo del laboratorio en Julia y continúa apoyando este trabajo.

El Laboratorio Julia sigue siendo una fuente constante de importantes innovaciones y contribuciones para Julia.

Otros grupos de investigación en el MIT:

Steven Johnson, profesor en MIT, encabeza su propio grupo, pero en su tiempo libre se ha convertido en uno de los contribuyentes más prolíficos de Julia (actualmente, el número 10).

Juan Pablo Vielma profesor del MIT y su grupo desarrollan los proyectos JuMP y Cassette.

Hay otros grupos de investigación en el MIT que utilizan a Julia en estos días, haciendo importantes contribuciones a su ecosistema y a la ciencia.

Otras universidades e instinutos de todo el mundo están utilizando y desarrolando Julia:

Caltech, Stanford, UC Berkeley, Harvard, Columbia, NYU, Oxford, NUS, UCL, Nantes, Alan Turing Institute, University of Chicago, Cornell, Max Planck Institute, Australian National University, University of Warwick, University of Colorado, Queen Mary University of London, London Institute of Cancer Research, UC Irvine, University of Kaiserslautern.

Los usuarios y socios de Julia incluyen:

Amazon, IBM, Intel, Microsoft, DARPA, Lawrence Berkeley National Laboratory, National Energy Research Scientific Computing Center (NERSC), Federal Aviation Administration (FAA), MIT Lincoln Labs, Moore Foundation, Nobel Laureate Thomas J. Sargent, Federal Reserve Bank of New York (FRBNY), Capital One, Brazilian National Development Bank (BNDES), BlackRock, Conning, Berkery Noyes, BestX, Path BioAnalytics, Invenia, AOT Energy, AlgoCircle, Trinity Health, Gambit, Augmedics, Tangent Works, Voxel8, UC Berkeley Autonomous Race Car (BARC) y muchos de los bancos de inversión, gestores de activos, gestores de fondos, analistas de divisas, aseguradoras, fondos de cobertura y reguladores.

Julia Computing:

Es una empresa que crea productos basados en Julia. Los fundadores de esta empresa incluyen a parte de los creadores, algunos colaboradores y otros especialistas en modelos de negocio.
https://juliacomputing.com/

Ofrece los entre otros los siguientes productos: JuliaSure, JuliaTeam, JuliaRun, JuliaAcademy

Ranking de popularidad de Julia:

TIOBE Index

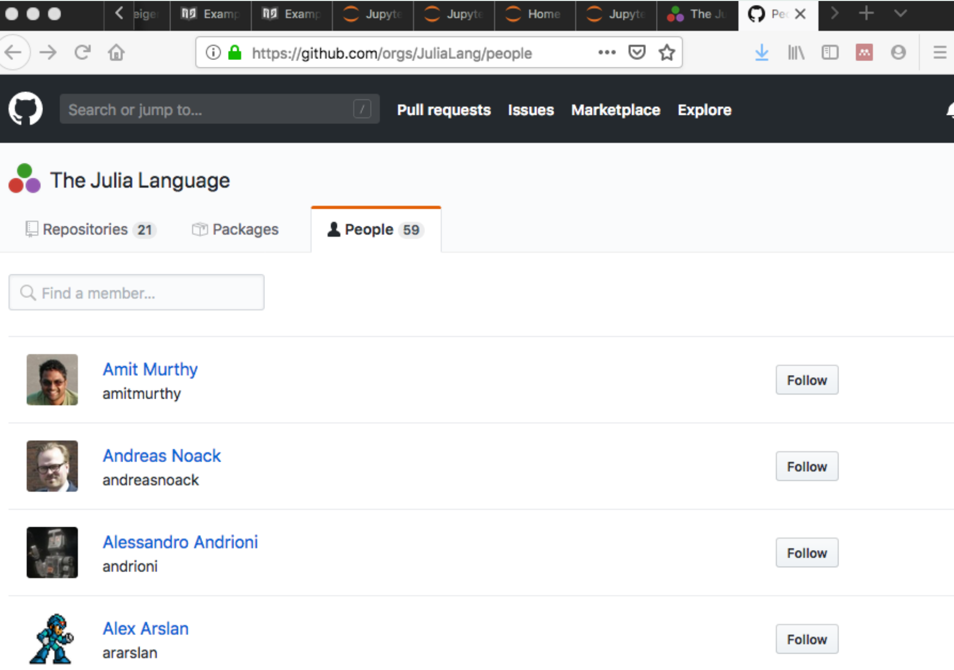
Nov 2019	Nov 2018	Change	Programming language	Ratings	Change
1	1		Java	16.246%	-0.50%
2	2		C	16.037%	+1.64%
3	4	↑	Python	9.842%	+2.16%
4	3	↓	C++	5.605%	-2.68%
5	6	↑	C#	4.316%	+0.36%
6	5	↓	Visual Basic .NET	4.229%	-2.26%
7	7		JavaScript	1.929%	-0.73%
8	8		PHP	1.720%	-0.66%
9	9		SQL	1.690%	-0.15%
10	12	↑	Swift	1.653%	+0.20%
11	16	↑↑	Ruby	1.261%	+0.17%
12	11	↓	Objective-C	1.195%	-0.28%
13	13		Delphi/Object Pascal	1.142%	-0.28%
14	25	↑↑	Groovy	1.099%	+0.50%
15	15		Assembly language	1.022%	-0.09%
16	14	↓	R	0.980%	-0.43%
17	20	↑	Visual Basic	0.957%	+0.10%
18	23	↑↑	D	0.927%	+0.25%
19	17	↓	MATLAB	0.890%	-0.14%
20	10	↓↓	Go	0.853%	-0.64%

PYPL Index (Worldwide)

Nov 2019	Change	Programming language	Share	Trends
1		Python	29.62 %	+4.2 %
2		Java	19.52 %	-2.2 %
3		Javascript	8.43 %	+0.2 %
4		C#	7.27 %	-0.4 %
5		PHP	6.39 %	-1.0 %
6		C/C++	5.89 %	-0.3 %
7		R	3.76 %	-0.2 %
8		Objective-C	2.55 %	-0.7 %
9		Swift	2.44 %	-0.2 %
10	↑↑	TypeScript	1.85 %	+0.3 %
11	↓	Matlab	1.82 %	-0.2 %
12	↑↑↑↑	Kotlin	1.63 %	+0.5 %
13		VBA	1.41 %	-0.1 %
14	↓↓↓	Ruby	1.39 %	-0.2 %
15	↑↑	Go	1.24 %	+0.3 %
16	↓↓	Scala	1.14 %	-0.1 %
17	↓↓	Visual Basic	0.98 %	-0.2 %
18	↑↑	Rust	0.62 %	+0.2 %
19	↓	Perl	0.55 %	-0.1 %
20	↑↑↑↑	Dart	0.37 %	+0.2 %
21	↓↓	Lua	0.34 %	-0.0 %
22	↓	Haskell	0.3 %	+0.0 %
23	↓	Julia	0.26 %	+0.0 %
24	↓	Delphi	0.24 %	-0.0 %

1.6. Alojado en GitHug

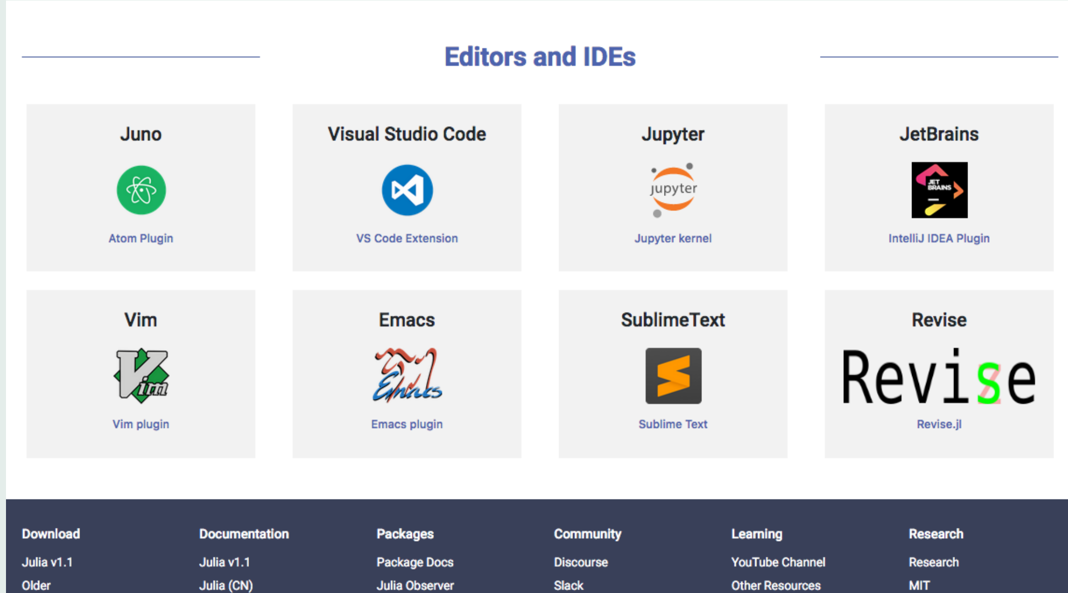
GitHub es un sistema de gestión de proyectos y control de versiones de código, así como una plataforma de red social diseñada para desarrolladores.
GitHub es también uno de los repositorios online más grandes de todo el mundo.



1.7. Instalación y documentación

Instalación de Julia en https://julialang.org/downloads/
Hay versiones para Windows, Mac Os y Linux.
Documentación
https://docs.julialang.org/en/v1/
Para aprender Julia
https://julialang.org/learning/

2. IDE (Entornos de desarrollo integrados) y Editores

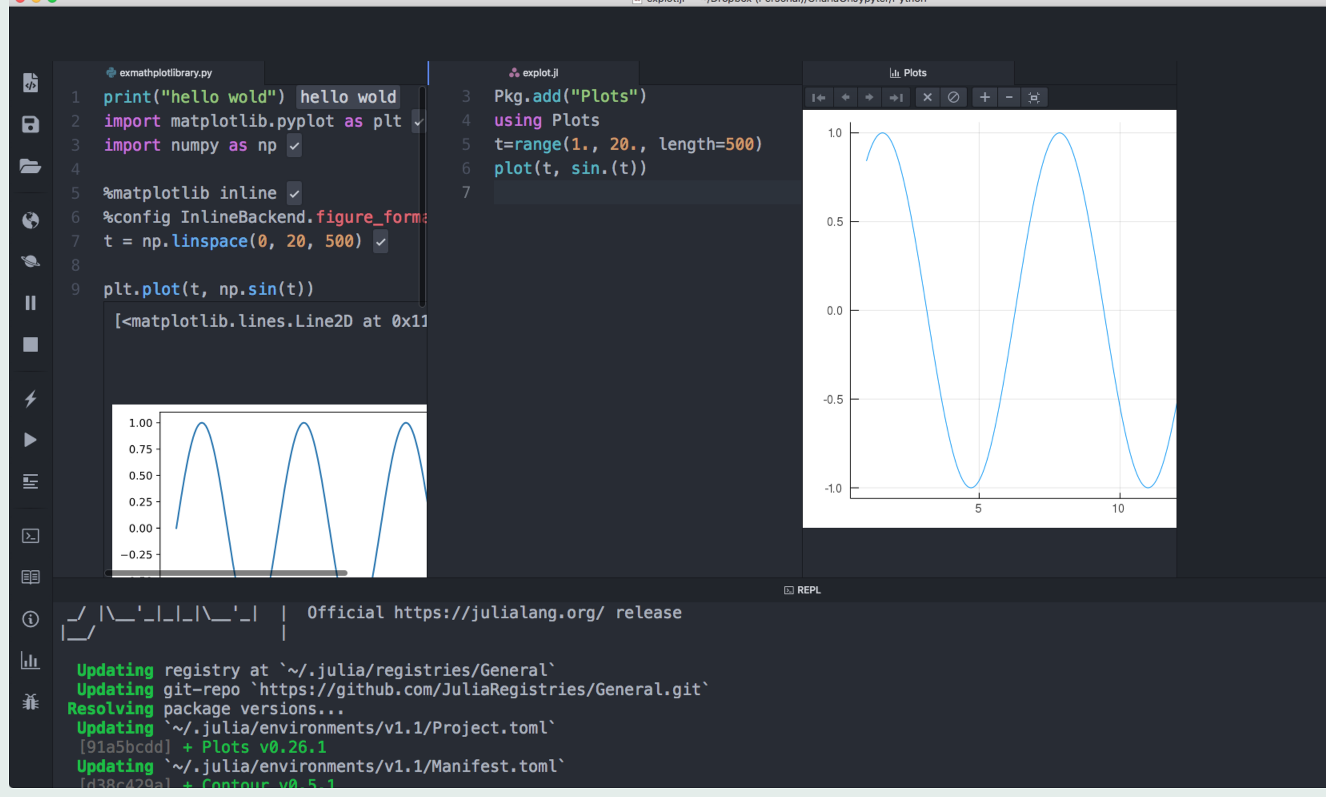


2.1. Atom: Entorno de Desarrollo para varios Lenguajes

Un entorno de desarrollo integrado (IDE) puede obtenerse con el editor Atom que se puede descargar en
https://atom.io
Atom dispone de numerosos paquetes que permiten ejecutar núcleos (kernels) de diversos lenguajes: Julia, Python, R, etc

- **Juno** (paquete de Atom que **permite ejecutar Julia**)
- **Hydrogen** (paquete de Atom que **permite ejecutar Julia, Phyton, R, etc**)

Python: Anaconda es una distribución de Python que además incorpora otras herramientas.
Instalación: https://www.anaconda.com/download


explot.jl — ~/Dropbox (Personal)/CharlaOnJypyter/Python
exmathplotlibrary.py
print("hello wold") hello wold
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
%config InlineBackend.figure_forma
t = np.linspace(0, 20, 500)
plt.plot(t, np.sin(t))
[<matplotlib.lines.Line2D at 0x11
1.00
0.75
0.50
0.25
0.00
−0.25
explot.jl
Pkg.add("Plots")
using Plots
t=range(1., 20., length=500)
plot(t, sin.(t))
Plots
1.0
0.5
0.0
-0.5
-1.0
5
10
REPL
Official https://julialang.org/ release
Updating registry at `~/.julia/registries/General`
Updating git-repo `https://github.com/JuliaRegistries/General.git`
Resolving package versions...
Updating `~/.julia/environments/v1.1/Project.toml`
[91a5bcdd] + Plots v0.26.1
Updating `~/.julia/environments/v1.1/Manifest.toml`

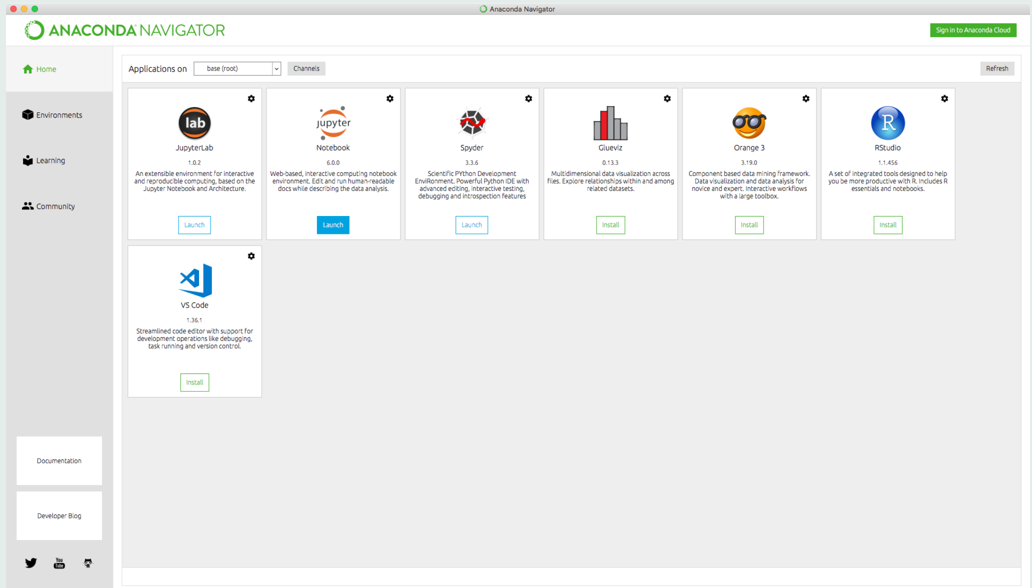
2.2. Jupyter

Jupyter es un proyecto que produce entornos interactivos basados en navegadores web para programación, matemáticas y ciencia de datos.

Es compatible con varios lenguajes a través como Python, Ruby, Haskell, R, Scala y Julia.

Jupyter Notebook es la aplicación tradicional y la más estable. **JupyterLab** tiene una nueva interfaz y es más adecuado para trabajar con proyectos más grandes que constan de varios archivos.

Anaconda instala también la aplicación Anaconda-Navigator. Al abrir la aplicación Anaconda-Navigator el navegador abre una página desde la cual se pueden ejecutar diversas versiones de notebooks:



En un notebook de Jupypter se pueden considerar varias celdas de manera que cada una de ellas se puede ejecutar con un kernel diferente..

Por ejemplo, una se puede ejecutar con el Kernel Julia y la siguiente con el Kernel Python.

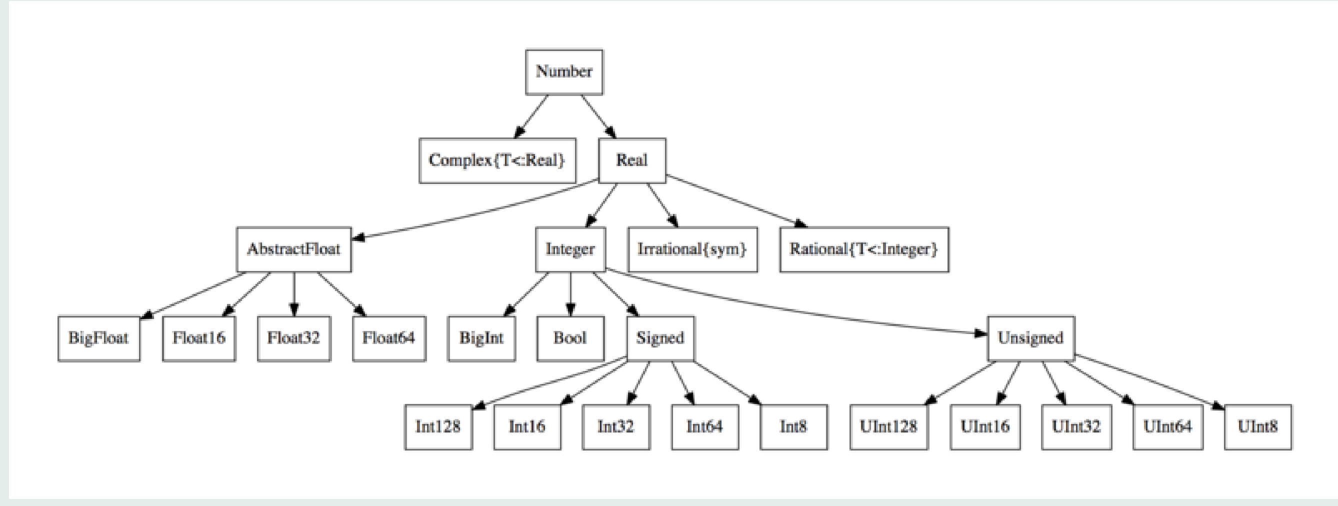
3. Multiple Dispatch (Tipos, Métodos y Funciones)

Una sesión básica (de programación o de ejecución) del lenguaje Julia se organiza mediante funciones a cuyos argumentos se les puede asignar tipos concretos o tipos abstractos.

3.1. Tipos

Julia utiliza variables que admiten valores a los que se les puede asignar un tipo.

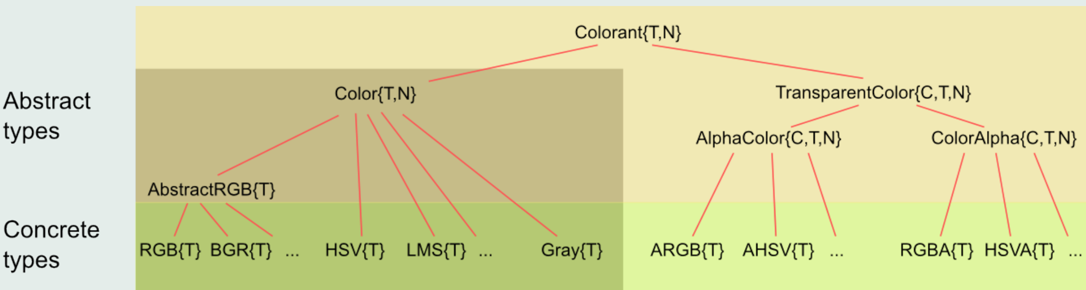
Estructura jerárquica de los algunos tipos de números:



Tipos abstractos y tipos concretos

Julia dispone de contructores de tipos (struct)

Estructura jerárquica de los algunos tipos de colores (Paquete ColorTypes):



A una variable de podemos asignar un valor

[1]:
```
a=2
```

[2]:
```
typeof(a)
```

[2]:
```
Int64
```

[3]:
```
b=1.2
```

[3]:
```
1.2
```

[4]:
```
typeof(b)
```

[4]:
```
Float64
```

[5]:
```
c=2+3.4*im
```

[5]:
```
2.0 + 3.4im
```

[6]:
```
typeof(c)
```

[6]:
```
Complex{Float64}
```

Julia es un lenguaje de programación con **tipos dinámicos**.

El compilador debe conocer el tipo de cada valor en **tiempo de ejecución** para decidir qué **método** aplicar a ese valor.

Ser un lenguaje de tipo dinámico significa que dicho conocimiento puede ser **explícito** (es decir, declarado por el usuario) o **implícito** (es decir, deducido por Julia con un motor de inferencia de tipo inteligente del contexto en el que se utiliza).

La escritura dinámica hace que el desarrollo de código con Julia sea flexible y rápido

3.2. Métodos

Las funciones en Julia analizan el tipo de valores pasados como argumentos y deciden cómo podemos operar con los valores.

Sumar 1 + 2 (dos enteros) será diferente de sumar 1.0 + 2.0 (dos flotantes) porque **el método para sumar dos enteros es diferente del método para sumar dos flotantes**.

En la implementacián base de Julia, hay muchos métodos diferentes para la función suma.

Se pueden enumerar con el comando methods():

[1]:
```
methods(+)
```

[1]:
```
# 163 methods for generic function "+":
[1] +(x::Bool, z::Complex{Bool}) in Base at complex.jl:277
[2] +(x::Bool, y::Bool) in Base at bool.jl:104
[3] +(x::Bool) in Base at bool.jl:101
[4] +(x::Bool, y::T) where T<:AbstractFloat in Base at bool.jl:112
[5] +(x::Bool, z::Complex) in Base at complex.jl:284
[6] +(a::Float16, b::Float16) in Base at float.jl:392
[7] +(x::Float32, y::Float32) in Base at float.jl:394
[8] +(x::Float64, y::Float64) in Base at float.jl:395
[9] +(z::Complex{Bool}, x::Bool) in Base at complex.jl:278
[10] +(z::Complex{Bool}, x::Real) in Base at complex.jl:292
[11] +(::Missing, ::Missing) in Base at missing.jl:96
[12] +(::Missing) in Base at missing.jl:83
[13] +(::Missing, ::Number) in Base at missing.jl:97

[162] +(J::LinearAlgebra.UniformScaling, A::AbstractArray{T,2} where T) in
LinearAlgebra at /Users/julia/buildbot/worker/package_macos64/build/usr/
 ↪share/ju
lia/stdlib/v1.1/LinearAlgebra/src/uniformscaling.jl:92
[163] +(a, b, c, xs...) in Base at operators.jl:502
```

3.3. Múltiple dispatch

Múltiple dispatch permite expresar muchos patrones de la programación orientada a objetos o a programación funcional.

Múltiple dispatch permite asociar con el mismo nombre de una función distintos métodos. Cada uno de ellos puede utilizar en sus argumentos diferentes tipos de datos. Se pueden añadir nuevos métodos con nuevos tipos de datos. También se puede definir funciones sin especificar los tipos.

Múltiple dispatch es diferente de la sobrecarga de operadores existentes en lenguajes como C ++. **Múltiple dispatch determina el método concreto en tiempo de ejecución, no en tiempo de compilación**.

En Julia, los métodos no están definidos dentro de las clases como se haría en la mayoría de los lenguajes orientados a objetos.

3.4. Funciones

1. Cómo declarar una función
2. Tipo de argumentos admitidos
3. Difusión (Broadcasting)
4. Añadiendo un método

3.4.1. Cómo declarar una función

Julia nos permite definir una función de varias maneras. La primera requiere de las palabras reservadas `function` y `end`

[1]:
```
function Hola(nombre)
    println("Hola $nombre, ¡me alegro de verte!")
end
```

[1]:
```
Hola (generic function with 1 method)
```

[2]:
```
function f(x)
    x^2
end
```

[2]:
```
f (generic function with 1 method)
```

Y las podemos llamar así

[3]:
```
Hola("C-3PO")
```

```
Hola C-3PO, ¡me alegro de verte!
```

[4]:
```
f(42)
```

[4]:
```
1764
```

Alternativamente, las podemos declarar en una sóla línea

[5]:
```
Hola2(nombre) = println("Hola $nombre, ¡me alegro de verte!")
```

[5]: Hola2 (generic function with 1 method)

[6]:
```
f2(x) = x^2
```

[6]: f2 (generic function with 1 method)

[7]:
```
Hola2("R2D2")
```

```
Hola R2D2, ¡me alegro de verte!
```

[8]:
```
f2(42)
```

[8]: 1764

Finalmente, pudimos declararlas como funciones "anónimas"

[9]:
```
Hola3 = nombre -> println("Hola $nombre, ¡me alegro de verte!")
```

[9]: #3 (generic function with 1 method)

[10]:
```
f3 = x -> x^2
```

[10]: #5 (generic function with 1 method)

3.4.2. Tipo de argumentos admitidos

En Julia, las funciones operarán con cualquier valor que tenga sentido.

[13]:
```
Hola(55595472)
```

```
Hola 55595472, ¡me alegro de verte!
```

Y `f` va a funcionar en una matriz.

[14]:
```
A = rand(3, 3)
A
```

[14]:
```
3×3 Array{Float64,2}:
 0.98379   0.466573  0.151611
 0.751848  0.45485   0.396506
 0.309114  0.592121  0.606748
```

[15]:
```
f(A)
```

[15]:
```
3×3 Array{Float64,2}:
 1.3655    0.761002  0.426141
 1.2042    0.79246   0.534919
 0.936842  0.772818  0.649788
```

`f` funcionará con "Hola" porque `*` para inputs de cadenas como concatenación.

[16]:
```
f("Hola")
```

[16]:
```
"HolaHola"
```

Por el otro lado, `f` no funcionará sobre un vector. A diferencia de `A^2`, la cual es una operación bien definida, el signifiado de `v^2` para un vector, `v`, es ambigua.

[17]:
```
v = rand(3)
```

[17]:
```
3-element Array{Float64,1}:
 0.7987915209545873
 0.7759415604994804
 0.6881524039195559
```

[18]:
```
f(v)
```

```
    MethodError: no method matching ^(::Array{Float64,1}, ::Int64)
Closest candidates are:
  ^(!Matched::Float16, ::Integer) at math.jl:795
  ^(!Matched::Missing, ::Integer) at missing.jl:124
  ^(!Matched::Missing, ::Number) at missing.jl:97
  ...
```

3.4.3. Difusión (Broadcasting)

Si ponemos `.` entre el nombre de la función y su lista de argumentos, le estamos diciendo a la función que se "difunda" (haga broadcasting) sobre los elementos del input.

Primero veamos la diferencia entre `f()` y `f.()`.

[24]:
```
A = [i + 3*j for j in 0:2, i in 1:3]
```

[24]:
```
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9
```

[25]:
```
f(A)
```

[25]:
```
3×3 Array{Int64,2}:
  30   36   42
  66   81   96
 102  126  150
```

Cómo se vio antes, para una matriz, `A`,

```
f(A) = A^2 = A * A
```

`f.(A)` por el otro lado va a regresar un objeto que contiene el cuadrado de `A[i, j]` en su entrada correspondiente.

[26]:
```
B = f.(A)
```

[26]:
```
3×3 Array{Int64,2}:
  1   4   9
 16  25  36
 49  64  81
```

[27]:
```
A[2, 2]
```

[27]:
```
5
```

[28]:
```
A[2, 2]^2
```

[28]:
```
25
```

[29]:
```
B[2, 2]
```

[29]:
```
25
```

3.4.4. Añadiendo un nuevo método

[30]:
```
function f(x::Complex{Float64})
    x^3
end
```

[30]:
```
f (generic function with 2 methods)
```

[31]:
```
f(2)
```

[31]:
```
4
```

[32]:
```
f(2.0+0*im)
```

[32]:
```
8.0 + 0.0im
```

4. Extensión del Lenguaje Julia mediante paquetes

4.1. Administrador de paquetes

Julia tiene màs de 1686 paquetes registrados, conformando una gran parte del ecosistema de Julia.

Para ver todos los paquetes, visita

https://pkg.julialang.org/

o bien

https://juliaobserver.com/

Ahora vamos a aprender a usarlos

La primera vez que quieres usar un paquete en Julia, hay que agregarlo

[5]:
```
using Pkg
Pkg.add("Example")
```

```
Resolving package versions...
 Updating `~/.julia/environments/v1.1/Project.toml`
[no changes]
 Updating `~/.julia/environments/v1.1/Manifest.toml`
[no changes]
```

Cada vez que usas Julia (empezar una nueva sesión en el REPL, abrir un notebook por primera vez, por ejemplo), tienes que cargar el paquete usando la palabra reservada `using`

[6]:
```
using Example
```

En el código fuente de `Example.jl` en
https://github.com/JuliaLang/Example.jl/blob/master/src/Example.jl
Vemos una función declarada como

```
hello(who::String) = "Hello, $who"
```

Si cargamos `Example`, debemos poder llamar `hello`

[15]:
```
hello("it's me. I was wondering if after all these years you'd like to␣
↪meet.")
```

[15]:
```
"Hello, it's me. I was wondering if after all these years you'd like to␣
↪meet."
```

Con el Admistrador de paquetes podemos cargar, actualizar, borrar paquetes.

Hay paquetes que dependen de otros paquetes. El administrador de paquetes carga también los paquetes que necesite el paquete que estamos cargando.

Cada paquete puede tener diferentes versiones para las diferentes versiones de Julia. El adminisstrador de paquetes de Julia resuelve las dependendias cargandos la versiones de los paquetes que sean compatibles con la versión de Julia que estemos utilizando.

4.2. Programación en paralelo y distribuida

Julia ofrece diferentes niveles de paralelismo que podemos dividirlos en tres categorías principales:

- Julia Coroutines (Green Threading)
- Multi Threading
- Procesamiento distribuido

Julia Tasks: *@task*, *@async*, *@spawn*, *@spawnat*, ...

Multi Threading: Julia también admite múltiples subprocesos, donde la ejecución se bifurca y se ejecuta una función en todos los subprocesos. Los subprocesos paralelos se ejecutan de forma independiente y, en última instancia, deben unirse en el subproceso principal de Julia para permitir que continúe la ejecución en serie.
El subprocesamiento múltiple se considera experimental, ya que Julia aún no es completamente segura para subprocesos.

Procesamiento distribuido: Existen paquetes externos útiles para programación distribuida como MPI.jl y DistributedArrays.jl.

Table 1: Native Julia concepts for parallel & distributed computing

Concept	Computation	Memory	Common Tools
Coroutines & Worker Processes	Single-core asynchronous, parallel, distributed	Not shared, shared (`SharedArray`)	`@distributed`, `@sync`, `@async`, `Channels`, `SharedArray`, `@spawn`, `@remotecall`, ...
Native OS Threads	Parallel	Shared	`@threads`, `@threadcall`, `atomic_add!`, ...

4.3. Relaciones con otros lenguajes

4.3.1. Relaciones con Python: Paquete PyCall.jl (Páll Haraldsson+ 133 contributions)

Este paquete proporciona la capacidad de llamar directamente e interoperar completamente con Python desde el lenguaje Julia. Puede importar módulos de Python arbitrarios desde Julia, llamar a funciones de Python (con conversión automática de tipos entre Julia y Python), definir clases de Python desde los métodos de Julia y compartir grandes estructuras de datos entre Julia y Python sin copiarlas.

4.3.2. SymPy.jl (john verzani)

El paquete 'SymPy' se conecta con la [librería symyy] de Python (http://www.sympy.org) a través de 'PyCall'. La idea básica es hacer un nuevo tipo de Julia 'Sym' para manejar objetos simbólicos. Para este tipo, las funciones básicas de SymPy y las funciones apropiadas. de 'Julia' se tratan simbólicamente y no se evalúan de inmediato.
Este tipo se crea con el constructor ‘Sym‘, la función‘ symbols‘ o la macro ‘@ vars‘.

sympy es una librería de Python para las matemáticas simbólicas. Se trata de un sistema de álgebra computacional con las funciones usuales. sympy está escrito enteramente en Python.
Ejemplo de ejecución con Python:

```
from sympy import *
x, y = symbols('x, y')
expand((x+y)**3)
```

$x^3 + 3x^2y + 3xy^2 + y^3$

Paquete SymPy.jl permite llevar a Julia la funcionalidad Sympy de Python a través de PyCall
Ejemplo de ejecución con Julia:

```
using SymPy
x,y =symbols("x,y")
expand((x+y)^3)
```

$x^3 + 3x^2y + 3xy^2 + y^3$

4.4. Julia and Big Data

Julia todavía no puede reemplazar a Python o a R ya que algunas bibliotecas útiles para realizar análisis de Big Data simplemente no están disponibles.

Sin embargo Julia ya dispone de paquetes para el estudio de datos como DataFrames.jl y numeroso recursos para obtener plots como Plots.jl.
DataFrames.jl dispone de los recursos usuales para manejar datos: leer, escribir, ordenar, etc.
Plots.jl dispone de diversos recursos para crear gráficos relacionados con el estudio de datos

DataFrames.jl maneja marcos de datos estructurados en filas y columnas. Normalmente cada columna tiene un tipo fijo y en cada fila los tipos van cambiando.

Dispone de métodos más o menos elaborados para cortar y cortar en cubitos los datos, como "seleccionar" filas, columnas y celdas por nombre o por número; filtrando filas; "recodificar" nombres de columnas y filas; normalizar datos (por ejemplo, convertir unidades de medida); agregar nuevas columnas (por ejemplo, sumar algunos campos)...

Proporciona medios para tratar los datos entrantes que violan las restricciones de integridad de los tipos de fila y columna.

5. Un paquete para visualizar conjuntos de Julia

El objetivo es desarrollar en Julia un paquete para obtener un plot en el que se puedan **visualizar las distintas cuencas de attracción obtenidas al iterar una función racional en la esfera de Riemann**. Además se pueden calcular otros invariantes como la **dimensión fractal, número de cuencas de atracctión, calculo de funciones de Lyapunov** etc.

El conjunto de Julia se puede "visualizar" como la frontera topológica de cualquiera de estas cuencas de atracción.

https://github.com/luisjavierhernandez/PBURF.jl

La dificultad surge al **implementar modelos matemáticos como: la esfera de Riemann**, funciones racionales, iteración de una función, métricas en la esfera de Riemann, etc.

Estos modelos hay que **manejarlos con los tipos que admite Julia** y **definir las funciones mediante la sintaxis de Julia**.

Otras dificultades que aparecen están relacionadas con uso de **matrices de gran tamaño** para la obtención de imagenes con buena resolución.

También hay que utilizar técnicas adecuadas para **evitar indeterminaciones** que aparecen al dividir por "números" próximos a cero.

En el siguiente ejemplo aplicamos el método de Newton a un polinomio para obtener la función racional correspondiente:

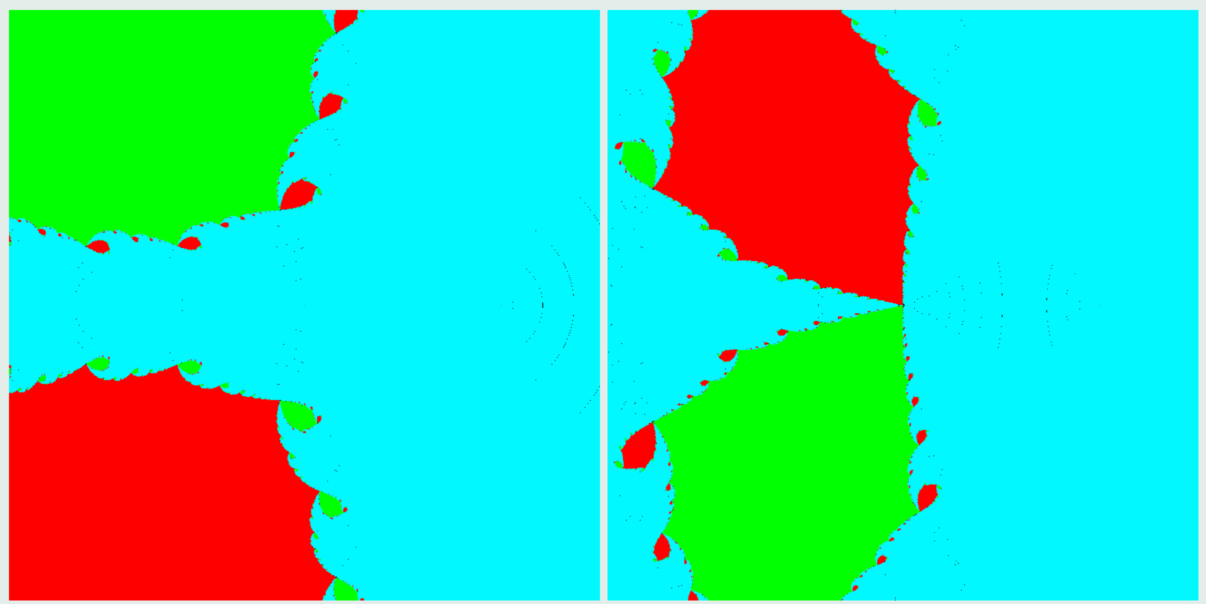
Tomaremos como polinomio $p(z) = (z^3 - 1)(z - 1)$, que tiene $z = 1$ como raíz doble y $z = e^{\frac{2\pi i}{3}}, z = e^{\frac{4\pi i}{3}}$ como raices simples, y aplicaremos el método de Newton.

$$N(p)(z) = \frac{-1 - 2z^3 + 3z^4}{-1 - 3z^2 + 4z^3}$$

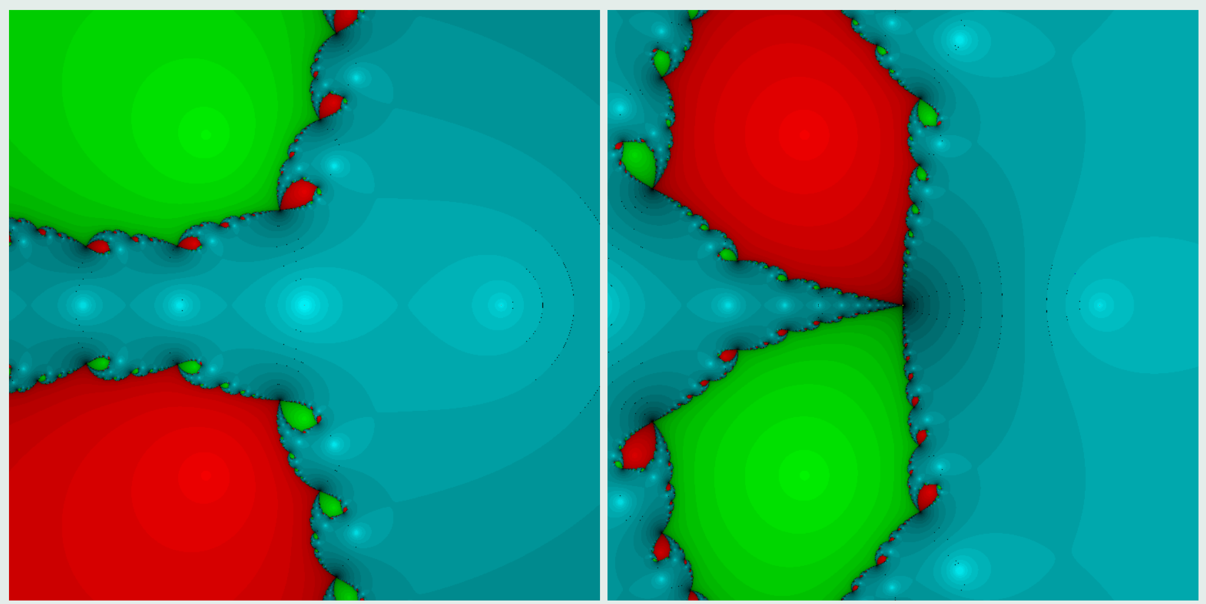
$$\mathrm{HN}(p)(z, t) = (-t^4 - 2z^3 t + 3z^4, -t^4 - 3z^2 t^2 + 4z^3 t),$$

$$\mathrm{HN}(p)^P([z : t]) = [-t^4 - 2z^3 t + 3z^4 : -t^4 - 3z^2 t^2 + 4z^3 t].$$

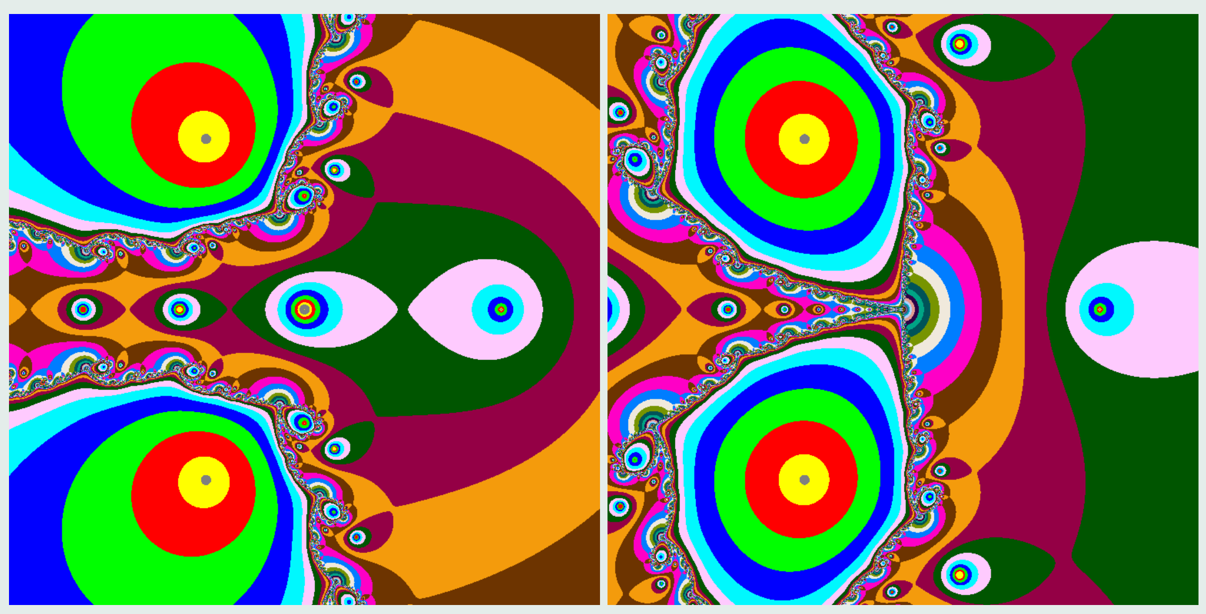
Criterio: posición

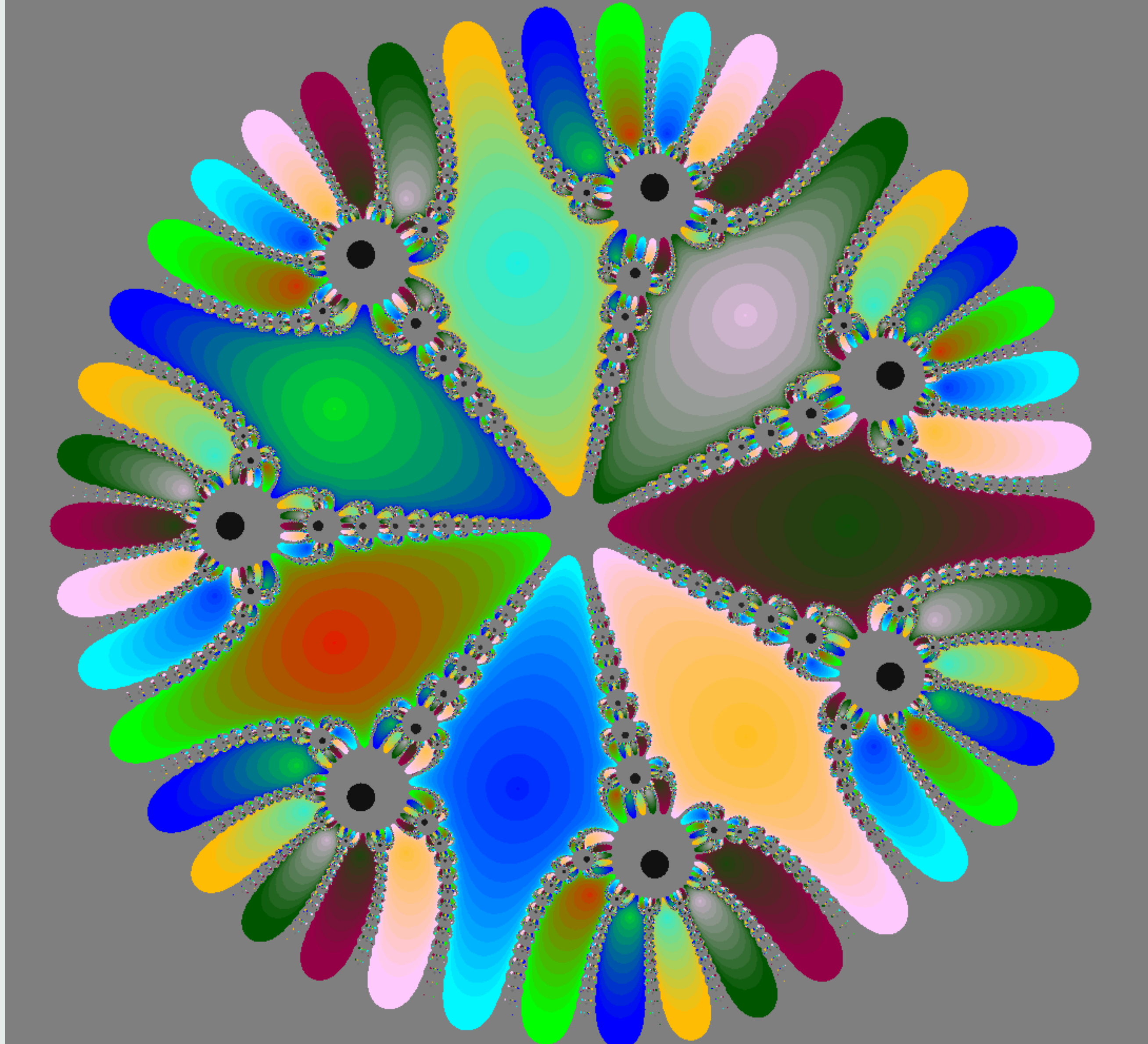


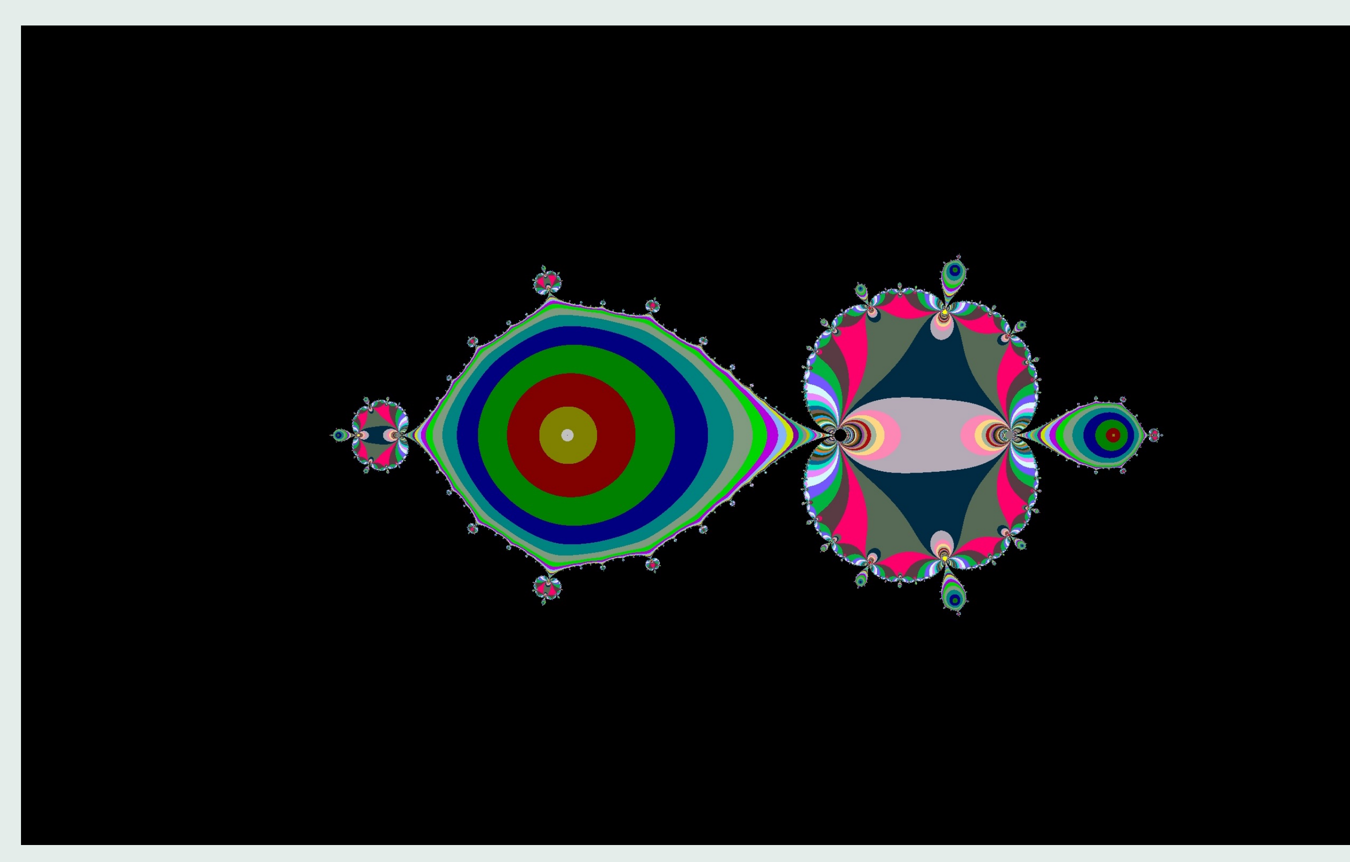
Criterio: posición-iteración



Criterio: iteración







The above discrete semi-flow has been studied by J. Milnor [5] by taking the iteration of the map $f\colon \hat{\mathbb{C}} \to \hat{\mathbb{C}}$ given by $f(z) = z^2 - 0.744336 + 0.121198i$, where $\hat{\mathbb{C}}$ denotes the Alexandroff compactification of $\mathbb{C}$. The fixed points of function f are: $\infty, -0.4990073 + 0.0606592i$ and $1.499003 - 0.0606592i$. The infinity point ∞ is super-attractor and both $-0.4990073 + 0.0606592i$ and $1.499003 - 0.0606592i$ are repulsors. The figures represent the attraction basin of the infinity point ∞ and one can also see the iterative logarithmic spiral structure that Milnor comments on in his article.

References

[1] MIGUEL RAZ GUZMÁN MACEDO, Tutorial de Julia en Español https://www.youtube.com/watch?v=LbTbs-0p0uc

[2] TUTORIALS, https://juliabox.com/, https://julialang.org/learning/

[3] GIRAY ÖKTEN, First Semester in Numerical Analysis with Julia.

[4] J. M. GARCÍA CALCINES, L. J. HERNÁNDEZ PARICIO M. MARAÑÓN GRANDES, AND M. T. RIVAS RODRÍGUEZ, *Regions of attraction, limits and end points of an exterior discrete semi-flow*, Journal of Homotopy and Related Structures, vol. 1(1), 2018, pp.1–26

[5] J. W. MILNOR, *Dynamics in One Complex Variable: Introductory Lectures*, Stony Brook IMS, 1990, arXiv:math/9201272.

MUCHAS GRACIAS POR SU ATENCIÓN!!!