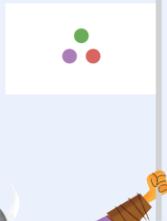


# An introduction to the **julia** language for researchers

Universidad de Sevilla Oct 2023

Raul Llamas

[raulcarlos.llamas@universidadeuropea.es](mailto:raulcarlos.llamas@universidadeuropea.es)



# Introduction

Course repository in Github:

[https://github.com/flt-acdesign/Mathematical\\_tourism\\_with\\_Julia](https://github.com/flt-acdesign/Mathematical_tourism_with_Julia)



# Objectives of this workshop

- Get a broad overview of what is Julia and what it is good for
- *Don't get lost in the details*, use “GPT” tools generously (not only for Julia)
- Experiment with notebooks and think of educational and personal uses
- Learn some basic concepts about parallelism and High Performance Computing
- Have fun and enjoy!

# What is Julia?



[What makes Julia so awesome?](#)

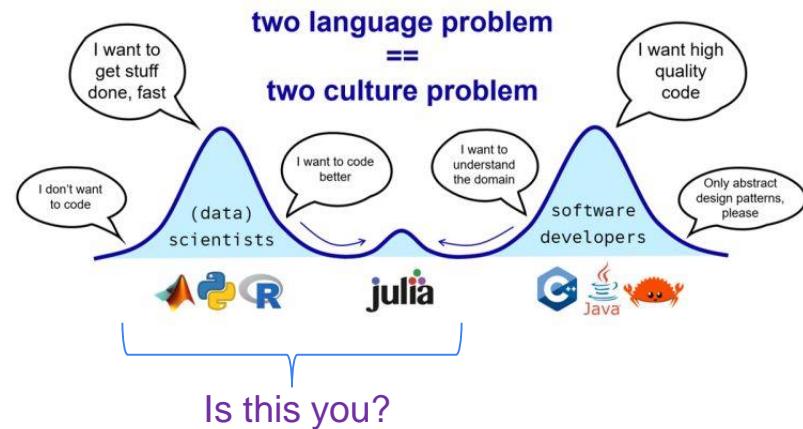
Julia is a language for **Scientific Computing**

**Scientific Computing** is the number crunching required by scientists (and Engineers) to either generate or process large amounts of numerical (or other) data.

Julia intends to solve the “two language problem”, whereby “domain experts” write rough prototypes in an “easy” language like Matlab or Python and then a team of professional programmers has to convert the code to C or equivalent for performance.

With Julia, a scientist can write high performance software from day 1.

The “**Julia**” name makes internet search a nightmare, instead, use “**Julialang**” in Google etc...



Why is it called Julia?

In an interview with [InfoWorld](#) in April 2012, Karpinski said of the name "Julia": "*There's no good reason, really. It just seemed like a pretty name.*"

*Yes, sure...*



# Where does it come from? How much does it cost?

Julia started as a project at MIT to develop a “high level language” (i.e. easy) but with high performance in numerical computations. It is heavily influenced by Lisp, Python, Matlab and others



2009

Work on Julia was started in 2009, by [Jeff Bezanson](#), [Stefan Karpinski](#), [Viral B. Shah](#), and [Alan Edelman](#)

2012

[Blogpost explaining the mission of Julia](#)



[Alan Edelman](#) and his dog, an important contributor to Julia, it seems

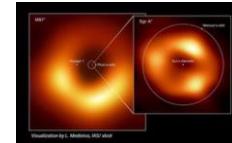
Julia is completely free and open source

Julia enters the “petaflops” club

2015 2017 2018

2023

[State of Julia](#)



High resolution black hole visualization (coming soon, these are current resolutions)

[JuliaHub](#) created, the company which develops and maintains Julia and provides (paid) HPC services

Julia 1.0 released

Julia gets 20<sup>th</sup> position in the Tiobe index

		Aug 2023	Aug 2022	Change	Programming Language	Ratings	Change
1	1	Python	13.23%	-2.30%			
2	2	C	11.41%	-3.35%			
3	4	C++	10.63%	+0.48%			
4	3	Java	10.33%	-2.34%			
5	5	C#	7.04%	+1.64%			
6	8	JavaScript	3.29%	+0.89%			
7	6	Visual Basic	2.63%	-2.28%			
8	9	SQL	1.52%	-0.31%			
9	7	Assembly language	1.34%	-1.41%			
10	10	PHP	1.27%	-0.09%			
11	21	Scratch	1.22%	+0.82%			
12	15	Go	1.16%	+0.20%			
13	17	MATLAB	1.09%	+0.27%			
14	18	Fortran	1.02%	+0.24%			
15	31	COBOL	0.96%	+0.93%			
16	16	Q#	0.92%	+0.01%			
17	19	Ruby	0.91%	+0.08%			
18	11	Swift	0.90%	-0.35%			
19	22	Rust	0.89%	+0.32%			
20	28	Julia	0.85%	+0.41%			

# Why it may be interesting for you? (or not)



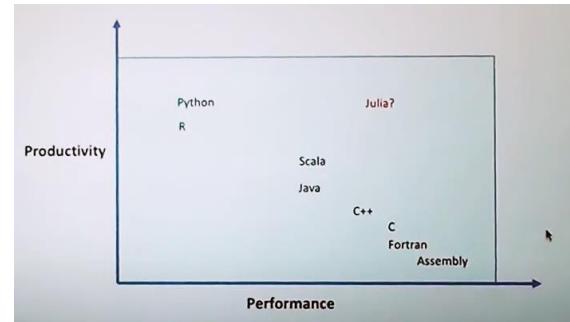
[Julia in 100 seconds](#)

## THE GOOD

Speed and productivity. Hundreds of times faster than Python and as fast as C

Easy to learn and use. Plenty of resources and a good ecosystem.

Extremely elegant language, feels like mathematics and it's excellent to express mathematical concepts.



From the Celeste project

## THE BAD

No generic way to compile executable files ( [although it can be done in some cases](#))

Not very easy support, or ecosystem, for 3D graphics, particularly for real time

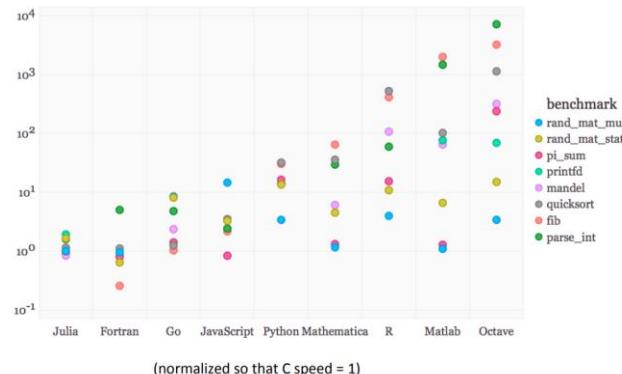
“Time to first plot”. Due to the compilation process Julia may seem quite slow the first time the code is executed (although its getting better with each version).

## THE UGLY

Its name (*for a computer language*)

## Performance on synthetic benchmarks

[ loops, recursion, etc., implemented in most straightforward style ]



# Some specific characteristics of Julia (1/2)

**Julia is Not Object Oriented (OOP)**, by a good design choice. Julia is mainly oriented to Functional Programming (FP), although it is not a “pure functional” language. Side effects are allowed (although discouraged)

The encapsulation of OOP can be reproduced using Modules and polymorphism is in fact a weak form of multiple dispatch



[Multiple Dispatch](#) A function in Julia can have many “methods” which are triggered depending on the type of the inputs. Julia will look at the types of the inputs and will “dispatch” the appropriate method.  [Multiple dispatch is extremely effective.](#)

Let’s define:

```
function add(x::Int, y::Int) = x + y  (a method which adds integer numbers in the usual way)
```

and then (in the same session);

```
function add(x::Fruit, y::Fruit) = "juice"  (a method “specialized” on data of type “Fruit”, returns the word “juice”)
```

The function `add` has now two methods (two specializations), but it’s still a unique function in Julia

# Some specific characteristics of Julia (2/2)

**Composability** is a consequence of functional programming and allows to safely “compose”, or plug, functions with functions. This facilitates scalability and code abstractions. Watch the great  [Alan Edelman](#) explain composability.

`(sin ∘ cos)(x)` (*composed function of cos with sin applied to x, which is similar, but different from*)

`sin(cos(x))` (*just watch the video...*)

## Just-in-time (JIT) compilation with LLVM.

When executed for the first time, the Julia code is parsed and “lowered” to increasingly optimized intermediate representations (IR) by the LLVM compiler.

At the end, highly optimized machine code for the current hardware is produced and run. Subsequent executions run on already compiled code, so the first execution is slower



[LLVM in 100 seconds](#)



LLVM is also the compiler of C, C++, Lisp, Fortran, Kotlin, Ruby, Swift, Lua, Rust and many others, so it's state of the art nowadays

# Language comparison:

For a “cheatsheet” comparing basic syntax between Matlab, Python and Julia use:

<https://cheatsheets.quantecon.org/>

Julia is most similar to Matlab, code conversion is almost trivial in simple cases

However, note that Matlab is much slower than Julia in most cases

Matlab is significantly slower than Julia on simple evaluation -  
MATLAB Answers – MATLAB Central (mathworks.com)

Julia is typically more than 100 times faster than Python (in fair comparisons – generally much more)

Julialang: Comparing the Speed of Julia, C, and Python  
(copyprogramming.com)

Check this official Julia documentation page for more information if you are transitioning from another language

Noteworthy Differences from other Languages · The Julia Language

## Matlab

Reshape (to 5 rows, 2 columns)

```
A = reshape(1:10, 5, 2)
```

```
A = A.reshape(5, 2)
```

```
A = reshape(1:10, 5, 2)
```

Convert matrix to vector

```
A(:)
```

```
A = A.flatten()
```

```
A[:, :]
```

Flip left/right

```
fliplr(A)
```

```
np.fliplr(A)
```

```
reverse(A, dims = 2)
```

Flip up/down

```
flipud(A)
```

```
np.flipud(A)
```

```
reverse(A, dims = 1)
```

Repeat matrix (3 times in the row dimension, 4 times in the column dimension)

```
repmat(A, 3, 4)
```

```
np.tile(A, (4, 3))
```

```
repeat(A, 3, 4)
```

Preallocating/Similar

```
x = rand(10)
y = zeros(size(x, 1), size(x, 2))
```

```
x = np.random.rand(3, 3)
y = np.empty_like(x)
```

```
x = rand(3, 3)
y = similar(x)
```

N/A similar type

```
# new dims
y = np.empty((2, 3))
```

```
# new dims
y = similar(x, 2, 2)
```

Broadcast a function over a collection/matrix/vector

```
f = @(x) x.^2
g = @(x, y) x + 2 + y.^2
x = 1:10
y = 2:11
f(x)
g(x, y)
```

Functions broadcast directly

```
def f(x):
    return x**2
def g(x, y):
    return x + 2 + y**2
x = np.arange(1, 10, 1)
y = np.arange(2, 11, 1)
f(x)
g(x, y)
```

Functions broadcast directly

```
f(x) = x^2
g(x, y) = x + 2 + y^2
x = 1:10
y = 2:11
f.(x)
g.(x, y)
```

Functions broadcast directly

# The future of Julia

Before you invest your career on a new computer language you may want to know if you are making the best choice.

Today the “safe choice” is **Python** but **Julia** was slated to be the “**Python killer**” due to its performance (about 100 times faster than **Python**) and its elegance.

And then, in May 2023, the creator of LLVM, the **Julia** compiler (and also of Apple’s Swift language, Chris Lattner), made public the “**Mojo**” project (talk about bad name choices...)

**In theory**, **Mojo** will be as fast as **Julia** but 100% compatible with **Python** syntax and will be oriented to machine learning (which is just like saying to numerical analysis).

*If **Mojo** succeeds many of the programming techniques covered in this course will still be of use.*



[Mojo in 100 seconds](#)



[Chris Lattner on Mojo and Julia](#)



# Mandelbrot in Julia vs Mojo

This code comparison shows that even a rather verbose **Julia** code is significantly more concise than **Mojo**, for a comparable performance

*Your call...*

using Plots

```
const xn = 960
const yn = 960
const xmin = -2.0
const xmax = 0.6
const ymin = -1.5
const ymax = 1.5
const MAX_ITERS = 200

function mandelbrot_kernel(c)
    z = c
    for i = 1:MAX_ITERS
        z = z * z + c
        if abs2(z) > 4
            return i
        end
    end
    return MAX_ITERS
end
```

Julia

```
function compute_mandelbrot()
    result = zeros(yn, xn)
```

```
    x_range = range(xmin, xmax, xn)
    y_range = range(ymin, ymax, yn)
```

```
    Threads.@threads for j = 1:yn
        for i = 1:xn
            x = x_range[i]
            y = y_range[j]
            result[i, j] = mandelbrot_kernel(complex(x, y))
        end
    end
    return result
end
```

```
result = compute_mandelbrot()
```

```
x_range = range(xmin, xmax, xn)
y_range = range(ymin, ymax, yn)
heatmap(x_range, y_range, result)
```

```
from benchmark import Benchmark
from complex import ComplexSIMD, ComplexFloat64
from math import pi
from python import Python
from runtime.lcli import num_cores, Runtime
from algorithm import parallelize, vectorize
from tensor import Tensor
from utils.index import Index
```

```
alias width = 960
alias height = 960
alias MAX_ITERS = 200

alias min_x = -2.0
alias max_x = 0.6
alias min_y = -1.5
alias max_y = 1.5

alias float_type = DType.float64
alias SIMD_width = SIMDwidthof[float_type]()
```

```
def show_plot(tensor: Tensor[float_type]):
    alias scale = 10
    alias dpi = 64

    np = Python.import_module("numpy")
    plt = Python.import_module("matplotlib.pyplot")
    colors = Python.import_module("matplotlib.colors")
```

```
    numpy_array = np.zeros((height, width), np.float64)

    for row in range(height):
        for col in range(width):
            numpy_array.itemset((col, row), tensor[col, row])

    fig = plt.figure(1, (scale * height / width), dpi)
    ax = fig.add_axes([0.0, 0.0, 1.0, 1.0], False, 1)
    light = colors.LightSource(315, 10, 0, 1, 1, 0)
```

```
    image = light.shade(
        numpy_array, plt.cm.hot, colors.PowerNorm(0.3), "hsv", 0, 0, 1.5
    )
    plt.imshow(image)
    plt.axis("off")
    plt.show()
```

Mojo

```
fn mandelbrot_kernel SIMD[
    SIMD_width: Int,
](c: ComplexSIMD[float_type, SIMD_width]) -> SIMD[float_type, SIMD_width]:
    """A vectorized implementation of the inner mandelbrot computation."""
    var z = ComplexSIMD[float_type, SIMD_width][0, 0]
    var iters = SIMD[float_type, SIMD_width][0]
```

```
var in_set_mask: SIMD[DType.bool, SIMD_width] = True
for i in range(MAX_ITERS):
    if not in_set_mask.reduce_or():
        break
    in_set_mask = z.squared_norm() <= 4
    iters = in_set_mask.select(iters + 1, iters)
    z = z.squared_add(c)
```

return iters

```
fn parallelized():
    let t = Tensor[float_type](height, width)
```

```
@parameter
fn worker(row: Int):
    let scale_x = (max_x - min_x) / width
    let scale_y = (max_y - min_y) / height
```

```
@parameter
fn compute_vector SIMD_width: Int | col: Int:
    """Each time we operate on a 'SIMD_width' vector of pixels."""
    let cx = min_x + (col + iota[float_type, SIMD_width]()) * scale_x
    let cy = min_y + row * scale_y
    let c = ComplexSIMD[float_type, SIMD_width](cx, cy)
    t.data().SIMD_set(c, SIMD_width)(
        row * width + col, mandelbrot_kernel SIMD[SIMD_width](c)
    )
```

```
# Vectorize the call to compute_vector where call gets a chunk of
pixels.
vectorize[SIMD_width, compute_vector](width)
```

with Runtime() as rt:

```
@parameter
fn bench_parallel SIMD_width: Int |():
    parallelize[worker](rt, height, 5 * num_cores())
```

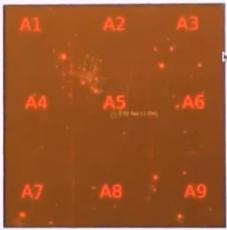
```
alias SIMD_width = SIMDwidthof[DType.float64]()
let parallelized = Benchmark().run(bench_parallel[SIMD_width]())
print("Parallelized:", parallelized, "ms")
```

```
try:
    _ = show_plot(t)
except e:
    print("Failed to show plot:", e.value)
```

```
def main():
    parallelized()
```

# Notable examples of Julia in industry and academia

## Celeste project



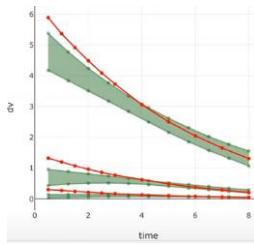
Catalogue of 180 million stars and galaxies in the visible universe (created in 15 minutes with 9000 nodes, 100\_000 cores)

Brought Julia to the “PetaFlop” club (after Fortran and C), first high level language to join the club

Key project for the maturity of Julia in High Performance Computing

[JuliaCon 2017 video](#)

## Pharma

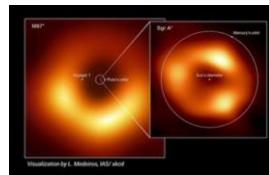


The pharmaceutical industry is a strong user of Julia mainly as a consequence of the proprietary libraries and companies built on the “DifferentialEquations” package.

In pharma, stiff differential equations representing chemical reactions are the norm and Julia excels at this. At least two covid vaccines were developed with the help Julia

[Pumas video](#)

## Black Holes



In 2017 the first synthetic image of the black hole at the center of the Messier 87 galaxy was released. This was generated using Python and C code and took about a week.

The lead researcher has now switched to Julia and can do the same job on a laptop in 15 minutes...

[Imaging Black Holes with the Event Horizon Telescope 2023](#)

[High performance Black Hole Imaging 2022](#)

## Boeing

A screenshot of a news article from Forbes. The headline reads "Deepak Vinchhi, on plans at JuliaHub after recent funding from a VC firm backed by Boeing". Below the headline, there is a snippet of text and a small image of Deepak Vinchhi. The article is dated "NEW YORK - JULY 27, 2022 07:00 ET".

There's something going on between Boeing and Julia...

A venture capital company owned by Boeing has invested 13 million dollars in JuliaHub, the company which maintains and develops Julia.

Boeing was sponsor of the [JuliaCon 2023](#)

Many other companies are using Julia at different levels, we'll see how this evolves in the future

# Installation



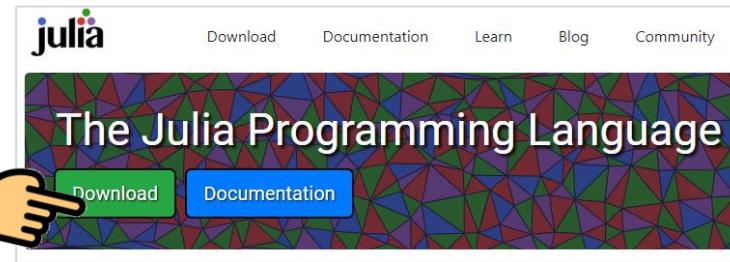
# Installing Julia



[How to install Julia and Visual Studio Code](#)

1 Go to [The Julia Programming Language web page \(julialang.org\)](https://julialang.org)

2 Click on the “Download” button and then on your preferred version. If you are in windows on a decent computer (64bits processor) select the version highlighted below



Download Julia

Please star us on [GitHub](#). If you use Julia in your research, please [cite us](#). If possible, do consider [sponsoring us](#).

Current stable release: v1.9.3 (August 24, 2023)

Checksums for this release are available in both [MD5](#) and [SHA256](#) formats.

Windows [help]	64-bit (installer), 32-bit (portable)	32-bit (installer), 32-bit (portable)
macOS x86 (Intel or Rosetta) [help]	64-bit (.dmg), 64-bit (.tar.gz)	
macOS (Apple Silicon) [help]	64-bit (.dmg), 64-bit (.tar.gz)	
Generic Linux on x86 [help]	64-bit (glibc) (GPG), 64-bit (musl) <sup>[1]</sup> (GPG)	32-bit (GPG)
Generic Linux on ARM [help]	64-bit (AArch64) (GPG)	
Generic Linux on PowerPC [help]	64-bit (little endian) (GPG)	
Generic FreeBSD on x86 [help]	64-bit (GPG)	
Source	Tarball (GPG)	Tarball with dependencies (GPG)
		Github

3 Accept all the proposed settings during the installation.

4 After the installation find the Julia icon in your desktop and double-click on it, it will open the Julia REPL



```
julia> print("Hello Sevilla")
Hello Sevilla
```

“The REPL”, an instance of Julia running

# REPL and package manager



The Julia REPL

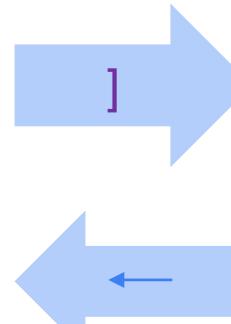
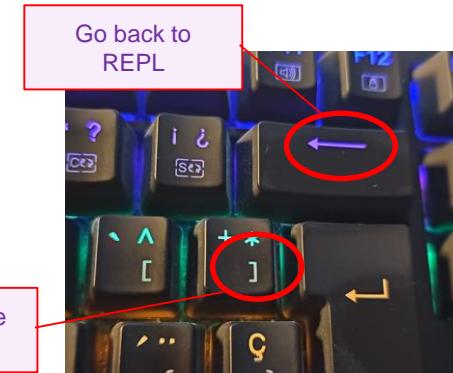
“REPL” stands for Read Evaluate Print Loop and it’s where you are just after starting Julia.

It is the most basic way in which Julia can be written using the “console”

The package manager is a “mode” of the REPL in which you can add “libraries” (packages)

It is accessed from the REPL by typing: **]**

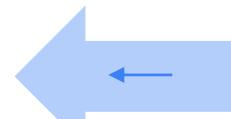
To exit the package manager and go back to the REPL type: **←**



```
Julia 1.9.0-rc2
Documentation: https://docs.julialang.org
Type "?" for help, "]?" for Pkg help.
Version 1.9.0-rc2 (2023-04-01)
Official https://julialang.org/ release

julia> print("Hello Sevilla")
Hello Sevilla
julia>
```

REPL



```
Julia 1.9.0-rc2
Documentation: https://docs.julialang.org
Type "?" for help, "]?" for Pkg help.
Version 1.9.0-rc2 (2023-04-01)
Official https://julialang.org/ release

julia> print("Hello Sevilla")
Hello Sevilla
(@v1.9) pkg>
```

Name of the current environment (where the packages will be added). In this case, the global environment

Package manager

# Basic usage of the package manager



[Managing Julia packages](#)

The Julia package manager enables to install “packages” (libraries of code), update them, delete them and various other actions.

**It is easily the best package manager of any language today.**

To add a package, inside the package manager (after typing `]` in the REPL) type:

`add name_of_the_package`

```
(@v1.9) pkg> add pluto
  Updating registry at `C:\Users\USUARIO\.julia\registries\General.toml`
  Downloading [=====] 91.9 %
```

*In this case we are adding the “Pluto” package, which we will use for this course*

To check the status (versions) of the packages type: `st`

To update all packages to the latest version type: `update`

# >>> Add required packages now <<<

Packages (libraries) are downloaded from an internet repository (very often Github) when “added” in the package manager, so you need an internet connection.

After downloading, the packages are pre-compiled for your specific hardware by Julia. This takes a while, so this is a good time to add various packages that we will use in this course.

- 1- Copy the following block of text from the box below (select with the mouse and right-click “copy” or type “Ctrl-c”)

```
add Pluto PlutoUI Plots
```

- 2- Enter the package manager (type `]`) and paste the text. If “Ctrl+v” does not work, just do a mouse right-click after `pkg>`
- 3- Press `enter` (if the installation does not start on its own) and let Julia download and precompile the packages.

The box below has packages that take longer to install, open a new Julia REPL (click on the desktop icon), enter the package manager and copy the text to add them.

```
add Flux BenchmarkTools Images JuMP Cbc Statistics DataFrames Sound
```

- 4- While the installation of the second group of packages goes on, go back to the first Julia session and exit the package manager (`←`)

# Pluto notebooks for productive Julia development



Pluto is a Julia “program” which creates an automatically opens a locally hosted **webpage**, connected with Julia, where you can write and execute Julia code and add other non-code information in a very pleasant way.

Formally, Pluto is a notebook, like **Jupyter** (which means **Julia**, **Python** and **R**). You can also use Jupyter to write Julia notebooks (and many people do) but the main advantage of Pluto is that it is **reactive**.

A **reactive notebook** makes sure that the state of the code you see is consistent, i.e., when you change a value, all the code which needs to change in response will be re-evaluated. A good example is an Excel sheet where the values of some cells depend on the values of others. When you change some of those values, the rest of the cells are automatically updated.



[Watch the great Fons introducing Pluto to the world](#)

Sharing fully interactive notebooks

Goal: Send someone a link to play with your notebook on the web.

Requirements: no web architecture/container skills & free (as in beer).

Possible solution

Pluto back-end is lightweight

Runs on free 512MB heroku server ([demo](#))

make it one click



[Watch more Fons, another peculiar character in the Julia world](#)

Using JavaScript inside Pluto

You have already seen that Pluto is designed to be interactive. You can make fantastic exploratory documents using just the basic inputs provided by Pluto, together with the wide range of visualization libraries that Julia offers.

However, if you want to take your exploratory document one step further, then Pluto offers a great framework for combining Julia with HTML, CSS and JavaScript.

using Pluto; #, HyperTextLiteral,

```
function()
    println("Hello")
    println("World")
    println("Julia")
    println("Pluto")
    println("JavaScript")
    println("HTML")
    println("CSS")
    println("HyperText")
    println("Interpolating")
    println("Interpolating into HTML - Hyphenated words are converted to camelCase")
    println("Script loading")
    println("Loading method 1: ES6 imports")
    println("Loading method 2: script tag")
    println("Advanced")
    println("Script injection & observability/handle")
    println("Expanding variables with this")
    println("Threaded execution with this")

```

Prerequisites

Live docs

**Pluto is also open source and free:**

Github homepage <https://github.com/fonsp/Pluto.jl>



# Installing and starting Pluto

Pluto homepage <https://plutojl.org/>

If you have already added the Pluto package (explained in the previous pages), you have access to Pluto. To start it type in the REPL:

```
import Pluto; Pluto.run()      (adding a package is like buying a book and “importing” a package is like opening the book)
```

`Pluto.run()` calls the run function of the Pluto package and will open an empty notebook as a tab in your internet browser (it will even start the browser if required).

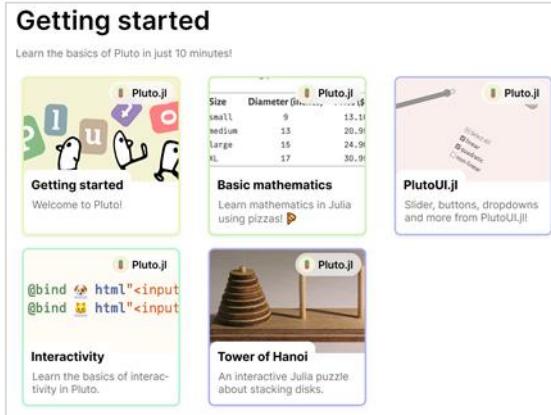
If you put the full path name or the url of an existing notebook in the brackets it will open it.

All the instructions are well explained in the Pluto homepage <https://plutojl.org/>

Go to

<https://featured.plutojl.org/>

for a good collection of Pluto examples running online



If you want to edit these examples click on the button at the top right;

Edit or run this notebook

and follow the instructions



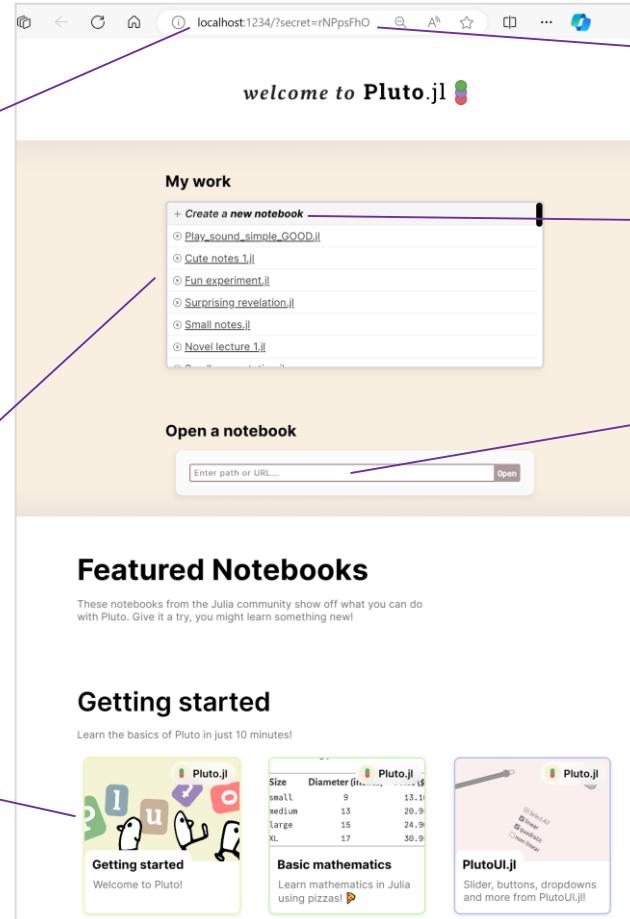
# Exploring Pluto

Pluto is a web page “served” by Julia, in this case in port 1234

These are previous notebooks on which you have worked, just click to open. The funny names (e.g. “cute notes”) are given automatically by Pluto to notebooks from which you exit without giving a name

A good way to learn Pluto is to open the example notebooks and play with it

## Pluto start page



The “secret” is a way to differentiate different Pluto sessions in different tabs, don’t worry about it

To **create** a notebook just click here

To **open** a Pluto notebook paste the full path to the file (or navigate by typing the path in the field) or paste a url of a file hosted in a cloud service

Both:

C:\Users\USUARIO\Lorenz\_Euler.jl

and

[https://github.com/MyRepo/MyProject/blob/main/Lorenz\\_Euler.jl](https://github.com/MyRepo/MyProject/blob/main/Lorenz_Euler.jl)

are well formatted names of possible Pluto (Julia) files



# Exploring Pluto

If you get this message when trying to open a file, and you trust the source, just click “Run notebook code” (top right)

The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** Shows the URL `localhost:1234/edit?id=7ac08010-7...`, the file name `SUARIO\lorenz_Euler_G000.jl`, and a button to `Run notebook code`.
- Code Cells:** There are three code cells, each with a "Code not executed in Safe preview" message.
  - Cell 1:** Contains `using Plots`.
  - Cell 2:** Contains the `Lorenz` function definition.
  - Cell 3:** Contains the `euler` function definition.
- Safe Preview Section:** A large box titled "Safe preview" contains the following text:

You are reading and editing this file without running Julia code.

When you are ready, you can run this notebook.

**Warning**  
Are you sure that you trust this file?  
A malicious notebook can steal passwords and data.

Pluto notebooks are pure Julia files but with some additional information (including the names and versions of the packages specified by the `using` keyword). A Pluto file can be run in Julia directly (unlike in the case of Jupyter notebooks)

Pluto is **REACTIVE**: the order of the cells in Pluto is not important, Pluto will track the dependencies of the variables and functions and execute everything that depends on anything which changes (e.g. a variable value changes or a cell is recompiled with “Shift+Enter”)

Use “Control+mouse\_wheel” or  
“Control+” or “Control-” to zoom in and  
out in Pluto (like in any other web page)

```

Pluto.jl ℹ

- using Plots ✓

lorenzf (generic function with 1 method)
• Function lorenzf([u₀, u, p, t])
  • u[0] = u₀
  • du[1] = u * (u[2] - u[1])
  • du[2] = u[1] + (p - u[3]) - u[2]
  • du[3] = u[1] * u[2] - β * u[3]
  • end

eulerf (generic function with 1 method)

plot.lorenz (generic function with 1 method)

# Parameters
• a = 10.0
• # Set parameters
• b = 10.0

p = 28.0
• p = 28.0

β = 2.4666666666666665
• β = 8/3

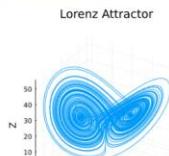
u₀ = [1.0, 0.0, 0.0]
• # Initial conditions
• u₀ = [1.0, 0.0, 0.0]

h = 0.01
• # Time parameters
• h = 0.01

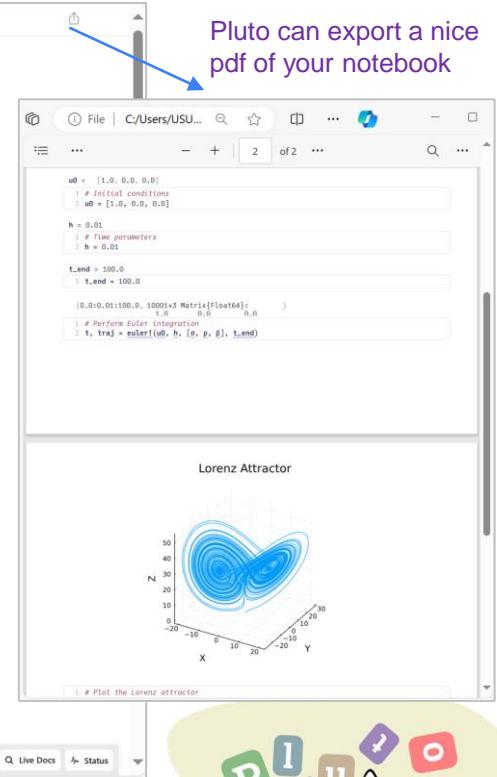
t_end = 100.0
• t_end = 100.0

• (0.0:0.01:100.0, 10001x3 Matrix{Float64};)
• # Perform Euler integration
• t, traj = eulerf(u₀, h, [a, β, B], t_end)

Lorenz Attractor



```



# Exploring Pluto

Exit the notebook (it will be saved with a funny name if you don't save it) and go back to the start page

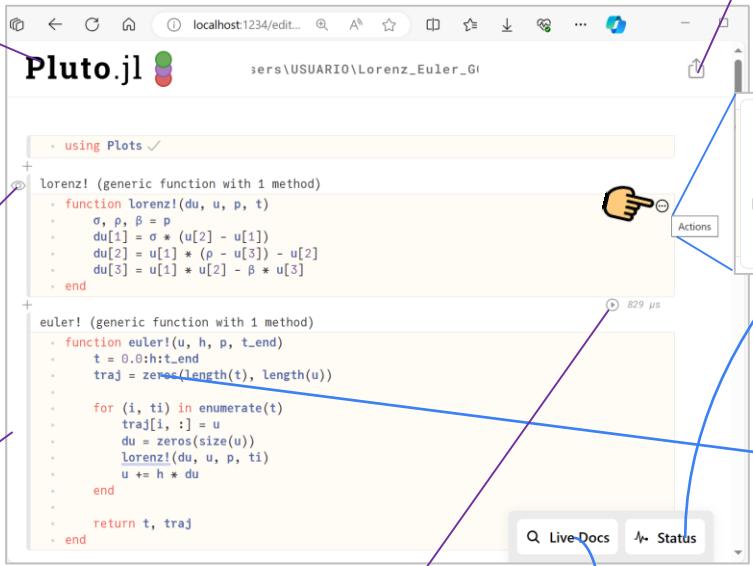
Hide/show the code in the cell. The results of the cell, which could be none, will always be shown above the cell

Click on the margin of the cell to move it up or down in the notebook.  
**Pluto does not care about the order of the cells**

An important peculiarity of Pluto is that a cell can only hold one "block of code" (excluding comments).

This is either a single variable assignment, a function definition or a "begin...end" block

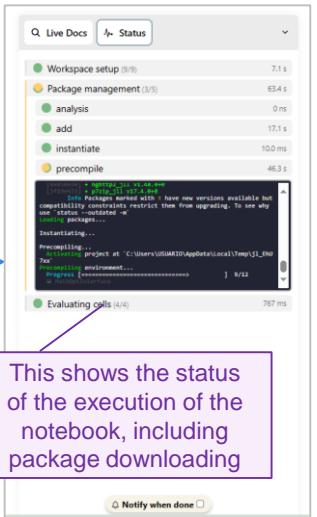
If you type several lines, Pluto will propose to either automatically split them or automatically wrap them inside a `begin...end` block



```
using Plots ✓
+
lorenz! (generic function with 1 method)
+   function lorenz!(du, u, p, t)
    ...
    du[1] = σ * (u[2] - u[1])
    du[2] = u[1] * (ρ - u[3]) - u[2]
    du[3] = u[1] * u[2] - β * u[3]
  end
+
euler! (generic function with 1 method)
+   function euler!(u, h, p, t_end)
    t = 0.0:t_end
    traj = zeros(length(t), length(u))
    for (i, ti) in enumerate(t)
      traj[i, :] = u
      du = zeros(size(u))
      lorenz!(du, u, p, ti)
      u += h * du
    end
    return t, traj
  end
```

Export the notebook, including as a pdf file

Disable cell prevents the execution of the code, use to debug



Q Live Docs ↗ Status

Workspace setup (9/9) 7.1s  
Package management (3/3) 65.4s  
analysis 0ms  
add 17.1s  
instantiate 10.0 ms  
precompile 46.3s

Evaluating cells (4/4) 767 ms

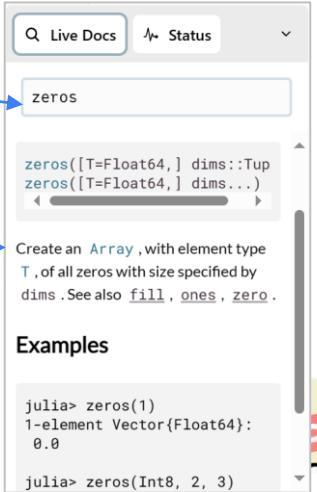
Notify when done

This shows the status of the execution of the notebook, including package downloading

To execute a cell, click here or, better, type "Shift+Enter"  
In theory you can also stop the execution with the stop button, but don't count on it.  
You will get the cell execution time at the end

In the REPL this would be `? zeros`

Click on a word in the code to get help, if available



Q Live Docs ↗ Status

zeros

zeros([T=Float64,] dims::Tuple{Int64,...})  
zeros([T=Float64,] dims...)

Create an `Array`, with element type `T`, of all zeros with size specified by `dims`. See also `fill`, `ones`, `zero`.

**Examples**

```
julia> zeros(1)  
1-element Vector{Float64}:  
0.0  
  
julia> zeros(Int8, 2, 3)
```

# Error messages in Julia

A large part of the time spent in developing software is used in fixing errors (“debugging”)

Error messages are always frustrating and not always very clear but [Julia tries to be as helpful as possible](#).

You can always copy the error message type and search in google, most likely you are not the first to encounter it...

“**no method matching**” is very typical, it happens when a function is called with input of a type for which there is no method (so check what you are sending to the function)

Julia REPL

```
ERROR: LoadError: MethodError: no method matching +(::Int64, ::String)
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...) at operators.jl:591
  +(::T, ::T) where T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128, UInt16,
  +(::Union{Int16, Int32, Int64, Int8}, ::BigInt) at gmp.jl:537
  ...
Stacktrace:
 [1] top-level scope
   @ ~/Learning/jul/first.jl:41
in expression starting at /Users/demo/Learning/jul/first.jl:41
```

Julia tries to help by proposing types of inputs for which there are methods

Pluto

```
BoundsError: attempt to access 3-element Vector{Float64} at index [31]
1. getindex @ essentials.jl:13 [inlined]
2. lorenz! @ Other: 5 [inlined]
3. euler!(::Vector{Float64}, ::Float64, ::Vector{Float64}, ::Float64) @ Other: 8
4. top-level scope @ Local: 2 [inlined]
  # Perform Euler integration
  t, traj = euler!(u0, h, [σ, ρ, β], t_end)
```

```
UndefVarError: `t` not defined
  # Plot the Lorenz attractor
  plot_lorenz(t, traj)
```

In Pluto a certain cell can give an error which is caused on a different cell. Click on the “Other...” to go to the line potentially causing the error.

All the cells affected by the error will show an error too, in this case “t” is not defined because its creating failed somewhere else, just go to the original error, fix it, and try to execute cell by cell to find when things fail

# Getting help in Julia (and you will need it)



[Getting Help in Julia](#)

Depending on where you are you can get immediate help from Julia in two ways:

In the REPL: Type “?” to enter the help mode and type the name of the function for which you need help

```
julia> # Type ? and then enter the keyword for which you want help
help> zeros
search: zeros count_zeros set_zero_subnormals get_zero_subnormals leading_zeros trailing_zeros zero iszero RoundToZero
zeros([T=Float64,] dims::Tuple)
zeros([T=Float64,] dims...)
Create an Array, with element type T, of all zeros with size specified by dims. See also fill, ones, zero.
```

In Pluto: Click on the button and then click on the keyword you are interested in.

In order for a function in Julia to provide help it has to be written with “Doc strings” like;

Write the “doc strings” just above the function definition with exactly the format shown

```
"""
ms2kt(v)

Convert a speed v from meters per second (m/s) to knots (kt)

# Examples
```julia-repl
julia> ms2kt(1)
1.94384449
```
"""

ms2kt(v) = v*1.94384449
```

If you click on “ms2kt” in the Pluto code or type it in the Live Docs form you will see this ->

The screenshot shows the Pluto interface with a search bar at the top. Below it, the word "ms2kt" is typed into a text input field. A tooltip or dropdown menu appears below the input field, displaying the function's docstring: "Convert a speed v from meters per second (m/s) to knots (kt)". Below the docstring, there is a section titled "Examples" containing the command "julia> ms2kt(1)" followed by the output "1.94384449".

The [Julia official documentation page](#) is an excellent source of information and examples. If you are like me you can download a [pdf file](#) with all the language documentation and print all of its 1644 pages to read in the beach.

# First Experiments



# Course notebooks (Rémy Vezy)



[Github repository for this course](#)

A good place to start learning the basic Julia syntax is a course developed by Rémy Vezy, who is a researcher in plant (vegetal...) structure.

He uses Pluto and has a collection of notebooks in Github accompanied by videos in YouTube

The screenshot shows a GitHub repository page for 'VEZY/julia\_course'. The repository has 2 branches and 0 tags. The main branch contains several files and folders related to the Julia course, such as 'Julia Remove cache', '.github/workflows', 'PlutoPages', 'content', 'www\_readme', 'LICENSE', and 'README.md'. The README file contains a section titled 'Julia course: from total beginner to power user' which describes the repository as a collection of resources to teach Julia programming. It also mentions a YouTube playlist and provides direct links to the notebooks. The repository has 47 commits, last updated 89d3de last month. On the right side, there is a sidebar with options like 'Clone' (HTTPS or GitHub CLI), 'Open with GitHub Desktop', and 'Download ZIP'. A large orange hand icon points to the 'Code' button at the top right of the main content area. A red line extends from this point to another orange hand icon pointing to the 'Download ZIP' button on the right side of the page.

# Opening the course notes in Pluto

In the following slides, an icon like  denotes that the text behind is the url to a Pluto notebook. Copy the text and paste it into the “Open a notebook” field in the Pluto start page.

**Direct links to the notebooks** 

You can use the notebooks either by using the direct links provided here, or by downloading/cloning the repository (see below).

To use the direct link, open julia, then type `using Pluto`, and execute the line of code provided below. If you need to install Pluto first, see below.

1. Variables and basic types in Julia



```
book = "https://raw.githubusercontent.com/VEZY/julia_course/main/content/1-variables_and_basic_types.jl"
```

welcome to Pluto.jl 

**My work**

- + Create a new notebook
- ① 00\_LIN\_FSD\_TOPO\_V0.119.jl
- ② 00\_LIN\_FSD\_TOPO\_V0.136.jl
- ③ Tiny\_theory.jl
- ④ 00\_LIN\_FSD\_TOPO\_V0.135.jl
- ⑤ 00\_LIN\_FSD\_TOPO\_V0.134.jl
- ⑥ Wild\_notes.jl
- ⑦ Important lecture.jl

**Open a notebook**

Enter path or URL...



[https://raw.githubusercontent.com/VEZY/julia\\_course/main/content/1-variables\\_and\\_basic\\_types.jl](https://raw.githubusercontent.com/VEZY/julia_course/main/content/1-variables_and_basic_types.jl)

# Basic Julia syntax (1/2) (Rémy Vezy)

These videos and Pluto notebooks below are an excellent starting point to learn basic Julia syntax



1- [Variables and basic types:](#)

[https://raw.githubusercontent.com/VEZY/julia\\_course/main/content/1-variables\\_and\\_basic\\_types.jl](https://raw.githubusercontent.com/VEZY/julia_course/main/content/1-variables_and_basic_types.jl)



2- [Arrays](#)

[https://raw.githubusercontent.com/VEZY/julia\\_course/main/content/2-arrays.jl](https://raw.githubusercontent.com/VEZY/julia_course/main/content/2-arrays.jl)



3- [Tuples](#)

[https://raw.githubusercontent.com/VEZY/julia\\_course/main/content/3-tuples.jl](https://raw.githubusercontent.com/VEZY/julia_course/main/content/3-tuples.jl)



4- [Dictionaries](#)

[https://raw.githubusercontent.com/VEZY/julia\\_course/main/content/4-dictionnaries.jl](https://raw.githubusercontent.com/VEZY/julia_course/main/content/4-dictionnaries.jl)



5- [Basic operators](#)

[https://raw.githubusercontent.com/VEZY/julia\\_course/main/content/5-basic\\_operators.jl](https://raw.githubusercontent.com/VEZY/julia_course/main/content/5-basic_operators.jl)

# Basic Julia syntax (2/2) (Rémy Vezy)



6- [String operators](#)

[https://raw.githubusercontent.com/VEZY/julia\\_course/main/content/6-string\\_operators.jl](https://raw.githubusercontent.com/VEZY/julia_course/main/content/6-string_operators.jl)



7- [Compound expressions \(begin, let...\)](#)

[https://raw.githubusercontent.com/VEZY/julia\\_course/main/content/7-compound\\_expressions.jl](https://raw.githubusercontent.com/VEZY/julia_course/main/content/7-compound_expressions.jl)



8- [Conditional statements \(if...else...\)](#)

[https://raw.githubusercontent.com/VEZY/julia\\_course/main/content/8-conditional\\_statements.jl](https://raw.githubusercontent.com/VEZY/julia_course/main/content/8-conditional_statements.jl)



9- [For loops \(each index, enumerate...\)](#)

[https://raw.githubusercontent.com/VEZY/julia\\_course/main/content/9-for\\_loops.jl](https://raw.githubusercontent.com/VEZY/julia_course/main/content/9-for_loops.jl)



10- [Functions](#)

[https://raw.githubusercontent.com/VEZY/julia\\_course/main/content/10-functions.jl](https://raw.githubusercontent.com/VEZY/julia_course/main/content/10-functions.jl)

# Linear Algebra and array virtuosity

Arrays are an extremely important data structure in Julia and the language provides an efficient syntax to use arrays and linear algebra support without the need to add any package.

It's very important to be "fluent" in the syntax around arrays. Here are some examples of usual code snippets (try yourself to follow these examples and play freely)

`A = rand(3,3); B = rand(3,3)` creates two 3x3 arrays of random Float64 numbers between 0 and 1. ";" here means "continue"

`inv(A)` same as `A^-1` calculates the inverse of A. `transpose(A) == A'` = `true` ways to compute the transpose of A

`b = rand(3)` creates a vector with random numbers. Vectors are **always column vectors** and Julia is "column major" (more later)

`x = A\b` solves the linear system  $Ax = b$

`A+B` adds all the elements of A and B, elementwise

`A^2` computes the operation `A * A`

`A.^2` computes the square of each individual element of the array (i.e.,  $a_{11}^2 \ a_{12}^2 \ a_{13}^2$  etc...)

`A[:,2]` extracts the elements intersection between all the elements in the first dimension and the elements of index 2 in the second dimension (so, in this case it returns the second column). `A[3,:]` returns the third row

`collect(1:5)` creates an array (in this case a column vector) from a range. `collect` is very general to create arrays from iterators

# Special symbols and idiosyncrasy (1/5)

The idiosyncrasy of a computer language can make the code look ugly and not easy to understand. Julia is quite reasonable in this aspect but it still uses some non-intuitive (but generally non arbitrary) notation.

The following is a list, ordered roughly by frequency of occurrence, of some symbols which are used in Julia:

**begin ... end** defines a block of code, necessary in Pluto but normally not very common in Julia.

( ) Function parameters, as in **function add(x,y) = x + y**

[ ] Array indices, as in **A[3,4]** Indices can also be given by a range, **A[1:2,4]** (elements in the first and second rows, column 4). This notation extends to **A[:, 4]**, which represents the elements in all the rows from the fourth column)

; After a statement it means “don’t show the result in REPL or Pluto”, it denotes the separator for keyword arguments in functions and the separator for rows in an array definition **A = [ 1 2 ; 3 4 ]**

(x, ) definition of tuple of a single element. Tuples are very important and, for example, the inputs to a function are a tuple.

# Special symbols and idiosyncrasy (2/5)

Julia is “1” indexed, i.e., the first element of an iterator V (e.g. a vector) is `V[1]` (unlike in C and other languages)

- Broadcasting operator (“dot operator”). It means that the operator which it precedes shall be applied elementwise to the arrays, as in `B .= A .* C`, where A, B and C are arrays of the same dimension and the result is an array of the products of the corresponding elements of B and C (a<sub>11</sub>\*c<sub>11</sub> a<sub>12</sub>\*c<sub>12</sub> etc...)

`{}` Denotes the type of a variable, as in `Vector{Float64}(undef, 3)`, which defines a vector of 3 dimensions of type `Float64`

`@` used to identify a macro, which is a function which operates on code and returns executable code. Common examples are `@time`, which returns the time taken to run a function and `@.`. Which “vectorizes” all the operations in its line (more later)

`::` Declares the variable type for an input in a function, as in `add(x::Float32, y::Float32)`

– Optional thousand separator, just for clarity, as in `1_000_000` to represent one million. `1000000` also works.

`%` remainder of an integer division, or modulo operator; `8 % 3` returns `2`. Same as `mod(8,3)`

`÷` result of an integer division; `10 ÷ 3` returns `3`

# Special symbols and idiosyncrasy (3/5)

`# This is a comment` everything after `#` is treated as a comment in the code in the same line

`"""`

`I am  
a comment`

a multiline comment

`"""`

`#=  
I am also  
a comment`

alternative multiline comment

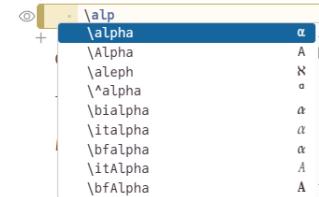
`=#`

! Convention for a mutating function (called “Bang!”). Mutating functions are those which change the value of the input passed to them. This is not “pure functional programming”, as this is a “side effect”, but can save memory allocations and make the code more efficient, for example in the function `mul!(Y, A, B) -> Y` which calculates the matrix-matrix or matrix-vector product \$AB\$ and stores the result in Y, overwriting the existing value of Y.

`\` escape character for \, for example to write file paths as in `c:\file.txt` equivalent to `c:\file.txt` in Windows

# Special symbols and idiosyncrasy (4/5)

Special symbols, Greek letters. Can be written by typing, for example `\alpha + tab` and choosing the option you want



- ☺ Emojis can be selected after typing “Win + .” and can be used as variables and function names

- Composition of functions, written by typing “`\circ + tab`”, as in `(sin ∘ cos)(2)`

`md"># ...` Markdown block, Pluto will print a non-code text above the cell of different size depending on how many “#s” are used, as in `md### Set Operating Point for calculations ↗` “

Set Operating Point for calculations ↗  
- `md### Set Operating Point for calculations ↗`

`==` Is equal? Note that `=` is used for assignment; `3 == 3` -> true    `a = 3`; `a -> 3`

`!=` not equal

`≈` “Is approximately equal” (`\approx + tab`). Used to compare numbers (particularly floating point) ignoring the machine truncation error

# Special symbols and idiosyncrasy (5/5)

\$ Denotes that the variable value shall be interpolated in a string (and also in the input to some macros like `@btime`)

A screenshot of a terminal window. The code is:

```
v = 3
  . v = 3
  .
  . print("The value of v is = $v")
> The value of v is = 3
```

The output is displayed in a dark-themed terminal window.

|> “Pipe operator”, very popular in functional programming. `|>(x, f)` or `x |> f` means `f(x)` and it allows composition or concatenation of functions with a simple syntax (like a pipe), as in `f(g(x))` written as `x |> g |> f`

`f(x) = x^2 ; g(x) = x/2`      `; 4 |> f |> g = 8`      `; |>(4,f) = 16`

... “Splat operator” ..., represents a sequence of arguments. ... can be used in function definitions, to indicate that the function accepts an arbitrary number of arguments. ... can also be used to apply a function to a sequence of arguments as in

`add([1, 2, 3]...)`

=> Is used in Dictionaries to assign values to keywords as in `Dict("A"=>1, "B"=>2)`

# Warming up



# Intermediate topics (MIT Computational Thinking)

A fantastic learning resource is the “Computational Thinking” course from MIT

<https://computationalthinking.mit.edu/Fall23/>

The screenshot shows the MIT Computational Thinking course website. On the left, there's a navigation sidebar with links like "Introduction to Computational Thinking", "Choose your track:", "WELCOME", "Software installation", "Class reviews", "Class logistics", "Cheatsheets", "Previous semesters", and "MODULE 1: IMAGES, TRANSFORMATIONS, ABSTRACTIONS". A red box highlights the "1.1 Images as Data and Arrays" link under the module section. Below the sidebar, there's a "Highlights" section with a "Real-world problems" card featuring a graph of "Global CO<sub>2</sub> emissions" over time.

The main content area has a large banner for "Julia: A Fresh Approach to Computing". Below it, a box states: "This class uses revolutionary programmable interactivity to combine material from three fields creating an engaging, efficient learning solution to prepare students to be sophisticated and intuitive thinkers, programmers, and solution providers for the modern interconnected online world." Another box below says: "Upon completion, students are well trained to be scientific ‘trilinguals’, seeing and experimenting with mathematics interactively as much as meant to be seen, and ready to participate and contribute to open source development of large projects and ecosystems."

The central part of the page features a "Section I.1" header with a "Lecture Video" thumbnail showing a "MIT x COMPUTATIONAL THINKING Spring 2021" video. To the right, there's a "Edit or run this notebook" button with a hand cursor pointing at it. A blue line connects this button to a "binder" button in a sidebar on the right. The sidebar also contains text about running the notebook in the cloud (experimental) and on a computer, with a "Copy the notebook URL:" link and a hand cursor pointing at it.

**Section I.1**  
Edit or run this notebook  
This notebook takes about 3 minutes to run.  
**In the cloud (experimental)**  
**binder**  
Binder is a free, open source service that runs scientific notebooks in the cloud! It will take a while, usually 2-7 minutes to get a session.  
**On your computer**  
(Recommended if you want to store your changes.)  
1. Copy the notebook URL:  
<https://computationalthinking.mit.edu/Fall23/>

**Images as Data and Arrays**  
Real-world problems  
Global CO<sub>2</sub> emissions

**Initialize packages**  
When running this notebook for the first time, this could take up to 15 minutes. Hang in there!

```
begin
    using Colors, ColorVectorSpace, ImageShow, FileIO, ImageIO
    using PlutoUI
    using PlutoTeachingTools
    using HypertextLiteral: @htl, @htlattr
end
```

**Images as examples of data all around us**  
Welcome to the Computational Thinking using Julia for Real-World Problems, at MIT in Spring 2021!

**Click on a lecture, edit the notebook, copy the url and paste it into the “Open notebook” cell in Pluto**

# MIT Computational Thinking key lectures (intermediate level)

Below is a selection of memorable lectures with links to the lecture page (containing a video) and the interactive notebooks



## 1.1- Images as data and Arrays



[https://computationalthinking.mit.edu/Fall23/generated\\_assets/hw1\\_395530a6.jl](https://computationalthinking.mit.edu/Fall23/generated_assets/hw1_395530a6.jl)



## 1.4- Transformations with images



[https://computationalthinking.mit.edu/Fall23/generated\\_assets/transforming\\_images\\_b0baefaa.jl](https://computationalthinking.mit.edu/Fall23/generated_assets/transforming_images_b0baefaa.jl)



## 2.9 Optimization



[https://computationalthinking.mit.edu/Fall23/generated\\_assets/optimization\\_5abd7af7.jl](https://computationalthinking.mit.edu/Fall23/generated_assets/optimization_5abd7af7.jl)



## 3.2 Differential equations



[https://computationalthinking.mit.edu/Fall23/generated\\_assets/odes\\_and\\_parameterized\\_types\\_b2f33ff1.jl](https://computationalthinking.mit.edu/Fall23/generated_assets/odes_and_parameterized_types_b2f33ff1.jl)



## 3.6 Snowball Earth and Hysteresis (excellent video)



[https://computationalthinking.mit.edu/Fall23/generated\\_assets/climate2\\_snowball\\_earth\\_bffbd696.jl](https://computationalthinking.mit.edu/Fall23/generated_assets/climate2_snowball_earth_bffbd696.jl)

# Interactive notebooks for education



[https://github.com/flt-acdesign/Low\\_speed\\_AC\\_performance/blob/main/21\\_04\\_08\\_Low\\_speed\\_perfo\\_v\\_0.0.1.jl](https://github.com/flt-acdesign/Low_speed_AC_performance/blob/main/21_04_08_Low_speed_perfo_v_0.0.1.jl)

This Pluto notebook is a simple example of an interactive resource to teach aircraft performance.

The students are expected to extend the code to plot additional graphics

Below is an example of how to document functions in Julia so that Pluto can provide “Live Docs”. Some of the plots are interesting to explore too.

## Aircraft Aerodynamic functions

```
begin
    # Aircraft aerodynamic coefficients, drag, power required and helper functions

    # NOTE: the text below, with exactly the format used, corresponds to the Julia
    # "docstrings" standard. The documentation needs to be exactly on the line above the
    # function definition. the "live docs" button at the bottom right of Pluto will show
    # the documentation of the function when the cursor is over the function name anywhere
    # in the code

    #
    """
    Vs1gTAS(W, h, CLmax, Sw)

    Calculate stall speed as TAS at 1g from weight (W) in Newtons, altitude (h) in
    meters, aircraft maximum lift coefficient (CL) and wing reference area (Sw) in m^2

    # Examples
    """julia-repl
    julia> Vs1gTAS(80000, 4000, 2.1, 20)
    48.241248922681834
    """
    """
    Vs1gTAS(W, h, CLmax, Sw) = ((W)/(ρ(h)*CLmax*Sw))^0.5

    #
    """
    Vs1gEAS(W, CLmax, Sw)
```

Q Live Docs ⌂ Status

Vs1gTAS

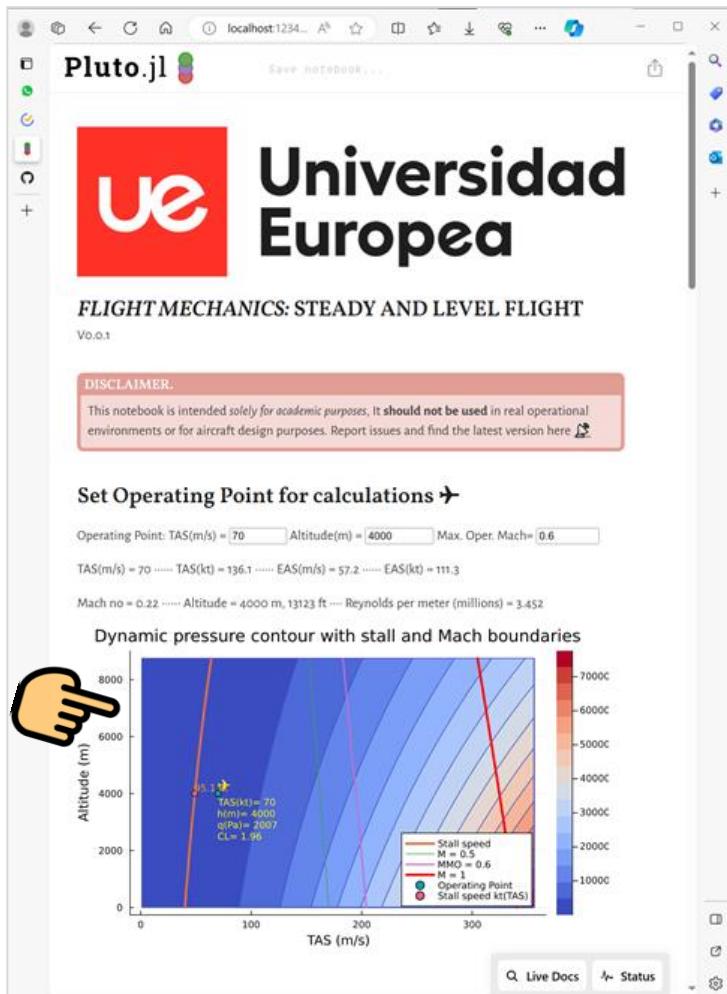
Vs1gTAS(W, h, CLmax, Sw)

Calculate stall speed as TAS at 1g from weight (W) in Newtons, altitude (h) in meters, aircraft maximum lift coefficient (CL) and wing reference area (Sw) in m<sup>2</sup>

Examples

```
julia> Vs1gTAS(80000, 4000, 2.1, 20)
48.241248922681834
```

Check the  
code for  
this plot for  
inspiration



# Enter chatGPT (or Bing chat or...)

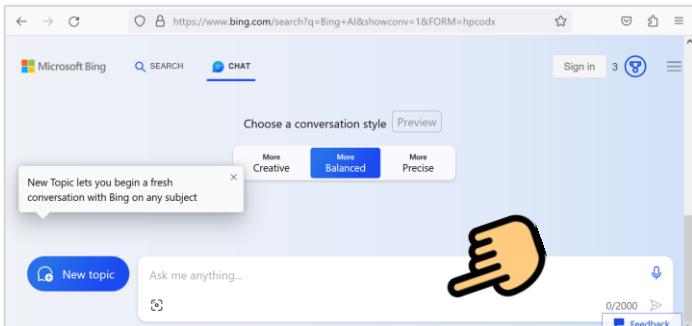
The world will never be the same since the advent of ChatGPT (and many other Large Language Models since).

Although probably the most robust automatic code generation occurs for Python, you can use ChatGPT to generate code in Julia

However, be prepared to edit the code manually or keep asking for refinements (“Regenerate”) until it works. **Note that you may get different answers every time as the system keeps on “learning”!**

The link below opens the Bing chat, which is not too bad for code generation and is completely free

<https://www.bing.com/search?q=Bing+AI&showconv=1&FORM=hpcodx>



This is called the “prompt” and should always start with “please” and be concrete, concise and cover all the points for which you need an answer in a logical way

A screenshot of the Microsoft Bing Chat interface. At the top, there are "SEARCH" and "CHAT" buttons, with "CHAT" being the active tab. On the right, a user profile shows "RAULCARLO..." and some icons. In the main area, a user prompt is shown in a blue box: "Please write a julia code to plot the mandelbrot set". Below this, a green checkmark indicates "Searching for: julia code mandelbrot set" and another green checkmark indicates "Generating answers for you...". A large text box displays the generated Julia code: 

```
Julia
AI-generated code. Review and use carefully. More Info on FAQ

using Plots

function mandelbrot(z)
    c = z
    for i in 1:100
        if abs2(z) > 4
            return i-1
        end
        z = z^2 + c
    end
    return 100
end

x = range(-2, 2, length=800)
y = range(-2, 2, length=800)
heatmap(x, y, mandelbrot.(x .+ y'))
```

 Below the code, a note explains its purpose: "This code uses the Plots package to create a heatmap of the Mandelbrot set. The mandelbrot function calculates the number of iterations required for a given complex number  $z$  to escape the circle of radius 2. The heatmap function then plots the number of iterations required for each point in the complex plane." At the bottom, a message says "I hope this helps! Let me know if you have any other questions."

This code does not create the Mandelbrot set, ask ChatGPT to rewrite it using complex numbers...

# Mandelbrot set – Raw GPT3.5 code

The code below has been generated automatically by GPT3.5 (the free version of ChatGPT). Copy it into a cell of a new Pluto notebook and click Shift+ENTER

```
using Plots

function mandelbrot(c, max_iter)
    z = c
    for i in 1:max_iter
        if abs2(z) > 4.0
            return i - 1
        end
        z = z * z + c
    end
    return max_iter
end

function plot_mandelbrot(x_center, y_center, zoom, width, height, max_iter)
    x_min = x_center - 0.5 / zoom
    x_max = x_center + 0.5 / zoom
    y_min = y_center - 0.5 / zoom
    y_max = y_center + 0.5 / zoom

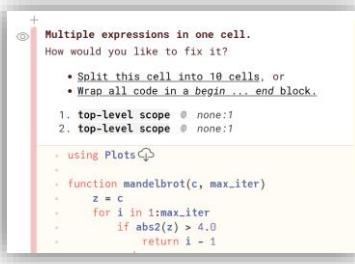
    x = LinRange(x_min, x_max, width)
    y = LinRange(y_min, y_max, height)

    mandelbrot_set = [mandelbrot(complex(r, i), max_iter) for r in x, i in y]

    heatmap(x, y, mandelbrot_set, c=:viridis, xlabel="Re", ylabel="Im", aspect_ratio=:equal)
end

# Set parameters
x_center = -0.5
y_center = 0.0
zoom = 1.0
width = 800
height = 800
max_iter = 100

# Plot Mandelbrot set
plot_mandelbrot(x_center, y_center, zoom, width, height, max_iter)
```



When you get this message in Pluto, click on “Split this cell into...”

Default (GPT-3.5)

UE please write a julia code to plot the mandelbrot set with center in coordinates x,y and a zoom level of z

Certainly! You can use the 'Plots' library in Julia to create a plot of the Mandelbrot set with a specified center and zoom level. First, make sure you have the 'Plots' package installed by running `import Pkg; Pkg.add("Plots")` if you haven't done so already.

Here's a simple Julia code to generate and plot the Mandelbrot set:

```
using Plots

function mandelbrot(c, max_iter)
    z = c
    for i in 1:max_iter
        if abs2(z) > 4.0
            return i - 1
        end
        z = z * z + c
    end
    return max_iter
end

function plot_mandelbrot(x_center, y_center, zoom, width, height, max_iter)
    x_min = x_center - 0.5 / zoom
    x_max = x_center + 0.5 / zoom
    y_min = y_center - 0.5 / zoom
    y_max = y_center + 0.5 / zoom

    x = LinRange(x_min, x_max, width)
    y = LinRange(y_min, y_max, height)

    mandelbrot_set = [mandelbrot(complex(r, i), max_iter) for r in x, i in y]

    heatmap(x, y, mandelbrot_set, c=:viridis, xlabel="Re", ylabel="Im", aspect_ratio=:equal)
end

# Set parameters
x_center = -0.5
y_center = 0.0
zoom = 1.0
width = 800
height = 800
max_iter = 100

# Plot Mandelbrot set
plot_mandelbrot(x_center, y_center, zoom, width, height, max_iter)
```

This code defines a 'mandelbrot' function to calculate the Mandelbrot set's iteration count for a given complex number 'c', and a 'plot\_mandelbrot' function to create a plot.

Click to copy the code

If you don't like the code, regenerate it

# A refined version of Mandelbrot starting from GPT code

Open this code in Pluto



[https://github.com/flt-acdesign/Mathematical\\_tourism\\_with\\_Julia/blob/main/mandelbrot\\_zoom\\_refined.jl](https://github.com/flt-acdesign/Mathematical_tourism_with_Julia/blob/main/mandelbrot_zoom_refined.jl)

```
using Plots ✓, PlutoUI ✓ # Load required packages (For plotting and interactivity)
1000
begin
    const width = 600 # Number of pixels in width of image
    const aspect_ratio::Float64 = 9/16 # Aspect ratio (defined inversely as normal)
    const height = Int64(round(width * aspect_ratio)) # Image height as integer value
    const max_iter = 1000 # Maximum number of iterations for the Mandelbrot equation
end

mandelbrot (generic function with 1 method)
# Mandelbrot algorithm for a complex point "c" with Re and Im components of type
# BigFloat in order to capture "arbitrarily" small numbers
function mandelbrot(c::Complex{BigFloat}, max_iter::Int)
    z = c # start at the point from the complex plane fed to the function
    for i in 1:max_iter # Loop up to max_iter (it may not get there...)
        if abs2(z) > 4.0 # If the length of the vector exceeds 4
            return i - 1 # exit the function returning the previous iteration number
        end
        #z = z ^2 + c # z * z changed to z^2 for readability
        z = muladd(z, z, c) # Calling BLAS library operators (25% faster than above)
    end
    return max_iter # If the iterative process does not diverge, return max_iter
end
```

**using** makes all the functions of the packages available in this program. Adding a package is like buying the book, using is like reading it

Using **const** before the value of a variable makes it a constant (it cannot be changed during the execution of the code) and, more importantly, it stores the value “closer” to the CPU, which is useful to improve speed of the code if the value is called millions of times (not this case).

The **mandelbrot** function takes a complex number  $x$  and the maximum number of iterations in the algorithm. The complex number has real and imaginary components of type “**BigFloat**”, which in Julia represents a number of arbitrary precision (a large number of bits, user defined). This is required to zoom into extremely small windows, smaller than about  $10^{-16}$

The **muladd** function multiplies  $z$  by itself (in this case) and adds  $c$ , storing the result in  $z$ . This is a “non allocating” function and is faster as it does not require memory access. In this case, given the number of times this operation is performed, it is worth it as it gives a 25% increase in speed.

# A refined version of Mandelbrot

```
generate_mandelbrot (generic function with 1 method)
  • # This function calls the Mandelbrot algorithm for each point in the rectangle of the
    # complex plane defined by its center coordinates and a "delta" ( $\delta$ ) on each side of the
    # center. Center and delta use the BigFloat type for arbitrary precision

  • function generate_mandelbrot(x_center::BigFloat,
    y_center::BigFloat,
    δ::BigFloat,
    width, aspect_ratio::Float64, max_iter)

    # Generate ranges of x and y coordinates
    x = LinRange(x_center - δ, x_center + δ, width)
    y = LinRange(y_center - δ * aspect_ratio, y_center + δ * aspect_ratio,
      Int(round(width * aspect_ratio)))

    """
    Matrix comprehension method
    mandelbrot_set = [mandelbrot(complex(BigFloat(r), BigFloat(i)), max_iter) for r in x,
    i in y]
    """

    # Explicit CPU multithreaded loop
    mandelbrot_set = zeros(length(x), length(y)) #initialize empty array to hold result

    # Loop for all the points in the rectangle x,y.
    # The "@inbounds" macro disables the internal checking of out of bounds in the arrays
    # The "@Threads" macro breaks down the loops into smaller loops sent to each core

    @inbounds Threads.@threads for index_r in 1:length(x)
      @inbounds Threads.@threads for index_i in 1:length(y)
        mandelbrot_set[index_r, index_i] = mandelbrot(
          complex(BigFloat(x[index_r]), BigFloat(y[index_i])), max_iter)
      end
    end

    return mandelbrot_set # This function returns an array with the number of iterations
    # before divergence for each point in the domain of the complex plane under study
  end
```

This function also takes **BigFloat**, it is important to specify the types of the inputs to all functions for the highest performance in Julia. This is called having “type stability”. Otherwise Julia will infer the types from the values, but in some cases it will not be able and will use “Any” which takes a lot of memory and is very slow.

The LinRange function generates a collection of elements equally spaced between the first and second input value, with the third input value stating how many elements are required. Type \delta+tab to get a  $\delta$

A “matrix comprehension” generates the elements of an array by applying a function to an iterator as in `[i^2 for i in 1:4]`, which creates an array of squares of all the integers between 1 and 4. It is not used here because we will generate the array of complex points using multithreading, which can be faster

The **@inbounds** macro tells Julia to not check whether the indices of the arrays are within the array limits. In this case we are sure and we can disable, if we are not sure and there is a problem we can crash the program due to memory overrun

The **Threads.@treads** macro tells Julia to break the loop into as many processes as threads are available in the machine. Simple loops are [“embarrassingly parallelizable”](#) and are a good candidate for multithreading.

# A refined version of Mandelbrot

```
Set zoom 10^zoom =  -0.4
· md"Set zoom 10^zoom = $($@bind zoom Slider(-.6:.01:7 ;default=-.4, show_value=true) ) "
6 = 2.511886431509580130949643717030994594097137451171875
· δ = BigFloat(1/(10^zoom)) # Convert the "zoom" into the width of the interval

xcenter =  -1.75 ycenter =  0.0
·
· md"
· xcenter = $($@bind x_center Slider(-3:.01:1; default = xcvideo , show_value=true) )
· ycenter = $($@bind y_center Slider(-1:.01:1; default = ycvideo , show_value=true) )
· "
108.18144970414201
· begin
·
· #setprecision(BigFloat, 500) # Enable this line to set arbitrary precision on the
BigFloat type. Default value is 128 bits.
·
· # The array to be plotted is the return value of the generate_mandelbrot function,
note that the array needs to be transposed in order to obtain the usual orientation
of the Mandelbrot set
array_to_plot = transpose(generate_mandelbrot(
·
·
·
·
·
BigFloat(x_center),
BigFloat(y_center),
δ,
width, aspect_ratio, max_iter))

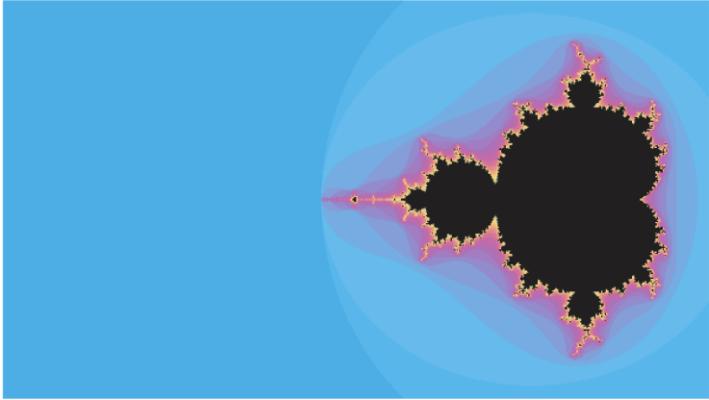
· # Calculate the average value of the array with the number of iterations. This is
used to set ad-hoc a color scale for plotting
· average = (sum(array_to_plot)/length(array_to_plot))
·
· end
```

This code creates a slider which can assign a value to the variable `zoom` in the range defined by -0.6 to 7 in steps of 0.1, and it shows the value next to the slider. You need the [PlutoUI](#) package for this

The `setprecision(BigFloat, 500)` is commented to improve performance for normal levels of zoom. When you zoom a lot you may need higher floating point precision and the previous statement sets the representation of floating point numbers at 500 bits (normally in Julia they have 64, as in [Float64](#)). The default number of bits for `BigFloat` is 128 bits.

Here you can see how flexible the Julia parser is to spaces and newlines. This can help a lot to make the code clearer).

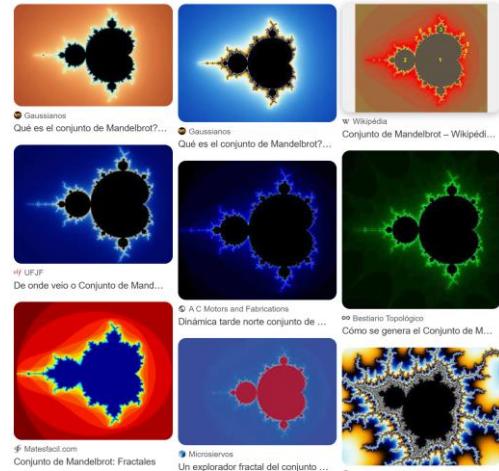
# A refined version of Mandelbrot



```
begin
    image = heatmap( # Assign the result of a "heatmap" plot to "image"
        array_to_plot,
        c=:cmyk, # Color scheme
        clim=(0, average /4), # Min and max values of the color range
        aspect_ratio=1,
        ticks=false,
        showaxis=false,
        legend=:none,
        size =(width, Int64(round(width * aspect_ratio))) # Image size
        #dpi=1200
    )

    savefig("mandelbrot_heatmap.png") # Save the current plot in a png file
    image
end
```

And here it is in a small part of its beauty... compare it with some of the results from other people. The color schemes are totally arbitrary and their selection comes down to aesthetics.



The image variable now holds the complete **heatmap** picture. A heatmap is a color picture, as shown above. It can be used to plot matrices as images. The parameters passed to the heatmap function specify various options, including the color scheme.

Go to the Plots documentation to see other [color schemes](#). Change the scheme **:cmyk** for others, for example **:flag**

This saves the image in .png format

# A refined version of Mandelbrot

```
. #save_animation() # Invoke this function to write a gif file with an animation

save_animation (generic function with 1 method)
  .# Create an animation with various images constructed with the Mandelbrot algorithm
  # for variations of one parameter, in this case, the zoom value
  .
  .function save_animation()
  .
  .# The @animate macro loops with the generation of the images
  anim_evolution = @animate for zoom in -.7:1:30
  .
  .array_to_plot = transpose(generate_mandelbrot( # Calculate the array to plot
  .                                BigFloat(x_center),
  .                                BigFloat(y_center),
  .                                BigFloat(1/(10^zoom)),
  .                                width, aspect_ratio, max_iter))
  .
  .heatmap( # Generate a "heatmap" image of the array to be plotted
  .        array_to_plot,
  .        c=:cmyk,
  .        clim=(0, average/5),
  .        aspect_ratio=1,
  .        ticks=false,
  .        showaxis=false,
  .        legend=:none,
  .        dpi=1200)
  .
  end
  .
  gif(anim_evolution, "mandelbrot_anim.gif", fps = 10) # Save the animation as .gif
  .
end
```

Remove the comment in this line to save the animation of a zoom into the set as a gif file

The animation is produced by the @animate macro which is an implicit loop on the variable zoom, in the range -0.7 to 30 in steps of 1. Note that a zoom of 30 means that you have amplified the image by 30 orders of magnitude. With normal BigFloat numbers (128 bits) you can zoom about 70 orders of magnitude, for higher zooms you need to increase the number of bits.

For more information on the Mandelbrot set watch:

[Numberphile](#)

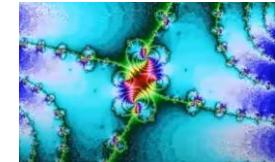
[Three blue one brown \(must watch\)](#)

[Veritasium \(excellent\)](#)

The initial coordinates are specified here and are taken from the video description of this YouTube video:



[Hardest Mandelbrot Zoom in 2017 - 750 000 000 iterations! - YouTube](#)



Coordinates from this video [https://www.youtube.com/watch?v=aSg2Db3jF\\_4](https://www.youtube.com/watch?v=aSg2Db3jF_4)

```
. begin
.   # Default coordinates in the complex plane of the center of the image
.   xcvideo =
.   BigFloat(-1.7499576837060935036022145060706997072711057972625207793024283782028600
.   8082972804887218672784431700831100544507655659531379747541999999999
.
.   yctvideo =
.   BigFloat(0.0000000000000000278793706563379402178294753790944364927085054500163081
.   37904393065018938684976520216947747055220132577233245472699999999
.
.   md" Coordinates from this video https://www.youtube.com/watch?v=aSg2Db3jF\_4"
```

# Lorenz Attractor – Euler method

[Visualizing the Lorenz Attractor in Julia | by Daryl Tng | Medium](#)

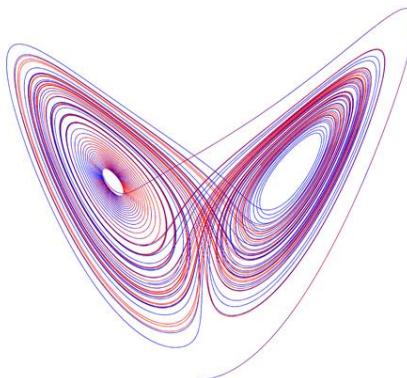
The Lorenz system of equations is very famous in mathematical and popular culture. It opened the field of Chaos which loosely means that infinitesimal variations of the initial conditions will give rise to totally different behaviours of a “chaotic” system (of which there are many, starting by the atmosphere, which is the origin of this system).

The phase plot of the Lorenz system is the “Lorenz attractor” which is created an popular association of chaos to the “[butterfly effect](#)” (“*Does the flap of a butterfly in Brazil set off a tornado in Texas?*”)

The Lorenz system

$$\begin{aligned}\frac{dx}{dt} &= \sigma y - \sigma x, \\ \frac{dy}{dt} &= \rho x - xz - y, \\ \frac{dz}{dt} &= xy - \beta z.\end{aligned}$$

The 3 governing equations of the Lorenz system



[Lorenz system – Wikipedia](#) with a Julia code!

Please write a julia code to plot and rotate the lorenz attractor using the euler method

Certainly! Below is a simple Julia code to plot and rotate the Lorenz attractor using the Euler method. The code uses the “Plots” library for plotting.

```
julia
using Plots

function lorenz!(du, u, p, t)
    σ, ρ, β = p
    du[1] = σ * (u[2] - u[1])
    du[2] = ρ * u[1] * (u[3] - u[2])
    du[3] = u[1] * u[2] - β * u[3]
end

function euler!(u, h, p, t_end)
    t = 0.0:h:t_end
    traj = zeros(length(t), length(u))

    for (i, t_i) in enumerate(t)
        traj[i, :] = u
        du = zeros(size(u))
        lorenz!(du, u, p, t_i)
        u += h * du
    end

    return t, traj
end

function plot_lorenz(t, traj)
    plot3d(
        traj[:, 1], traj[:, 2], traj[:, 3],
        xlabel="X", ylabel="Y", zlabel="Z",
        title="Lorenz Attractor",
        legend=false,
    )
end

# Set parameters
σ = 10.0
ρ = 28.0
β = 8/3

# Initial conditions
u₀ = [1.0, 0.0, 0.0]

# Time parameters
h = 0.01
t_end = 100.0

# Perform Euler integration
t, traj = euler!(u₀, h, [σ, ρ, β], t_end)

# Plot the Lorenz attractor
plot_lorenz(t, traj)
```

Make sure you have the “Plots” library installed by running “`using Pkg; Pkg.add("Plots")`” in the Julia REPL if you haven’t installed it already.

You can adjust the parameters such as “`σ`”, “`ρ`”, “`β`”, “`u₀`”, “`h`”, and “`t_end`” to explore different behaviors of the Lorenz attractor.

# Lorenz Attractor – Euler method

Copy the code into a new Pluto notebook



[https://github.com/flt-acdesign/Mathematical\\_tourism\\_with\\_Julia/blob/main/Lorenz\\_Euler\\_GOOD.jl](https://github.com/flt-acdesign/Mathematical_tourism_with_Julia/blob/main/Lorenz_Euler_GOOD.jl)

```
• using Plots ✓

lorenz! (generic function with 1 method)
• function lorenz!(du, u, p, t)
•     σ, ρ, β = p
•     du[1] = σ * (u[2] - u[1])
•     du[2] = u[1] * (ρ - u[3]) - u[2]
•     du[3] = u[1] * u[2] - β * u[3]
• end
```

```
euler! (generic function with 1 method)
• function euler!(u, h, p, t_end)
•     t = 0.0:h:t_end
•     traj = zeros(length(t), length(u))
•
•     for (i, ti) in enumerate(t)
•         traj[i, :] = u
•         du = zeros(size(u))
•         lorenz!(du, u, p, ti)
•         u += h * du
•     end
•
•     return t, traj
• end
```

```
plot_lorenz (generic function with 1 method)
• function plot_lorenz(t, traj)
•     plot3d(
•         traj[:, 1], traj[:, 2], traj[:, 3],
•         xlabel="X", ylabel="Y", zlabel="Z",
•         title="Lorenz Attractor",
•         legend=false,
•     )
• end
```

We'll solve this problem with pretty basic Julia, we just need Plots for plotting the attractor (Lorenz did not have that luxury in the 60's)

The system of non-linear differential equations is isolated in this function. The state of the system is stored in a vector  $u[3]$ .  $du[3]$  corresponds to the vector  $\frac{du}{dt}$ .

Note how the parameters of the differential equation are inferred from a vector  $p$

The `euler!` function is mutating, this means that it takes the state vector and operates on it "in place" (without memory allocations). This is important for performance but it's not pure functional programming. However there is no merit in pure almost anything if you have good reasons not to be pure...

The returned value is a "tuple", a group of two values: the time array (a collection) and an array with the trajectory in phase space.

This function plots the phase state in 3D by providing a single array for each component of the trajectory.

# Lorenz Attractor – Euler method

```
o = 10.0
· # Set parameters
· o = 10.0
```

```
p = 28.0
· p = 28.0
```

```
β = 2.6666666666666665
· β = 8/3
```

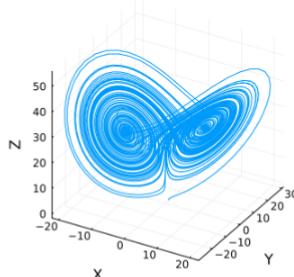
```
u0 = [1.0, 0.0, 0.0]
· # Initial conditions
· u0 = [1.0, 0.0, 0.0]
```

```
h = 0.01
· # Time parameters
· h = 0.01
```

```
t_end = 100.0
· t_end = 100.0
```

```
► (0.0:0.01:100.0, 10001×3 Matrix{Float64};)
· # Perform Euler integration
· t, traj = euler!(u0, h, [o, p, β], t_end)
```

Lorenz Attractor



```
· # Plot the Lorenz attractor
· plot_lorenz(t, traj)
```

This part of the code sets the initial value and parameters of the system and is written in a not very elegant way because Pluto has expanded each line into its own cell.

A better way is to group all these cells into a block of code

This cell assigns to t and traj the result of the euler! Function call with the parameters defined above, containing the phase trajectory.

And this is the plot of the Lorenz attractor.

For more information watch:

[Chaos | Chapter 7 : Strange Attractors - The butterfly effect – YouTube](#)

# Lotka-Volterra differential equations using Euler method

Open this code in Pluto



[https://github.com/flt-acdesign/Mathematical\\_tourism\\_with\\_Julia/blob/main/Lotka\\_Volterra\\_Euler\\_GOOD](https://github.com/flt-acdesign/Mathematical_tourism_with_Julia/blob/main/Lotka_Volterra_Euler_GOOD)

```
using Plots ✓

lotka_volterra_euler (generic function with 1 method)
• function lotka_volterra_euler(α, β, γ, δ, u₀, T, dt)
•     t_values = 0:dt:T
•     u_values = zeros(length(t_values), length(u₀))
•     u_values[1, :] .= u₀
•
•     for i in 2:length(t_values)
•         du = lotka_volterra(u_values[i-1, :], α, β, γ, δ)
•         u_values[i, :] = u_values[i-1, :] + dt * du
•     end
•
•     return t_values, u_values
• end

lotka_volterra (generic function with 1 method)
• function lotka.volterra(u, α, β, γ, δ)
•     du1 = α * u[1] - β * u[1] * u[2]
•     du2 = δ * u[1] * u[2] - γ * u[2]
•
•     return [du1, du2]
• end

▶ (0.1, 0.02, 0.1, 0.01)
• # Parameters
• α, β, γ, δ = 0.1, 0.02, 0.1, 0.01

u₀ = ▶ [100.0, 20.0]
• # Initial conditions
• u₀ = [100.0, 20.0]

T = 1000.0
• # Time span and step size
• T = 1000.0

dt = 0.1
• dt = 0.1

▶ (0.0:0.1:1000.0, 10001×2 Matrix{Float64}):
    100.0      20.0
• # Solve using Euler method
• t_values, u_values = lotka.volterra_euler(α, β, γ, δ, u₀, T, dt)
```

This code has been created automatically by ChatGPT and works right out of the box, probably because the Lotka-Volterra problem is very well covered in examples and the AI had plenty of opportunity to train.

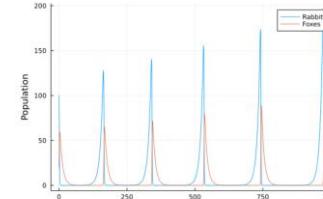
Here it was explicitly asked to use the Euler method (to prevent GPT from proposing packages for differential equations, which we will see later).

The **Lotka–Volterra equations**, also known as the **Lotka–Volterra predator–prey model**, are a pair of first-order **nonlinear differential equations**, frequently used to describe the **dynamics of biological systems** in which two species interact, one as a **predator** and the other as prey. The populations change through time according to the pair of equations:

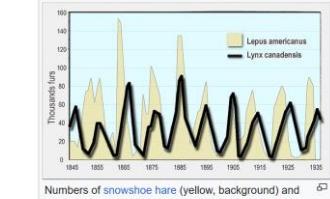
$$\begin{aligned}\frac{dx}{dt} &= \alpha x - \beta xy, \\ \frac{dy}{dt} &= \delta xy - \gamma y,\end{aligned}$$

where

- the variable  $x$  is the **population density** of prey (for example, the number of **rabbits** per square kilometre);
- the variable  $y$  is the population density of some **predator** (for example, the number of **foxes** per square kilometre);
- $\frac{dy}{dt}$  and  $\frac{dx}{dt}$  represent the instantaneous growth rates of the two populations;
- $t$  represents time;
- The prey's **parameters**,  $\alpha$  and  $\beta$ , describe, respectively, the maximum prey **per capita** growth rate, and the effect of the presence of predators on the prey growth rate.
- The predator's parameters,  $\gamma$ ,  $\delta$ , respectively describe the predator's **per capita** death rate, and the effect of the presence of prey on the predator's growth rate.



```
# Plot the results
plot(t_values, u_values, label=["Rabbits", "Foxes"], xlabel="Time",
      ylabel="Population", legend=:topright)
```



Numbers of snowshoe hare (yellow, background) and Canada lynx (black line, foreground) furs sold to the Hudson's Bay Company. Canada lynxes eat snowshoe hares.

[Lotka–Volterra equations - Wikipedia](#)