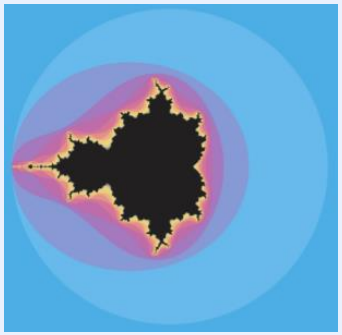
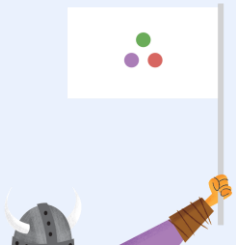


An introduction to the julia language for researchers

Universidad de Sevilla Oct 2023

Raul Llamas

raulcarlos.llamas@universidadeuropea.es



Introduction

Course repository in Github:

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?

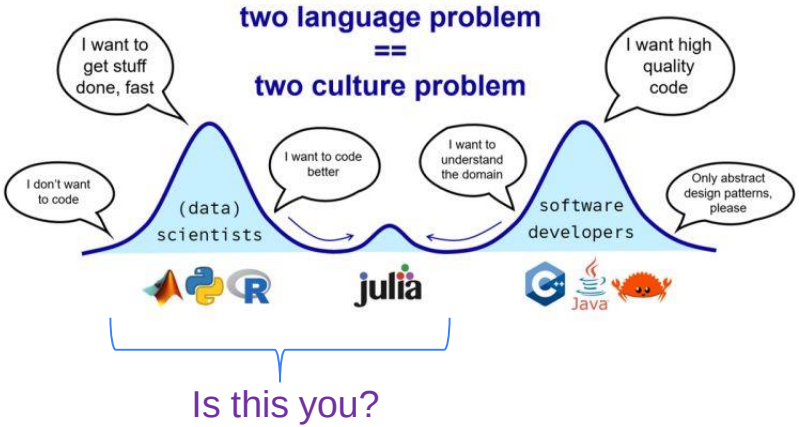
What makes Julia so awesome?

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** do 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

Here be dragons

Dall-e

2009 — 2012 — 2015 — 2017 — **2018** — **2023**

Julia enters the “petaflops” club

State of Julia

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

Work on **Julia** was started in 2009, by Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman

Blogpost explaining the mission of Julia

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

Julia 1.0 released

Julia gets 20th position in the Tiobe index

Alan Edelman and his dog, an important contributor to Julia, it seems

Aug 2023	Aug 2022	Change	Programming Language	Ratings	Change
1	1		Python	13.33%	-2.30%
2	2		C	11.41%	-3.35%
3	4	↑	C++	10.63%	+0.69%
4	3	↓	Java	10.33%	-2.14%
5	5		C#	7.04%	+1.64%
6	8	↑	JavaScript	3.29%	+0.89%
7	6	↓	Visual Basic	2.63%	-2.26%
8	9	↑	SQL	1.53%	-0.14%
9	7	↓	Assembly language	1.34%	-1.41%
10	10		PHP	1.27%	-0.09%
11	21	⇈	Scratch	1.22%	+0.63%
12	15	↑	Go	1.16%	+0.20%
13	17	⇈	MATLAB	1.05%	+0.17%
14	18	⇈	Fortran	1.02%	+0.24%
15	31	⇈	COBOL	0.96%	+0.59%
16	16		R	0.92%	+0.01%
17	19	↑	Ruby	0.91%	+0.18%
18	11	↓	Swift	0.90%	-0.35%
19	22	↑	Rust	0.89%	+0.32%
20	28	⇈	Julia	0.85%	+0.41%

Julia is completely free and open source

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

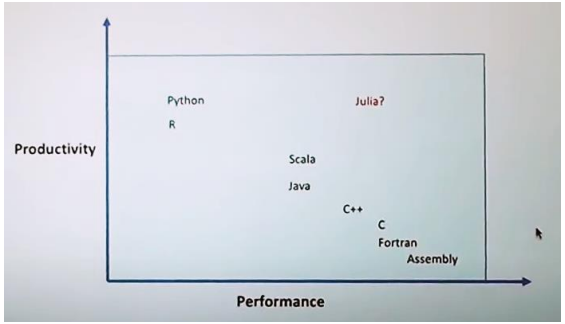
Julia in 100 seconds

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.



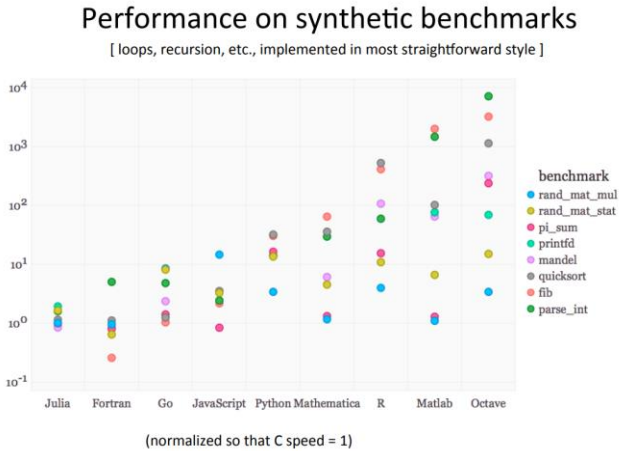
From the Celeste project

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).



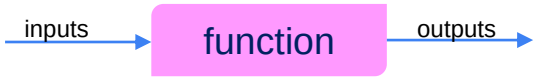
THE UGLY

Its name *(for a computer language)*

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: Matlab Python Julia

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

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))
```

N/A similar type

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

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

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

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.

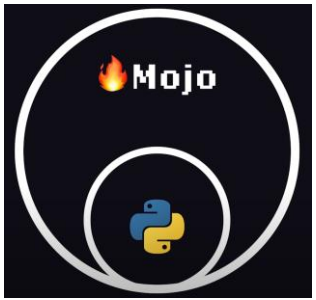
Today the “safe choice” is Python but Julia was slated to be the “Python killer” due to its performance (about 100 times faster than Python) and its elegance.

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

Julia & Mojo Mandelbrot Benchmark - General Usage - Julia Programming Language (julialang.org)

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

Your call…

Julia

```
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


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

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

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

Mojo

```
from benchmark import Benchmark
from complex import ComplexSIMD, ComplexFloat64
from math import iota
from python import Python
from runtime.llcl 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, 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()
```

```
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 oeprate 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_store[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())

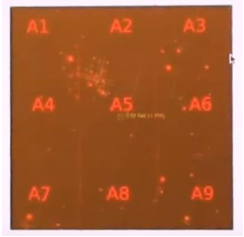
        alias simd_width = simdwidthof[DType.float64]()
        let parallelized = Benchmark().run[bench_parallel[simd_width]]() / 1e6
        print("Parallelized:", parallelized, "ms")

    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



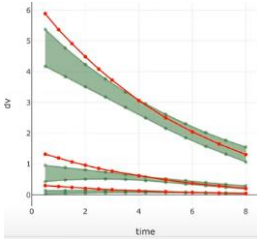
Catalogue of 180 million stars and galaxies in the visible universe (created in 15 minutes with 9000 nodes, 100_000 cores)

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

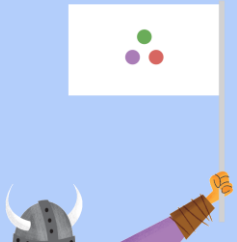
There’s something going on between Boeing and Julia…

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

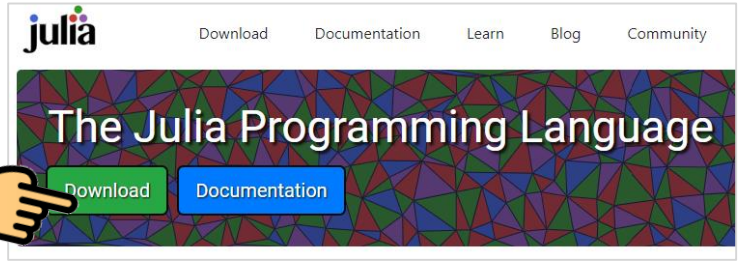


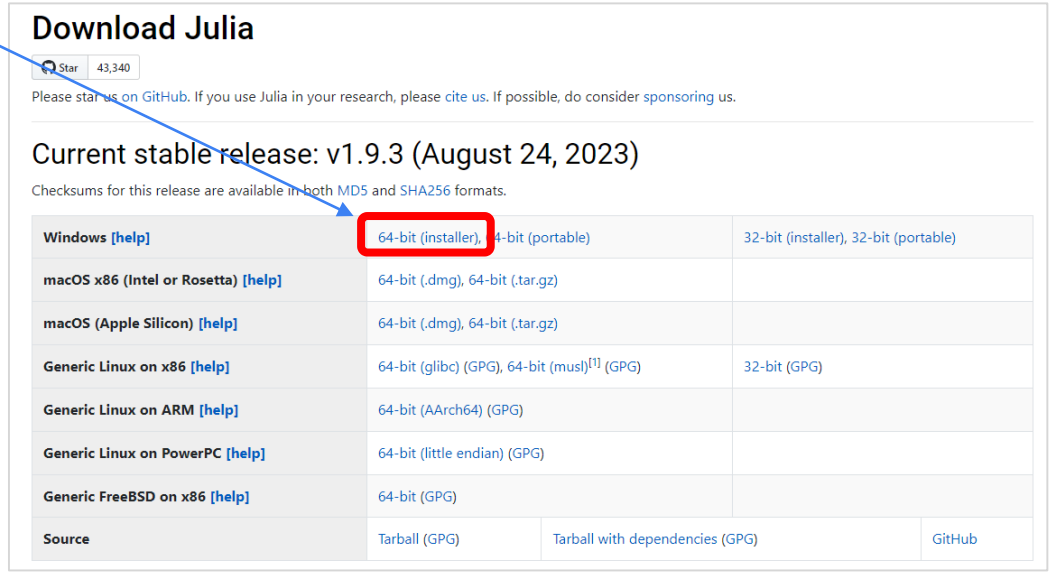
Installing Julia

How to install Julia and Visual Studio Code

1 Go to The Julia Programming Language web page (julialang.org)

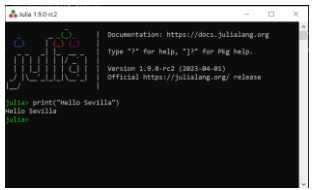
2 Click on the "Download" button and then on your preferred version. If you are in windows on a decent computer (64bits processor) select the version highlighted below





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



"The REPL", an instance of Julia running

REPL and package manager

The Julia REPL

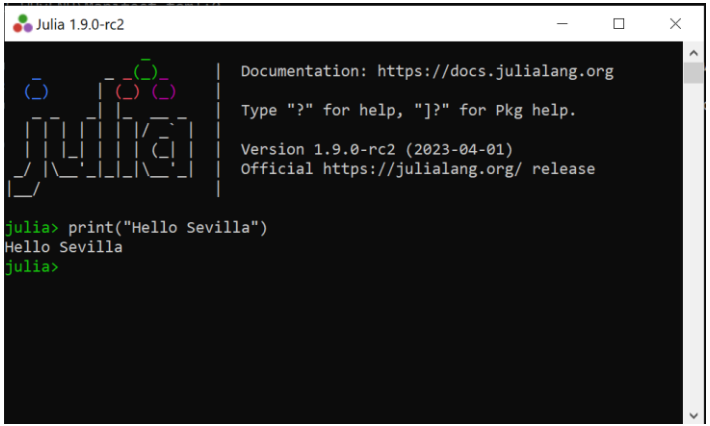
"REPL" stands for Read Evaluate Print Loop and it's where you are just after starting Julia.

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: ←



]

←

REPL

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

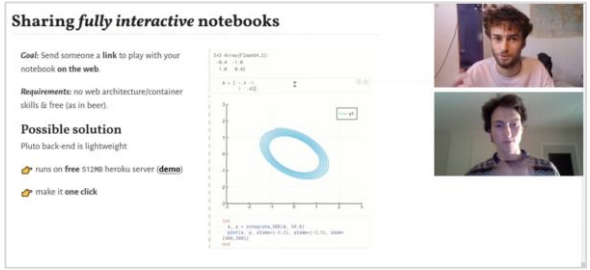
Julia + Pluto

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.

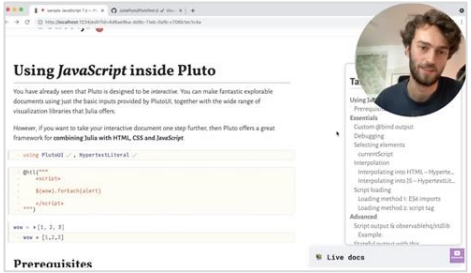
Formally, Pluto is a notebook, like **Jupyter** (which means **Ju**lia, **Py**thon and **R**). You can also use Jupyter to write Julia notebooks (and many people do) but the main advantage of Pluto is that it is ***reactive***.

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



Watch more Fons, another peculiar character in the Julia world



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).

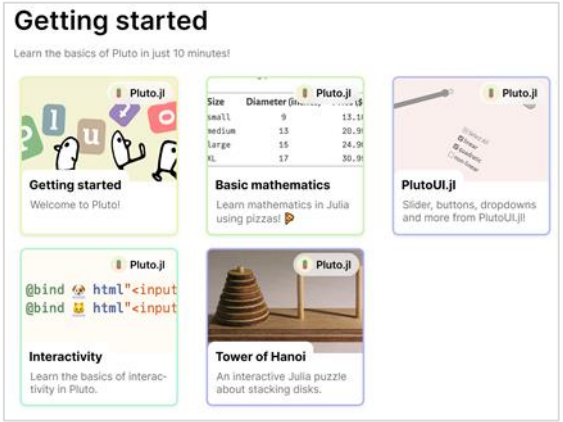
If you put the full path name or the url of an existing notebook in the brackets it will open it.

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;

and follow the instructions

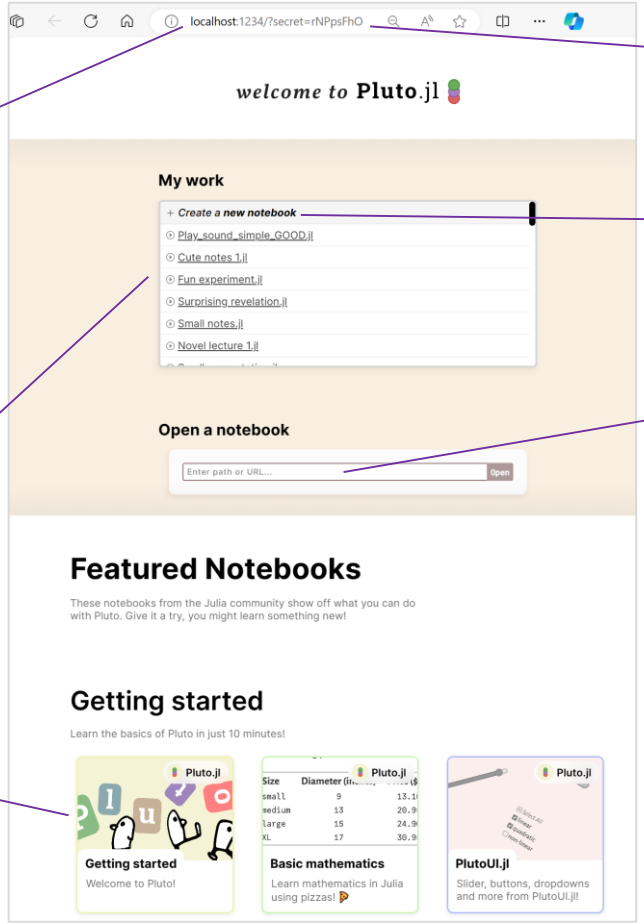
Exploring Pluto

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 wat 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

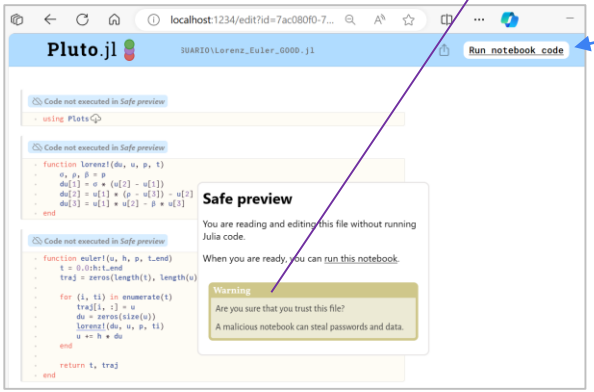
and

https://github.com/MyRepo/MyProject/blob/main/Lorenz_Euler.jl

are well formatted names of possible Pluto (Julia) files

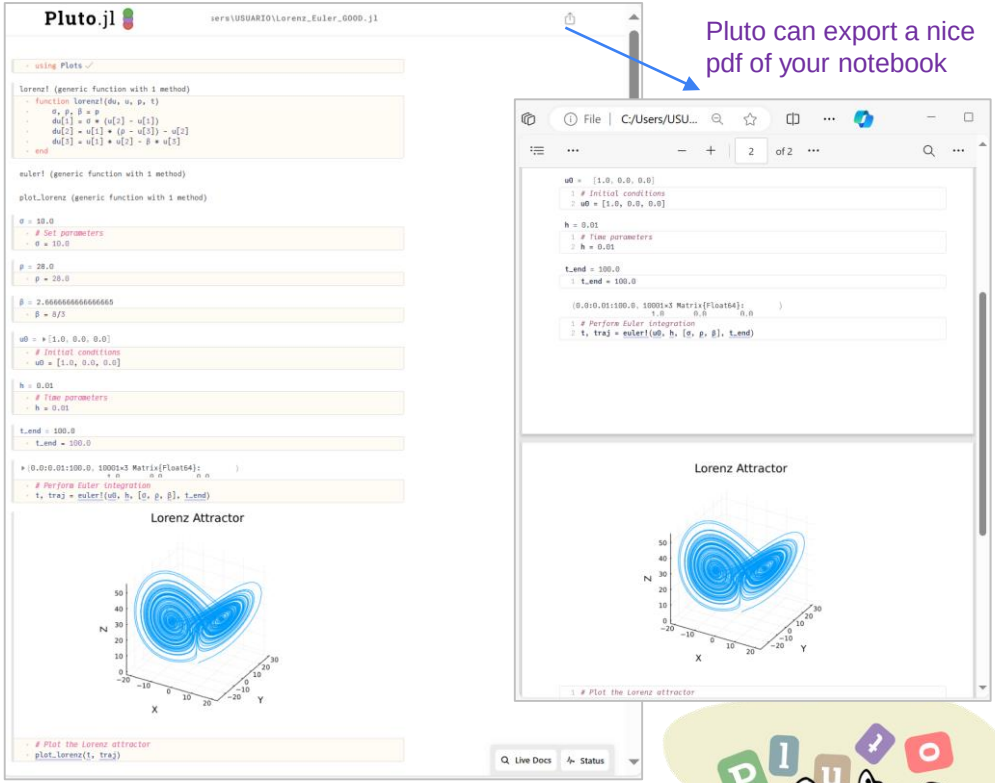
Exploring Pluto

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



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

Pluto can export a nice pdf of your notebook



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”)

Exploring Pluto

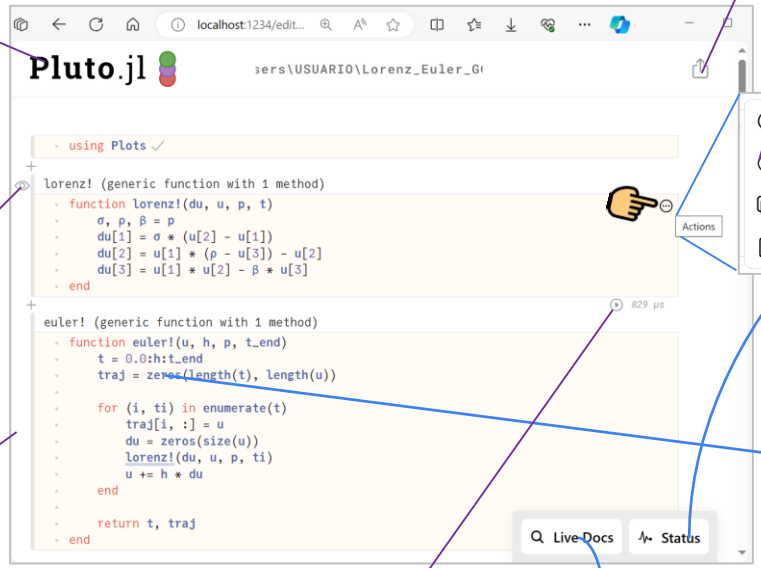
Exit the notebook (it will be saved with a funny name if you don't save it) and go back to the start page

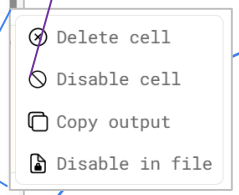
Hide/show the code in the cell. The results of the cell, which could be none, will always be shown above the cell

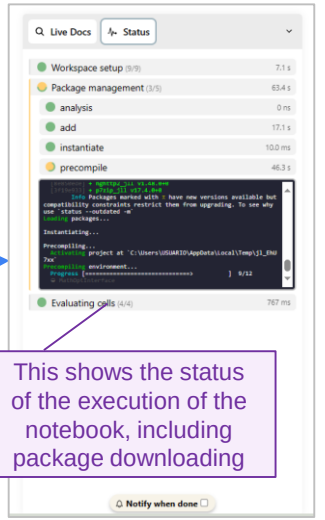
Click on the margin of the cell to move it up or down in the notebook. **Pluto does not care about the order of the cells**

Export the notebook, including as a pdf file

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

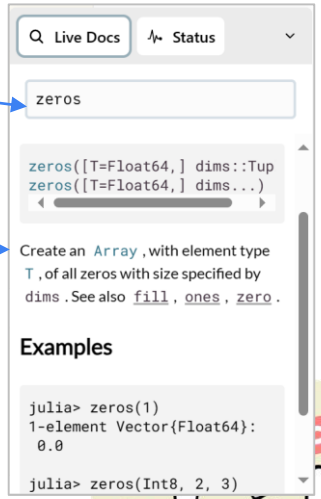






This shows the status of the execution of the notebook, including package downloading

In the REPL this would be `?` and then `zeros`



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

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

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

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 Live Docs 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)

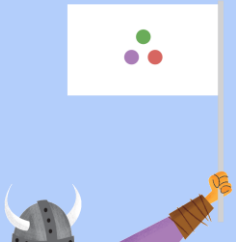
Convert a speed v from meters per second (m/s) to knots (kt)

# 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 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

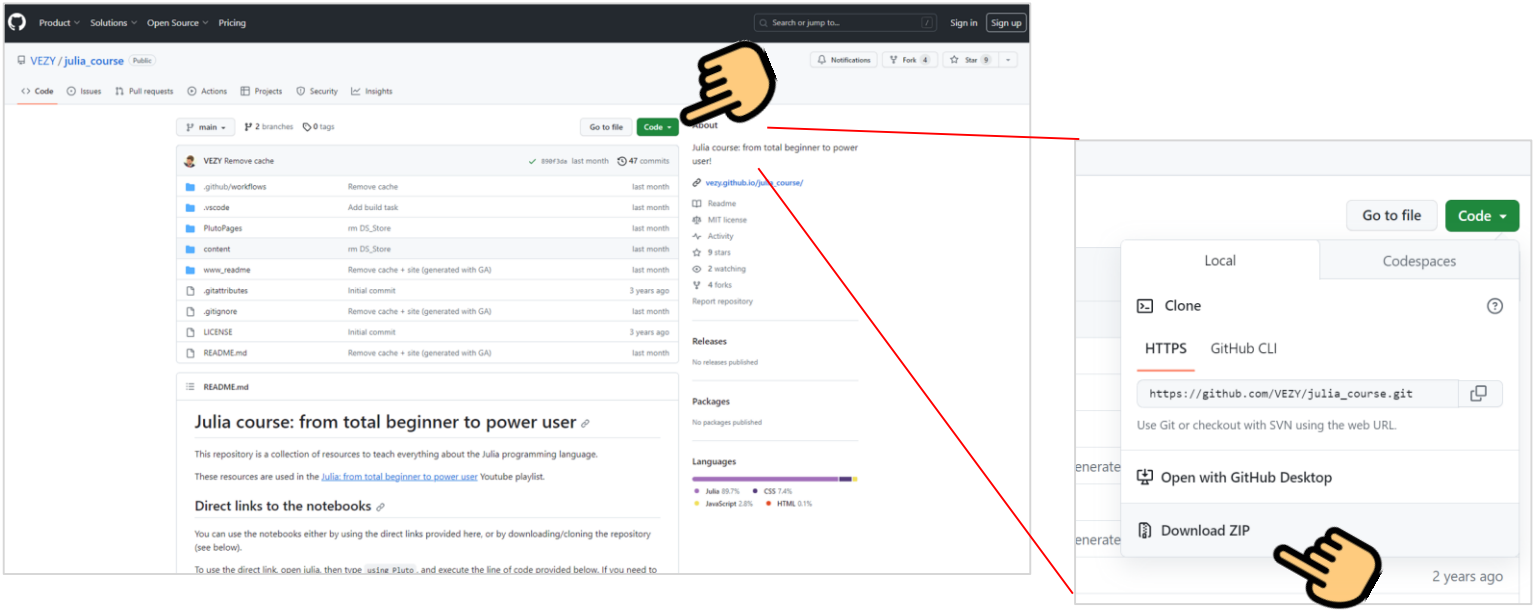


Course notebooks (Rémy Vezy)

[Github repository for this course]

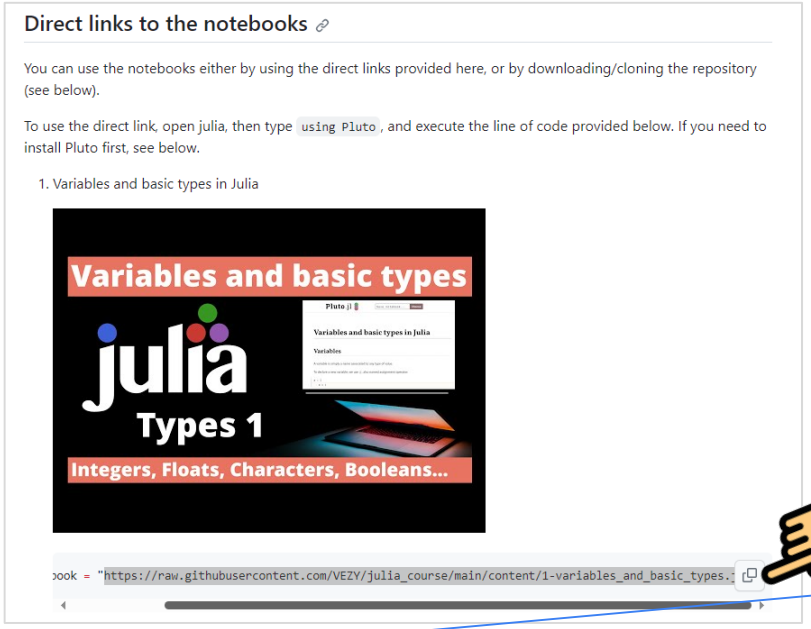
A good place to start learning the basic Julia syntax is a course developed by Rémy Vezy, who is a researcher in plant (vegetal…) structure.

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

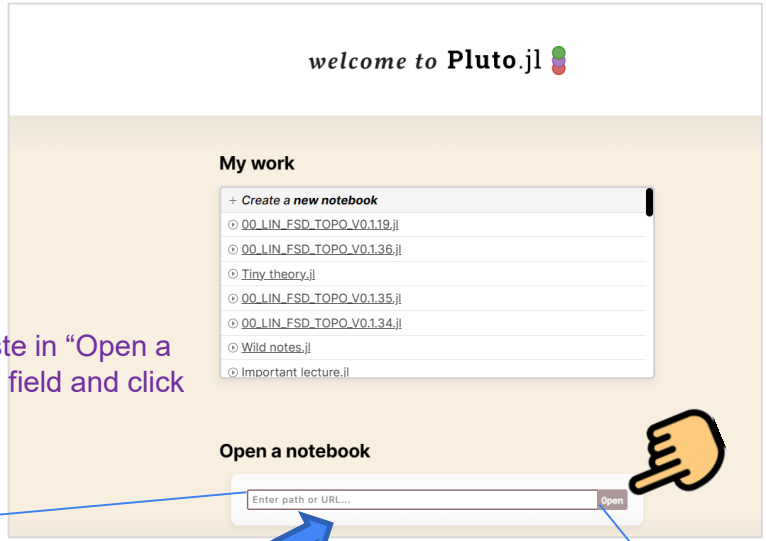


Opening the course notes in Pluto

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.



Copy, paste in “Open a notebook” field and click “Open”



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

2- Arrays

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

4- Dictionaries

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

Basic Julia syntax (2/2) (Rémy Vezy)

6- String operators

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

8- Conditional statements (if...else...)

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

10- Functions

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., a11^2 a12^2 a13^2 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 (a11*c11 a12*c12 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

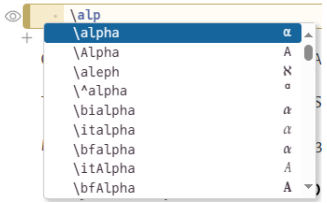
```
#=
I am also
a comment
=#
```
alternative multiline comment

! 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)

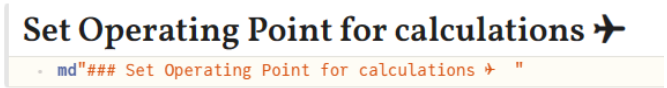
Special symbols, Greek letters. Can be written by typing, for example *\alph + tab* and choosing the option you want



☺ 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 ✈ “`



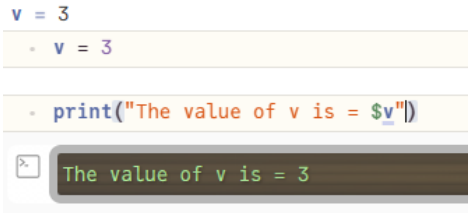
`==` Is equal? Note that = is used for assignment; `3 == 3` -> true `a = 3` ; a -> 3

`!=` 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**)



|> "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**

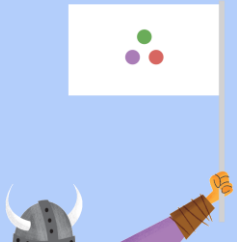
f(x) = x^2 ; g(x) = x/2 ; 4 |> f |> g = 8 ; |>(4,f) = 16

… "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)**

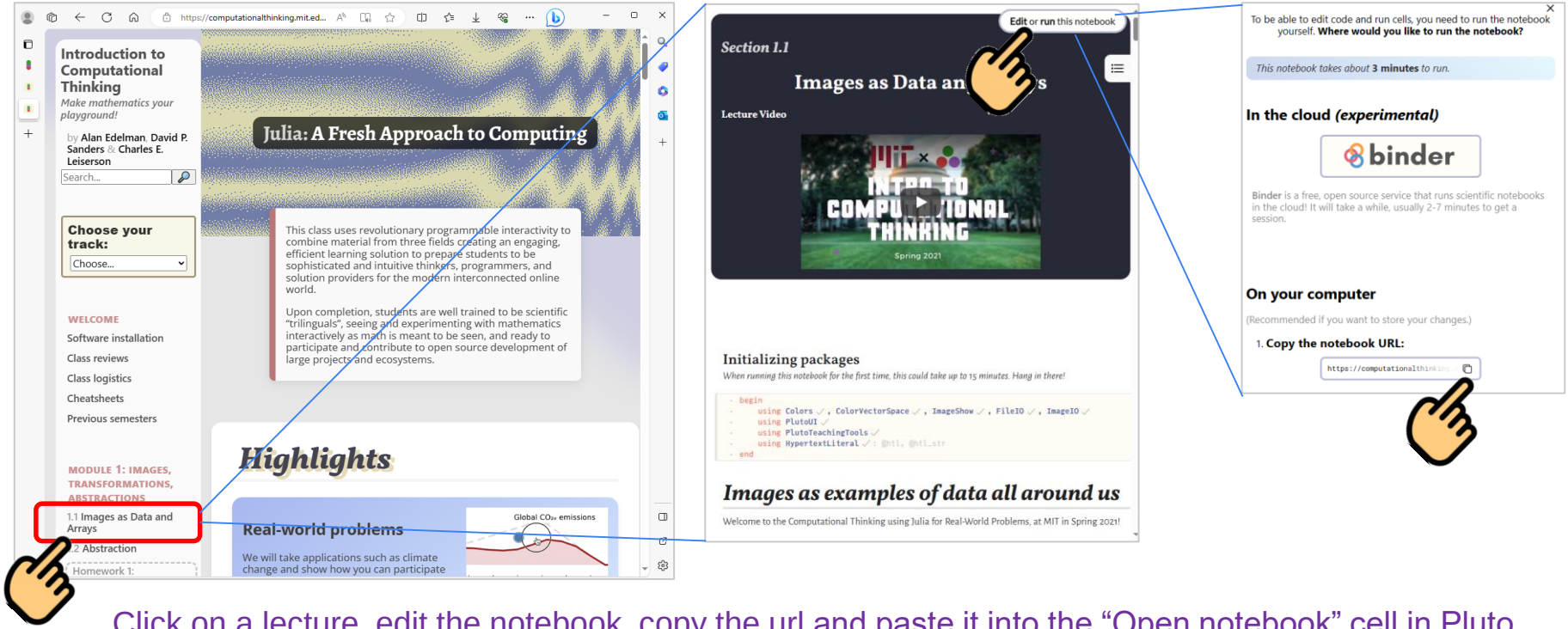
Warming up



Intermediate topics (MIT Computational Thinking)

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

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



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

1.4- Transformations with images

https://computationalthinking.mit.edu/Fall23/generated_assets/transforming_images_b0baefaa.jl

2.9 Optimization

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

3.6 Snowball Earth and Hysteresis (excellent video)

https://computationalthinking.mit.edu/Fall23/generated_assets/climate2_snowball_earth_bffbd696.jl

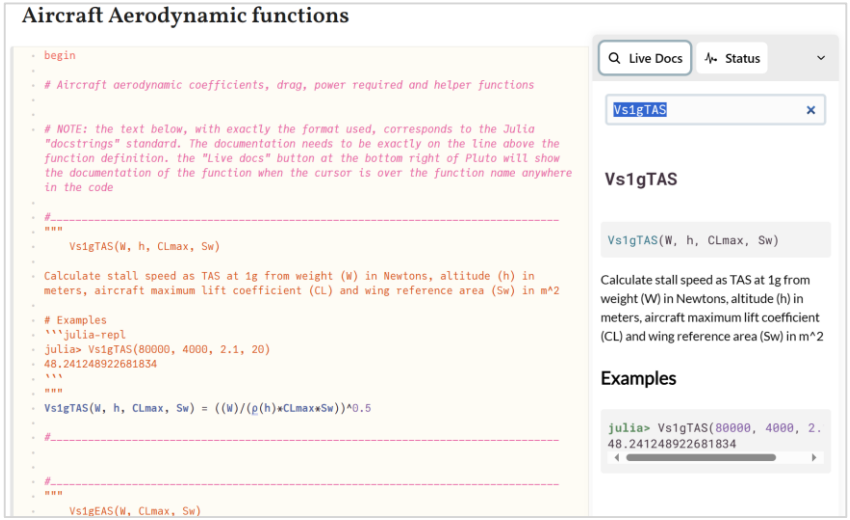
Interactive notebooks for education

https://github.com/flt-acdesign/Low_speed_AC_performance/blob/main/21_04_08_Low_speed_perfo_v_0.0.1.jl

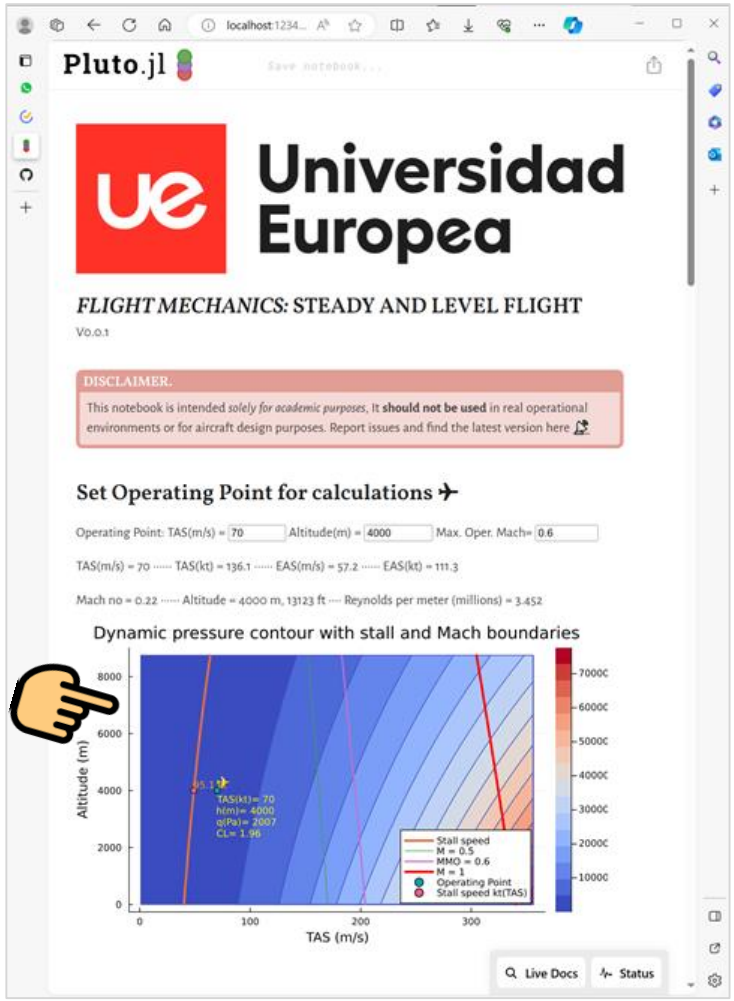
This Pluto notebook is a simple example of an interactive resource to teach aircraft performance.

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.



Check the code for this plot for inspiration



Enter chatGPT (or Bing chat or…)

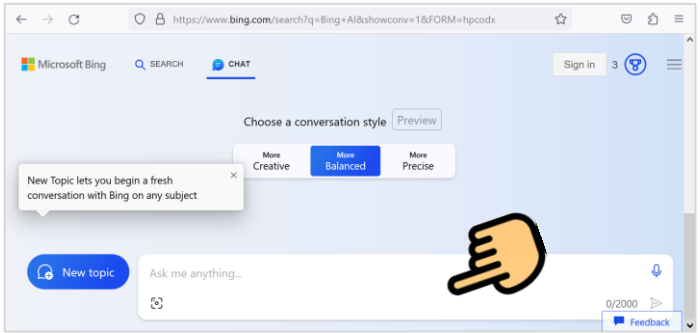
The world will never be the same since the advent of ChatGPT (and many other Large Language Models since).

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

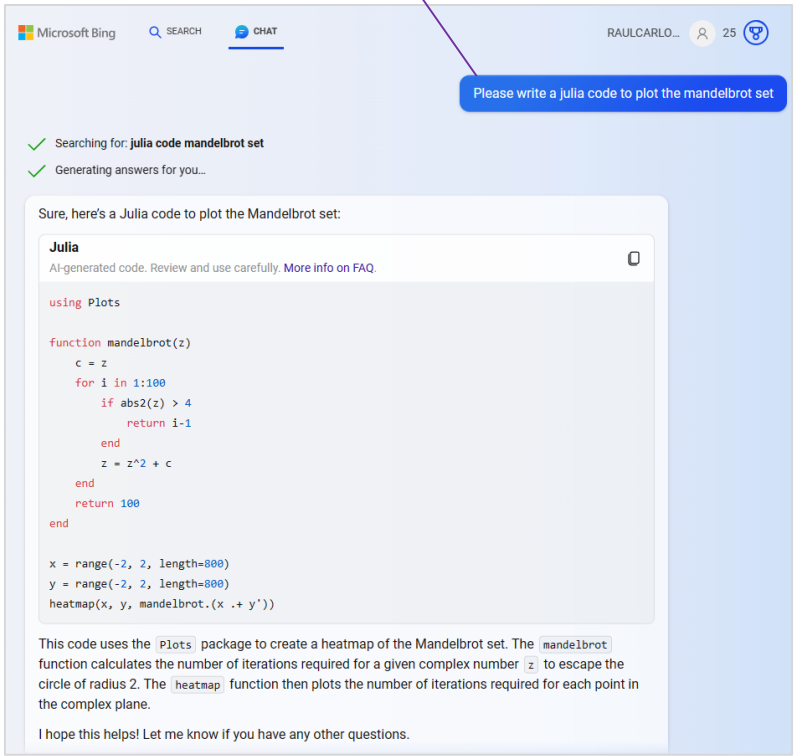
However, be prepared to edit the code manually or keep asking for refinements (“Regenerate”) until it works**. Note that you may get different answers every time as the system keeps on “learning”!**

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



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

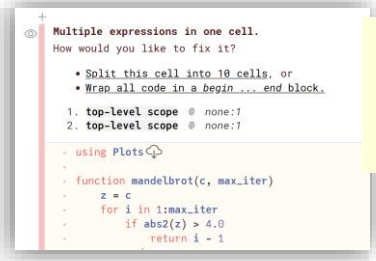
    x = LinRange(x_min, x_max, width)
    y = LinRange(y_min, y_max, height)

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

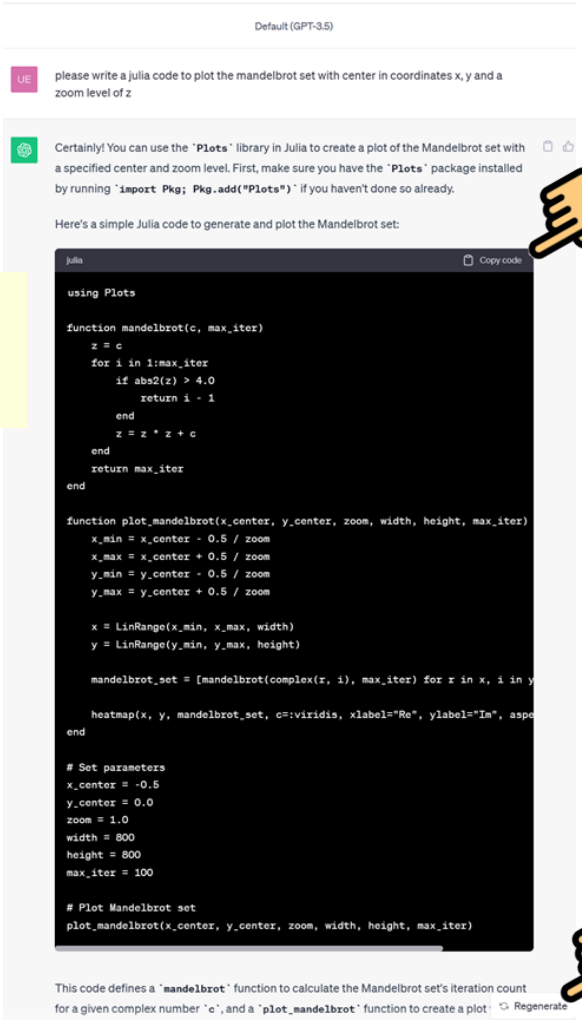
    heatmap(x, y, mandelbrot_set, c=:viridis, xlabel="Re", ylabel="Im", aspect_ratio=:equal)
end

# 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…”



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*

```
using Plots ✓, PlutoUI ✓   # Load required packages (For plotting and interactivity)
```

1.9 s

```
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" (δ) 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 **`δ`**

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.

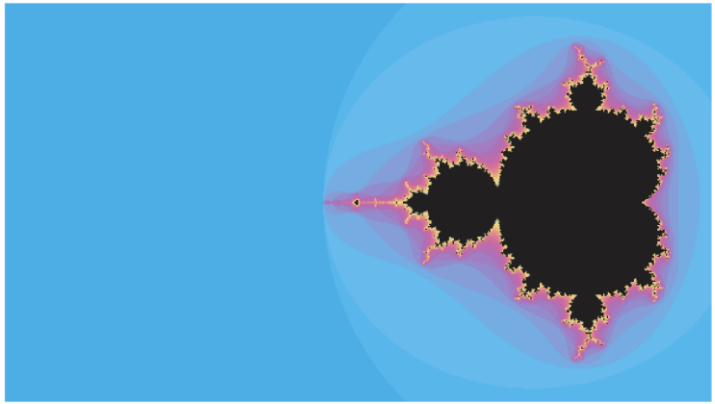
A refined version of Mandelbrot

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

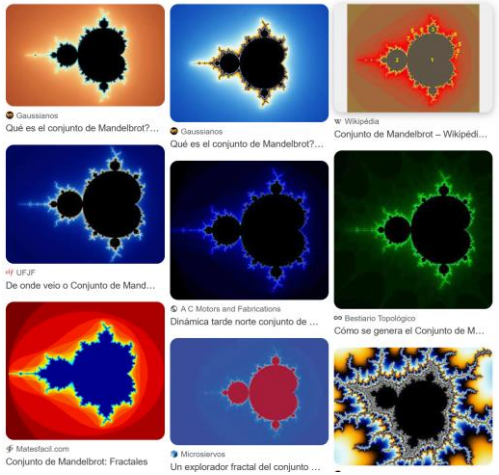
The `setprecision(BigFloat, 500)` is commented to improve performance for normal levels of zoon. When you zoom a lot you may need higher floating point precision and the previous statement sets the representation of floating point numbers at 500 bits (normally in Julia they have 64, as in `Float64`).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



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.




```
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
```

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)

Coordinates from this video https://www.youtube.com/watch?v=aSg2Db3jF_4

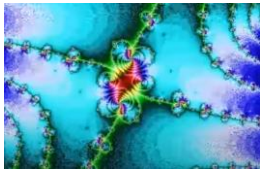
```
begin
    # Default coordinates in the complex plane of the center of the image
    xcvideo =
    BigFloat(-1.749957683706093503602214506070699707271105797262520779302428378202860080829728048872186727844317008311005445076556595313797475419999999995)

    ycvideo =
    BigFloat(0.000000000000000002787937065633794021782947537909443649270850545001630813790439306501893868497652021694774705522013257723324547269999999995)

    md" Coordinates from this video  https://www.youtube.com/watch?v=aSg2Db3jF_4"
end
```

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



Lorenz Attractor – Euler method

Visualizing the Lorenz Attractor in Julia | by Daryl Tng | Medium

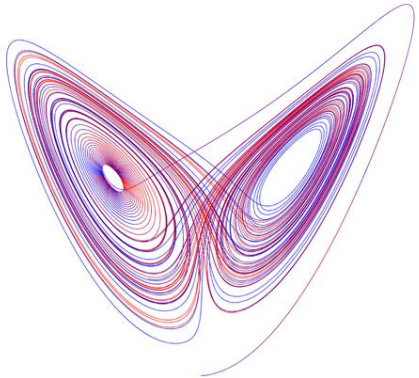
The Lorenz system of equations is very famous in mathematical and popular culture. It opened the field of Chaos which loosely means that infinitesimal variations of the initial conditions will give rise to totally different behaviours of a “chaotic” system (of which there are many, starting by the atmosphere,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

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

The 3 governing equations of the Lorenz system



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] = u[1] * (ρ - 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, ti) in enumerate(t)
        traj[i, :] = u
        du = zeros(size(u))
        lorenz!(du, u, p, ti)
        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 system – Wikipedia with a Julia code!

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

```
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
```

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 du/dt.

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

```
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
```

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.

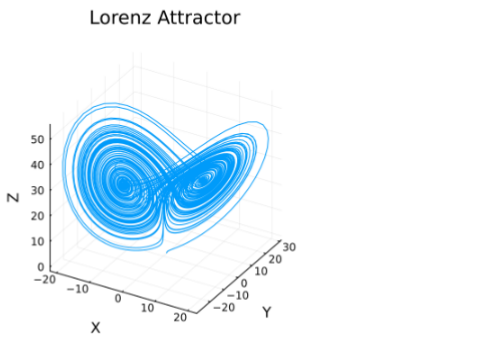
```
plot_lorenz (generic function with 1 method)
```

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

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

Lorenz Attractor – Euler method

```
σ = 10.0
```
```
# Set parameters
σ = 10.0
```
```
ρ = 28.0
```
```
ρ = 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}:
 1.0   0.0   0.0
```
```
# Perform Euler integration
t, traj = euler!(u0, h, [σ, ρ, β], t_end)
```




```
# 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*

```
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).

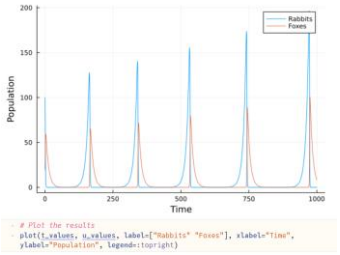
The **Lotka–Volterra equations**, also known as the **Lotka–Volterra predator–prey model**, are a pair of first-order nonlinear differential equations, frequently used to describe the dynamics of biological systems in which two species interact, one as a predator and the other as prey. The populations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

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

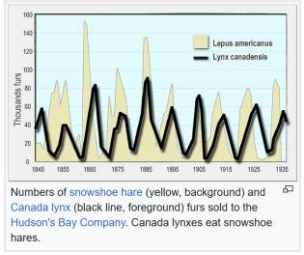
Lotka–Volterra equations - Wikipedia

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, α and β, 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, γ, δ, respectively describe the predator's per capita death rate, and the effect of the presence of prey on the predator's growth rate.

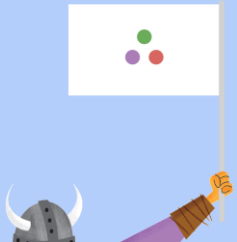



```
# Plot the results
plot(t_values, u_values, label=["Rabbits" "Foxes"], xlabel="Time",
ylabel="Population", legend=:topright)
```



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.

Getting serious with Julia



Julia Types

Types · The Julia Language

The “type” of a variable refers to the information that it can contain and it is required by the computer to store the value in memory.

Different types require different numbers of “bits” of memory and in the case of numbers can represent different ranges of numbers with different level of precision

In Julia it’s not obligatory to declare the types of variables and constants (the compiler will detect them), but it is good practice to help Julia store the data in an orderly way and as close to the CPU as possible

```
a = 3; typeof(a)
```
-> `Int64`

```
b = 3.0 ; typeof(b)
```
-> `Float64` This is the default type for numbers when you write a “decimal point” anywhere (0.4, .4, 3., 3.0). It has a “precision” of `eps(Float64)` -> 2.220446049250313e-16

Languages that detect the types are called “dynamically typed” (like Julia, Python, Basic…) and those which need explicit declaration of the types of variables are called “statically typed” (like C, Fortran, Typescript…).

Julia is considered to be “dynamically typed” but is has the performance of a “statically typed” language, specially if you help her…

To declare a variable, function input or function output value of a certain type use `::` as in ;

```
x::Int = 10 ;
function sinc(x)::Float64
```

The Julia type system

"Any" is the most general and memory-expensive type and is the default value when you declare an empty array like

```
Vector{Any} (alias for Array{Any, 1})
  B = [] ; typeof(B)
```

In Julia only "concrete" types can hold values.

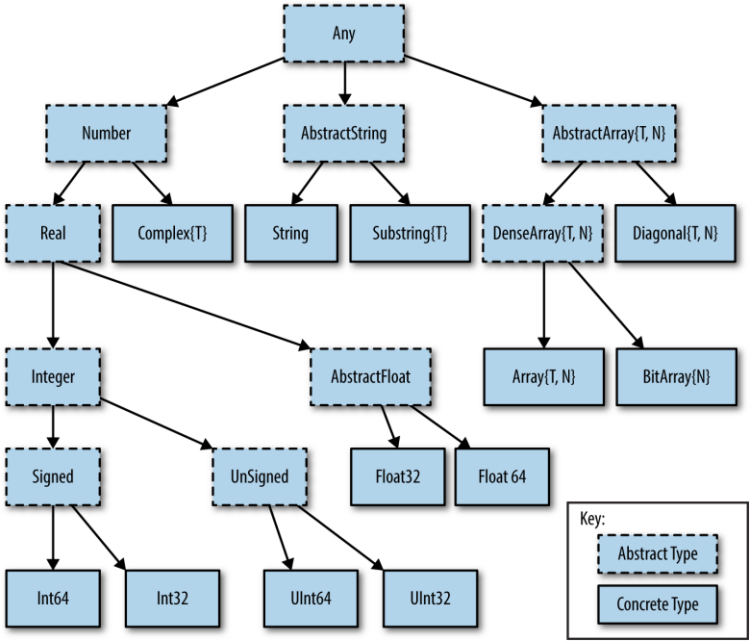
"Abstract" types are "supertypes" in the type system to group related concrete types.

A function defined with inputs of the abstract type "Number" will work with any of its subtypes.

The hierarchy can be explored with `<:` as follows;

```
julia> Integer <: Number
true

julia> Integer <: AbstractFloat
false
```

If you want to specialize it for a particular concrete type you can do it creating a new method (more later)



Multiple dispatch: Functions and Methods

[01x07] Julia's Secret Sauce? Methods, Multiple Dispatch and More; Tutorial 7/13 Julia for Beginners - YouTube

A key characteristic of Julia is “multiple dispatch” which means that a function with the same name can have various implementations with different input types. Each implementation is called a “method” of the function.

Julia will “dispatch” or call the corresponding method to the input types and if no method exist will give a “***no method matching***” error.

For example, the Julia function * can work in very different ways in these two cases (note that *(x,y) = x * y);

```
julia> *(3,4)
12
```

Usual number mutiplication operating on integers

```
julia> *(”Hello Sevilla")
”Hello Sevilla”
```

String concatenation (The Julia manual explais why ”*” instead of the usual ”+”)

To know how many methods a function ”f” has implemented type `methods(f)`

You can write a new method for the `*` function by just stating the types of the inputs

```
function *(x::Fruit, y::Fruit)
```

where Fruit is a custom type, or a structure, that you have created.

Type safety

Although Julia can infer the types by “their looks” (3 is Int and 3.0 is Float), performance and “code safety” (avoiding errors) is improved if the types are explicitly specified

Making sure that Julia can know the types of data flowing through the code is called “type safety” and is a key performance enhancer as it allows proper and efficient memory allocation.

A classical example where this is not observed is;

```
function myfun(a; tol = 2)
          x = (a > 42 ? a : 42.)
          return x + tol
         end
```

the type of the return variable will change at run time depending on the value of one input

The macro `@code_warntype` will analyze the code and identify the types flowing, in this case there is a “Union”, which is inefficient

```
julia> @code_warntype myfun(2)
MethodInstance for myfun(::Int64)
  from myfun(a; tol) @ Main REPL[8]:1
Arguments
  #self#::Core.Const(myfun)
  a::Int64
Body::Union{Float64, Int64}
1 ─ %1 = Main.:(var"#myfun#3")(2, #self#, a)::Union{Float64, Int64}
└──      return %1
```

LLVM and intermediate representations of code

Julia with the LLVM compiler is extremely efficient because it interprets the code in various phases, optimizing the code representation at each level (this is called code lowering)

The last phase is machine code but the intermediate representations (IR) can be used (by advanced coders) to optimize the code and understand bottlenecks. The IR can be inspected as in the example below:

```
my_fun (generic function with 1 method)
```

```
my_fun(x) = x + x^2
```

Function definition, untyped

```
CodeInfo(
1 — %1 = Main.var"workspace#22".:^
│   %2 = Core.apply_type(Base.Val, 2)
│   %3 = (%2)()
│   %4 = Base.literal_pow(%1, x, %3)
│   %5 = x + %4
└──      return %5
)
```

```
@code_lowered my_fun(3)
```

First level of code lowering by LLVM. To carry on type : `@code_llvm my_fun(3)` and `@code_native my_fun(3)`

```
CodeInfo(
1 — %1 = Base.mul_int(x, x)::Int64
│   %2 = Base.add_int(x, %1)::Int64
└──      return %2
) ⇒ Int64
```

```
@code_typed my_fun(3)
```

Lowered code showing the type interpretation for the given input values

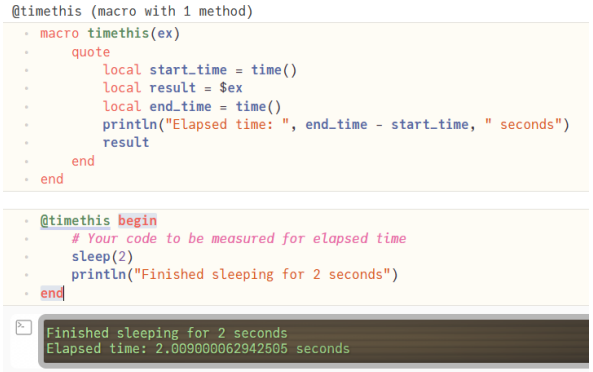
Macros and metaprogramming

Metaprogramming · The Julia Language

Macros are functions that take code as input, transform it and output new code which is immediately evaluated as part of the program flow.

“Metaprogramming” refers to this technique where a running program processes itself as if it was data.

In Julia, the name of macros is called with a “@” prefix just to differentiate them from normal functions



A simple example is a macro which introduces additional code around a function or block of code to measure the time elapsed to evaluate a function.

This is done by the base Julia macro @time, but its not the best way to measure performance because it can include compilation time etc..

Metaprogramming is an advanced topic but any user can benefit from already available macros in base Julia and many packages. Very commonly used macros in Julia are `@time`, `@btime` (from BenchmarkTools), `@.` , `@inline` and others

Inlined functions (the reason why Julia can be faster than C)

```
sq(x) = x^2
```

```
for i in 1:1000_1000
    sq(i)
end
```

In this trivial (and unrealistic) example, the function sq would be called a million times. It is unrealistic because the LLVM compiler is smart enough to embed the code of the

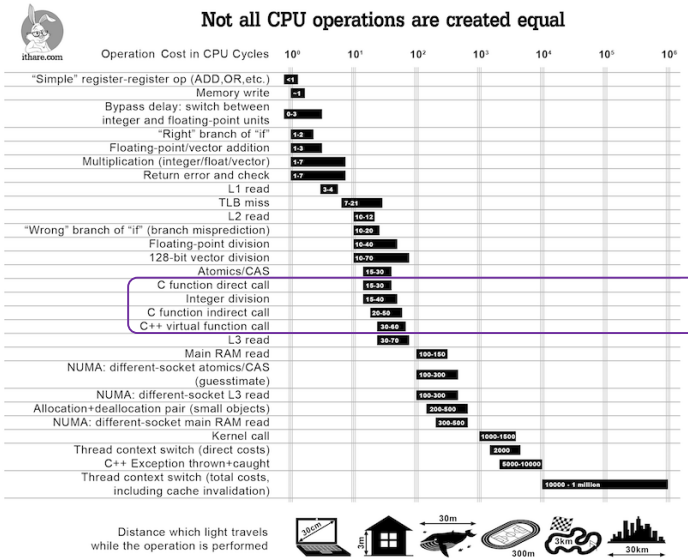
function in the loop when the code is lowered, saving the function call, which is expensive. This is called “inlining” and for larger functions can be “suggested to the compiler with the macro @inline as in this example:

Defining the function as `@inline sq(x) = x^2`

would effectively result in the loop becoming :

```
for i in 1:1000_1000
    x^2
end
```

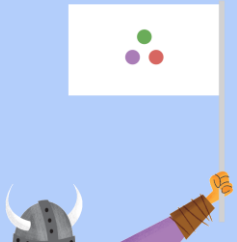
which avoids the function call (although in this simple case the inlining will happen anyway)



Calling a function takes about 100 CPU cycles.

Inlining is the reason why Julia can appear to be faster than (old) C in some benchmarks, due to the avoidance of function calls

The Julia ecosystem

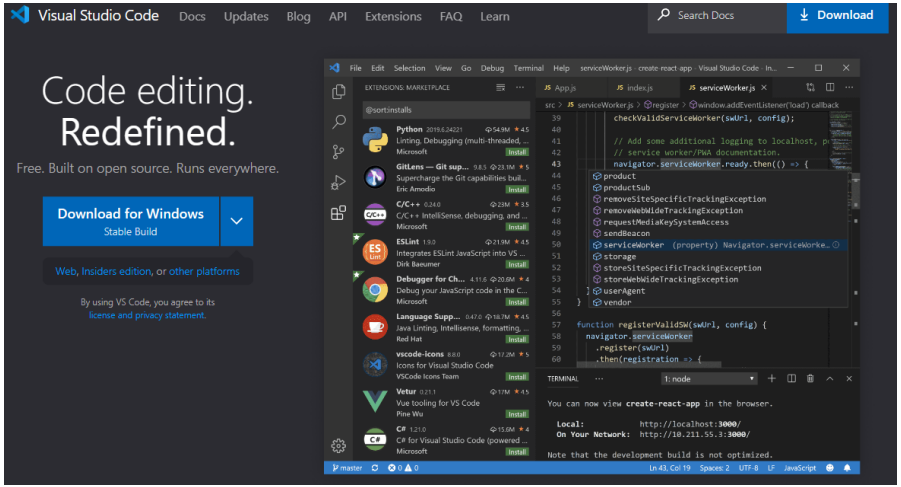


Visual Studio Code

[10x06] How to use Julia in VS Code - YouTube

Although Pluto is even more powerful than it seems, when the code becomes too complex or takes long to execute it shows its limitations

The “serious” way to code in Julia is using Visual Studio code (or “VScode). In fact this is the case now for most open source languages)



The Julia extension for Vscode is continuously being developed and offers many advanced features

What's new in the Julia extension for VS Code | Sebastian Pfitzner | JuliaCon 2023 - YouTube

In this workshop we will stick to Pluto even for the advanced topics

Julia Packages

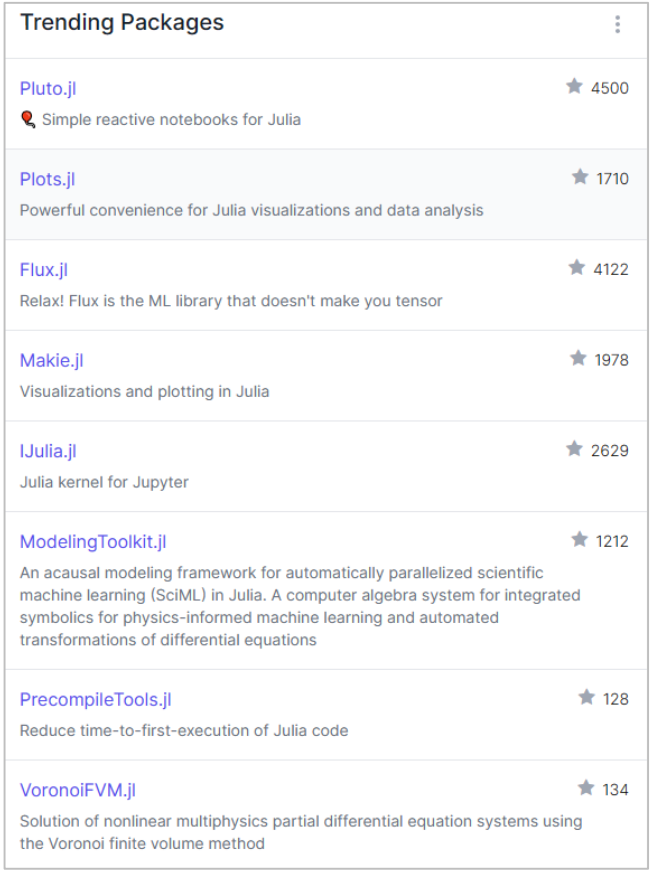
There are thousands of Julia packages freely available, you can find a curated list here Julia Packages (as you can see, in Oct 2023 Pluto is the most popular of all)

Not all the packages are equally well documented or useful. Eventually the eco-system will converge to a reduced number of “good” packages per domain.

The source code of packages and, in most cases, the documentation (if it exists) is located in a Github repository.

The complete Julia language lives in a Github repository (and can be explored and even improved by you)

You can also create packages for the Julia eco-system, check 5. Creating Packages · Pkg.jl (julialang.org)



Creating local environments. Project.toml and Manifest.toml

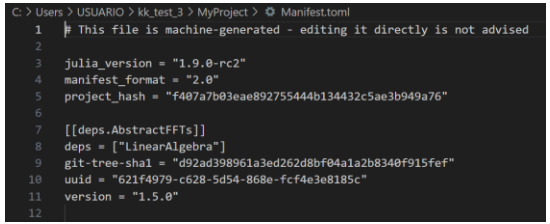
How to setup Project Environments in Julia | by René | Towards Data Science

So far we have just added packages in the package manager without taking care of where these are installed. This is a “global” installation and makes the packages available to all Julia programs as long as all the paths are set correctly in the installation.

But as packages evolve, their new versions can be incompatible with dependent packages installed by other packages, which can lead to compatibility problems if all the packages “live” in the same “environment”

To create a local environment for each project you start do the following:

1- Using a console (terminal) go to the folder where you want to create for your project called **`MyProject`**
2- Start Julia by typing **`julia`** and enter the package manager (type **`]`**)
3- Type **`generate MyProject`** This will create a directory called **`MyProject`** that contains a **`Project.toml`** file and a folder for the source code. **`Project.toml`** contains general information about the project creation.
4- type activate **`MyProject`**
5- Add the packages you need, for example type **`add Pluto Sound`** At this point the **`Manifest.toml`** file will appear, which contains all the information relative to the packages installed in this environment. When you’re done go back to REPL



You don’t need to edit these files but they contain all the information to guarantee the consistency of the libraries (packages) used in your project. Even if the package versions keep evolving, you can always use the correct ones to which the **`Manifest.toml`** file points

10. Project.toml and Manifest.toml · Pkg.jl (julialang.org)

Image convolution using Images

https://github.com/JuliaImages/Images.jl

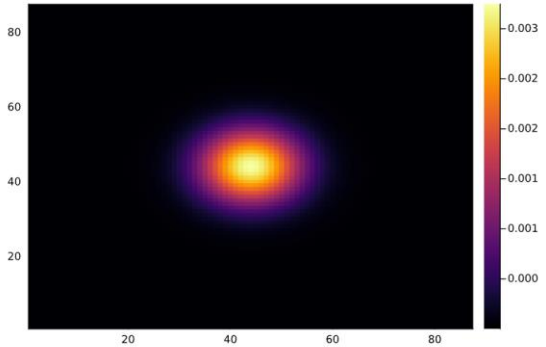
Open this code in Pluto https://github.com/flt-acdesign/Mathematical_tourism_with_Julia/blob/main/gaussian_convolution_GOOD2_withlayout.jl

- Convolution 3blueone..

```
using Images ✓, PlutoUI ✓
```

```
using Plots ✓
```

```
gaussian_kernel (generic function with 1 method)
function gaussian_kernel(σ)
    k = 2 * ceil(Int, 3.0 * σ) + 1
    kernel = [exp(-(i^2 + j^2) / (2 * σ^2)) for i in -k:1:k, j in -k:1:k]
    kernel /= sum(kernel)
    return kernel
end
```




```
heatmap(gaussian_kernel(σ))
```

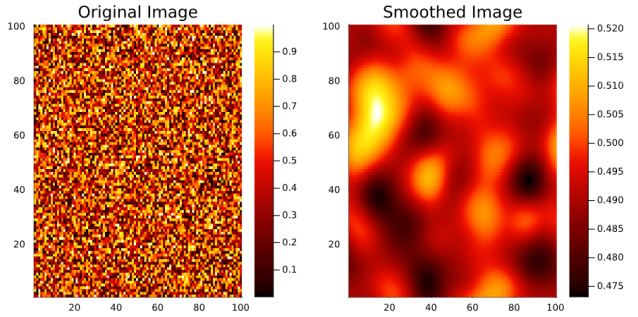
```
gaussian_convolution (generic function with 1 method)
function gaussian_convolution(image, σ)
    kernel = gaussian_kernel(σ)
    return imfilter(image, kernel, "circular")
end
```

```
# Example usage
img = rand(100, 100);  # Replace this with your image data
```

```
@bind σ Slider(1:10)
```

72.7 ms

```
smoothed_img = gaussian_convolution(img, σ);
```




```
# Display the original and smoothed images using Plots
plot(
    heatmap(img, c=:fire, title="Original Image"),
    heatmap(smoothed_img, c=:fire, title="Smoothed Image"),
    layout=(1,2),
    size=(800,400)
)
```

JuMP - Modeling language for Optimization

Open this code in Pluto  *https://github.com/flt-acdesign/Mathematical_tourism_with_Julia/blob/main/JuMP_example_GOOD*

```
using JuMP ✓
```

```
using Cbc ✓   # or GLPK if you prefer GLPK
```

```
model = A JuMP Model
        Feasibility problem with:
        Variables: 0
        Model mode: AUTOMATIC
        CachingOptimizer state: EMPTY_OPTIMIZER
        Solver name: COIN Branch-and-Cut (Cbc)
```

```
# Create a JuMP model
model = Model(Cbc.Optimizer)  # or Model(GLPK.Optimizer) if you prefer GLPK
```

$$x$$

```
# Define variables
@variable(model, x >= 0)
```

$$y$$

```
@variable(model, y >= 0)
```

$$3x + 2y$$

```
# Define the objective function
@objective(model, Max, 3x + 2y)
```

$$2x + y \leq 20$$

```
# Define constraints
@constraint(model, constraint1, 2x + y <= 20)
```

$$4x - 5y \geq -10$$

```
@constraint(model, constraint2, 4x - 5y >= -10)
```

```
# Solve the optimization problem
optimize!(model)
```

UE please write an example of mathematical optimization using the JuMP package in Julia

Sure, I'll provide a simple example of mathematical optimization using the JuMP package in Julia. In this example, we'll solve a linear programming problem:

$$\begin{aligned} &\text{Maximize } 3x + 2y \\ &\text{Subject to} \\ &2x + y \leq 20 \\ &4x - 5y \geq -10 \\ &x \geq 0, y \geq 0 \end{aligned}$$

Make sure you have the JuMP and a solver (like Cbc.jl or GLPK.jl) installed. You can install them using the following commands:

JuMP

Pluto code for the plot *https://github.com/flt-acdesign/Mathematical_tourism_with_Julia/blob/main/Ilustrate_JuMP_solution_GOOD*

```
# Solve the optimization problem
optimize!(model)
```

```
status = OPTIMAL::TerminationStatusCode = 1
# Check the optimization status
status = termination_status(model)
```

```
println("Optimization Status: $status")
```

```
Optimization Status: OPTIMAL
```

```
# Display the results
println("Optimal value of x: ", value(x))
```

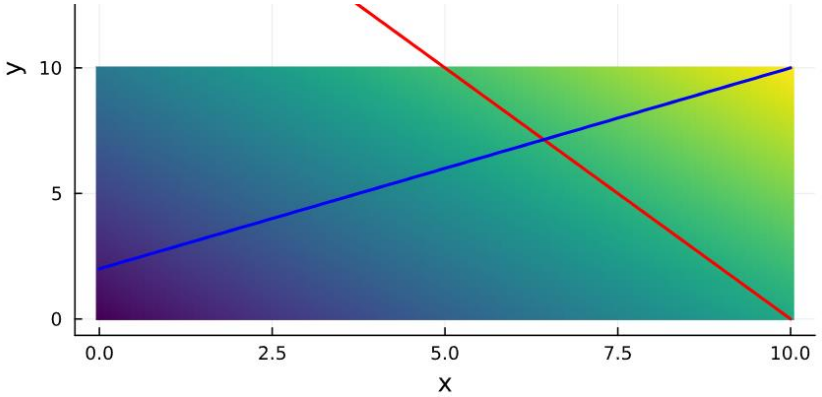
```
Optimal value of x: 6.428571428571429
```

```
println("Optimal value of y: ", value(y))
```

```
Optimal value of y: 7.142857142857143
```

```
println("Optimal objective value: ", objective_value(model))
```

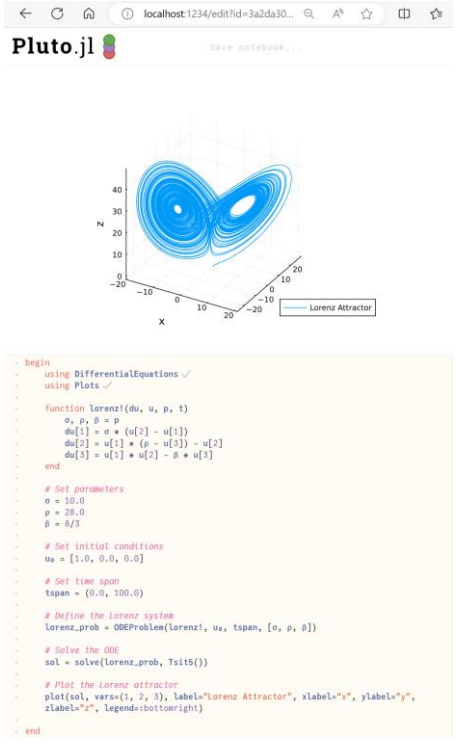
```
Optimal objective value: 33.57142857142857
```



Differential Equations

Lotka-Volterra DifferentialEquations

Lorenz attractor – DifferentialEquations package



Default (GPT-3.5)

UE Please write a julia code to plot the lorenz attractor

Certainly! The Lorenz attractor is a set of chaotic solutions to a system of three ordinary differential equations. Here's a simple Julia code using the `DifferentialEquations` and `Plots` packages to simulate and plot the Lorenz attractor:

First, make sure you have the required packages installed. You can install them by running:

julia Copy code

```
using Pkg
Pkg.add("DifferentialEquations")
Pkg.add("Plots")
```

Now, you can use the following code:

julia Copy code

```
using DifferentialEquations
using Plots

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

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

# Set initial conditions
u₀ = [1.0, 0.0, 0.0]

# Set time span
tspan = (0.0, 100.0)

# Define the Lorenz system
lorenz_prob = ODEProblem(lorenz!, u₀, tspan, [σ, ρ, β])

# Solve the ODE
sol = solve(lorenz_prob, Tsit5())

# Plot the Lorenz attractor
plot(sol, vars=(1, 2, 3), label="Lorenz Attractor", xlabel="x", ylabel="y",
```

This code defines the Lorenz system using the `lorenz!` function, sets up the parameters, initial conditions, and time span, and then solves the system using the `solve` function from the `DifferentialEquations` package. Finally, it uses the `Plots` package to visu

Regenerate

Lorenz attractor in 3D.

flux - Machine Learning

Welcome · Flux (fluxml.ai)

Machine Learning is a part of mathematics/computer science which focuses on the development of statistical algorithms which can exploit large amounts, generally high dimensional, data to create predictive models.

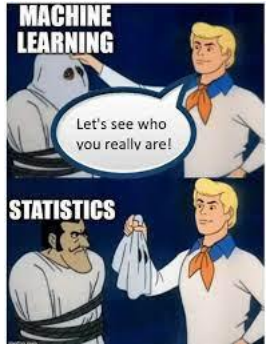
The mathematical models (simple regressions, Neural Networks, decision trees, Bayes, Support Vector Machines…) tend to require an iterative process to adjust the model parameters to the available data. This process is usually called “learning”.

Julia has various packages for Machine Learning but the most mature is Flux. It is not as powerful as Tensorflow or Pytorch but the code is completely open and you can do things with Flux which are not possible with these other frameworks (like hacking the algorithms…)

Flux exploits all the power of Julia’s composability to use automatic differentiation to train the models.

We shall focus on understanding Deep Neural Networks in Flux.

Useful videos: Building Deep Learning models in Flux , Flux introduction (Doggo…) ,

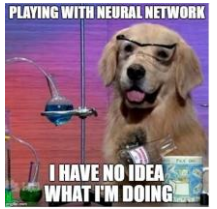


But what is a Deep Neural Network?

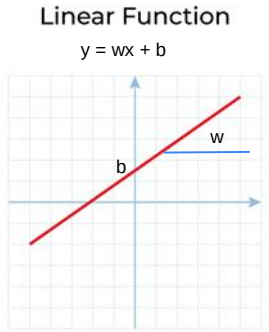
Three Blue One Brown Neural networks – YouTube

It’s nonlinear algebra.

The language of Neural Networks is full of terminology that can obfuscate what things really are.

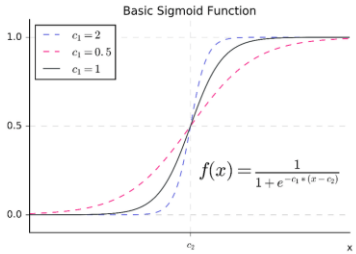


What is a neuron?



The equation of a line, `y = w · x + b` , is linear. This is studied in Linear Algebra. w is the slope and b is the intercept with the y axis at x = 0. In machine learning these are called “weight” and “bias”.

The equation `y = sigma( w ·x + b)` , where sigma is a nonlinear function (like a sigmoid), represents a “bent line”. The nonlinear function sigma is called the “activation function”. The previous equation is the equation of a single “neuron”.

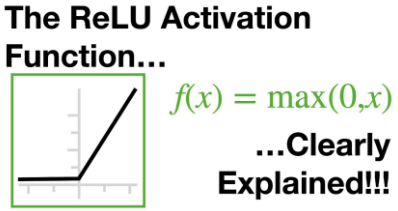


If we compose linear functions with linear functions we get a linear function (principle of superposition, a key tenet of linearity)

If we compose nonlinear functions with nonlinear functions we get a neural network.

But what is a Deep Neural Network?

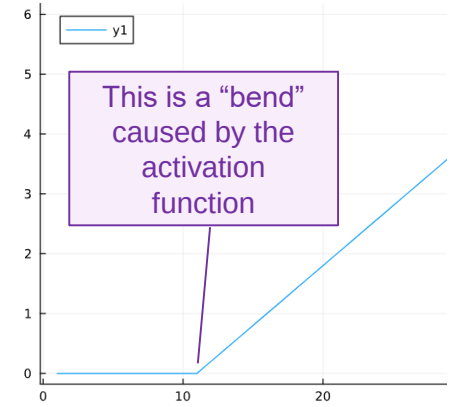
Neural Networks Pt. 3: ReLU In Action!!! - YouTube



The simplest neural network

A very simple (and useful) activation function is the ReLu – Rectified Linear Unit. (another fancy name for something as simple as saying “if it’s negative make it 0”).

```
relu (generic function with 1 method)
  relu(x) = max(0, x)

y (generic function with 1 method)
  y(x, w, b) = w * x + b

neuron (generic function with 1 method)
  neuron(x, w, b) = relu(y(x, w, b))
```




```
plot([neuron(i, 2, 2) for i in -2:.1:2])
```

Any nonlinear activation function “bends” the line defined by the equation

`y = w · x + b`

The composition of a neuron function with another neuron function rotates and shifts the curves at each level of nesting (the effect of the “weights” and “biases”) and the activation functions apply successive “bends” to the curves.

If you create a concatenation of neuron compositions you get a “deep” one-dimensional neural network of equation (in this case with 2 “layers”):

“Prediction” = `DNN(x) = ReLU(w2 · ReLU(w1·x + b1) + b2)` for x in test range.

If you can now find the values of the w’s and the b’s that minimize the error prediction you have just trained your neural network.

Linear regression “loss” space

Fitting a Line · Flux (fluxml.ai)

Pluto code  *https://github.com/flt-acdesign/Mathematical_tourism_with_Julia/blob/main/Linear_regresion_error_space_GOOD.jl*

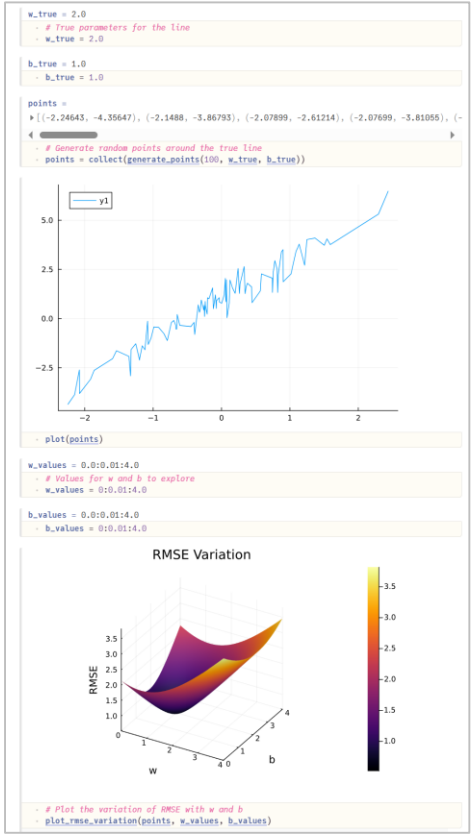
```
using Plots ✓

rmse (generic function with 1 method)
# Function to compute root mean square distance
function rmse(points, w, b)
    n = length(points)
    distances = [abs(y - w * x - b) for (x, y) in points]
    return sqrt(sum(distances.^2) / n)
end

generate_points (generic function with 2 methods)
# Function to generate random 2D points
function generate_points(n, w_true, b_true, noise=0.5)
    x_vals = sort(randn(n))
    y_vals = w_true .* x_vals .+ b_true .+ noise * randn(n)
    return zip(x_vals, y_vals)
end

# Enter cell code...

plot_rmse_variation (generic function with 1 method)
# Function to plot the variation of RMSE with w and b
function plot_rmse_variation(points, w_vals, b_vals)
    rmse_matrix = zeros(length(w_vals), length(b_vals))

    for (i, w) in enumerate(w_vals)
        for (j, b) in enumerate(b_vals)
            rmse_matrix[i, j] = rmse(points, w, b)
        end
    end


    print(rmse_matrix)

    surface(w_vals, b_vals, rmse_matrix, xlabel="w", ylabel="b", zlabel="RMSE",
    title="RMSE Variation")

    #heatmap(w_vals, b_vals, rmse_matrix, xlabel="w", ylabel="b", zlabel="RMSE",
    title="RMSE Variation", c = :roma100)

    #rmse_matrix
end
```

“Training” the Neural Network becomes an optimization problem where we want to minimize some form of aggregate error of the model with respect to the training set of points (we want to find the “best” parameters of the model, the w’s and the b’s)

Let’s plot the error (“loss”, in AI parlance) function of a purely linear approximation to a set of points as a function of the parameters w and b of the line

This code has been generated using ChatGPT3.5 with the prompt below (after “regenerating” 3 times) and manually edited.

Prompt: *Please write a julia code that computes the root mean square distance between a cloud of 2D points and a line defined as y = w*x + b and then plot the variation of the root mean square distance as a function of the parameters w and b*



Learning a simple function with a simple Neural Network

Pluto code  https://github.com/flt-acdesign/Mathematical_tourism_with_Julia/blob/main/Flux_quadratic_function_GOOD_MUSIC.jl

```
using Flux ✓, Plots ✓, PlutoUI ✓, Sound ✓  # import libraries for machine learning and plotting
```

```
1×100 Matrix{Float32}:
 -1.77889  1.64984  0.295516  -0.134305  0.115341  …  0.941935  -0.47048  -0.519015
```

```
begin
    # Generate training data
    x_train, y_train = generate_data(100)  # obtain the data for training (1oo points)
    # convert the data into a row vector for Flux
    Y_train = Float32.(reshape(y_train, 1, :)) # Flux prefers Float32 data (for GPU)
    X_train = Float32.(reshape(x_train, 1, :))
end
```

```
ground_truth (generic function with 1 method)
```

```
# Define the ground truth, the underlaying true data
ground_truth(x) = 3 * x^3
```

```
generate_data (generic function with 1 method)
```

```
# Generate the training data using the ground truth function and add noise
function generate_data(n)
    x_vals = randn(n)  # generate an array of random numbers
    y_vals = ground_truth.(x_vals) + randn(n)  # add some Gaussian noise to the truth
    return x_vals, y_vals  # return the training points as x and y values
end
```

Learning a simple function with a simple Neural Network

```
10
@bind n_neurons Slider(1:20, default = 10, show_value = true )
```

86.5 ms

```
model = Chain(
          Dense(1 => 10, relu),                # 20 parameters
          Dense(10 => 10, relu),               # 110 parameters
          Dense(10 => 1),                      # 11 parameters
        )                   # Total: 6 arrays, 141 parameters, 948 bytes.
```

```julia
# Define the neural network as a multilayer perceptron
model = Chain(   # stack layers of "neurons"
    Dense(1, n_neurons, relu),  # a layer with 1 input, n_neurons "neurons"
    Dense(n_neurons, n_neurons, relu),  # a layer with n_neurons inputs and n_neurons "neurons"
    Dense(n_neurons, 1)  # output layer, taking n_neurons inputs from the previous layer into a single value
)
```

```
▸ [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
```

```julia
begin
    # Define neural network training parameters
    loss(x, y) = Flux.mse(model(x), y)  # Use a mean squared error loss
    optimizer = Descent(0.01)  # Choose an optimizer
    num_epochs = 100    # number of training iterations
    training_loss = zeros(num_epochs) # allocate a vector to store the loss history
end
```

```
err_Ground_vs_Model (generic function with 1 method)
```

```julia
# function giving the difference between the ground truth and the model at a point x
err_Ground_vs_Model(x) = abs(ground_truth(x) - (model(reshape(Float32.([x]), 1, :)))'[1])
```

Learning a simple function with a simple Neural Network

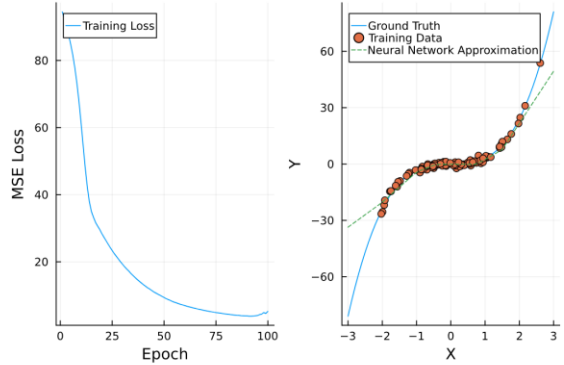
```
begin
    # Train the neural network using Flux
    for epoch in 1:num_epochs   # iterate the training "num_epochs" times
        Flux.train!(loss,               # specify the loss function to use
                    Flux.params(model), # pass the model weights and biases
                    [(X_train, Y_train)], # pass the set of training data
                    optimizer) # use the optimizer function defined before

        current_loss = Flux.mse(model(X_train), Y_train) # current loss
        training_loss[epoch] = current_loss # store loss per iteration

        # Make a "chord" with frequencies corresponding to loss and local errors ar
        some points
        beep(hcat([current_loss], [err_Ground_vs_Model(p) for p in [-1 -.5 0 .5
        1].*500] )) # [current_loss and local errors]

    end
end
```

100%

100%

100%

Learning a simple function with a simple Neural Network




```
begin

# create an independent set of x points to check the model (different from the points
used in the training data to check for overfitting and other issues)
    x_test = -3:.1:3  # a range from .3 to 3 in steps of 0.1

# Create plot for the training loss evolution through the "epochs"
     loss_evolution = plot(training_loss, xlabel="Epoch", ylabel="MSE Loss",
    label="Training Loss", legend=:topleft)

# Create a plot with the ground truth, training data and model (neural
network)approximation
        model_fit = plot(x_test, ground_truth.(x_test), label="Ground Truth",
        xlabel="X", ylabel="Y", legend=:topleft)

        scatter!(x_train, y_train, label="Training Data", xlabel="X", ylabel="Y",
        legend=:topleft)

        plot!(x_test, (model(reshape(Float32.(collect(x_test)), 1, :)))',
        label="Neural Network Approximation", linestyle=:dash)

# Plot the two plots from above in a grid with 1 row and 2 columns
    plot(loss_evolution, model_fit, layout = (1,2))
end
```

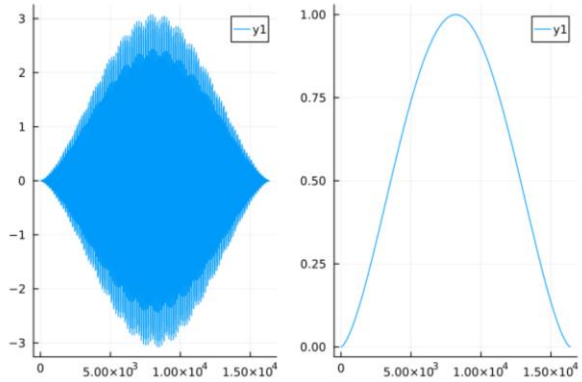
1.6 ms

Playing sounds

Pluto code  *https://github.com/flt-acdesign/Mathematical_tourism_with_Julia/blob/main/Play_sound_simple_GOOD.jl*

```
beep (generic function with 1 method)
```

```
function beep(freq_array::Array; t = .02, plot_wave = false)

    S = 8192 # sampling rate in Hz
    scale = 1/t

    modu  = @. sin( 2π*(1:S÷scale) * .5*scale/S )^1.5 # Modulation shape

    wave(f) = @. modu * (sin( 2π*(1:S÷scale) * f/S) ) # Define function to create wave

    wave_array = reshape([wave(f) for f in freq_array], length(freq_array),1)

    wavr = sum(wave_array, dims=1)[1] # right channel
    wavl = wavr # left channel

    soundsc([wavr wavl], S) # scale to unit amplitude

 if plot_wave plot(plot(wavr), plot(modu),layout = (1,2)); end


end
```




```
beep([500 1500 200 150]; t = 2, plot_wave = true) # test
```

2.3 s

DataFrames - library for data processing

Pluto code  *https://github.com/flt-acdesign/Mathematical_tourism_with_Julia/blob/main/DataFrames_example_GOOD*

DataFrames - library for data processing

```
using DataFrames ✓
```

```
using Random ✓, Statistics ✓
```

```
▸TaskLocalRNG()
# Generate some random data
Random.seed!(42)  # for reproducibility
```

```
n = 10
n = 10
```

```
data =
```

	ID	Name	Age	Score
1	1	"Person 1"	31	86
2	2	"Person 2"	27	37
3	3	"Person 3"	27	69
4	4	"Person 4"	28	88
5	5	"Person 5"	21	7
6	6	"Person 6"	32	63
7	7	"Person 7"	38	25
8	8	"Person 8"	29	72
9	9	"Person 9"	23	96
10	10	"Person 10"	24	82

```
data = DataFrame(
    ID = 1:n,
    Name = ["Person $i" for i in 1:n],
    Age = rand(20:40, n),
    Score = rand(0:100, n)
)
```

112 μs

```
# Display the DataFrame
println("Original DataFrame:")
```

```
Original DataFrame:
```

```
display(data)
```

```
10×4 DataFrame
 Row │ ID     Name       Age    Score
     │ Int64  String     Int64  Int64
─────┼──────────────────────────────────
   1 │     1  Person 1      31     86
   2 │     2  Person 2      27     37
   3 │     3  Person 3      27     69
   4 │     4  Person 4      28     88
   5 │     5  Person 5      21      7
   6 │     6  Person 6      32     63
   7 │     7  Person 7      38     25
   8 │     8  Person 8      29     72
   9 │     9  Person 9      23     96
  10 │    10  Person 10     24     82
```

DataFrames - library for data processing

```
display(data)
```

```
10×4 DataFrame
 Row │ ID     Name       Age    Score
     │ Int64  String     Int64  Int64
─────┼──────────────────────────────────
   1 │     1  Person 1      31     86
   2 │     2  Person 2      27     37
   3 │     3  Person 3      27     69
```

```
ages = [31, 27, 27, 28, 21, 32, 38, 29, 23, 24]
```

```
# Accessing columns
ages = data.Age
```

```
display(ages)
```

```
10-element Vector{Int64}:
 31
 27
 27
 28
 21
```

```
["C", "A", "A", "B", "C", "C", "B", "B", "A", "A"]
```

```
# Adding a new column
data[!, :Grade] = rand(["A", "B", "C"], n)
```

```
println("\nDataFrame with a new 'Grade' column:")
```

```
DataFrame with a new 'Grade' column:
```

```
display(data)
```

```
10×5 DataFrame
 Row │ ID     Name       Age    Score  Grade
     │ Int64  String     Int64  Int64  String
─────┼─────────────────────────────────────────
   1 │     1  Person 1      31     86  C
   2 │     2  Person 2      27     37  A
   3 │     3  Person 3      27     69  A
```

DataFrames - library for data processing

```
filtered_data =
```

	ID	Name	Age	Score	Grade
1	1	"Person 1"	31	86	"C"
2	6	"Person 6"	32	63	"C"
3	7	"Person 7"	38	25	"B"

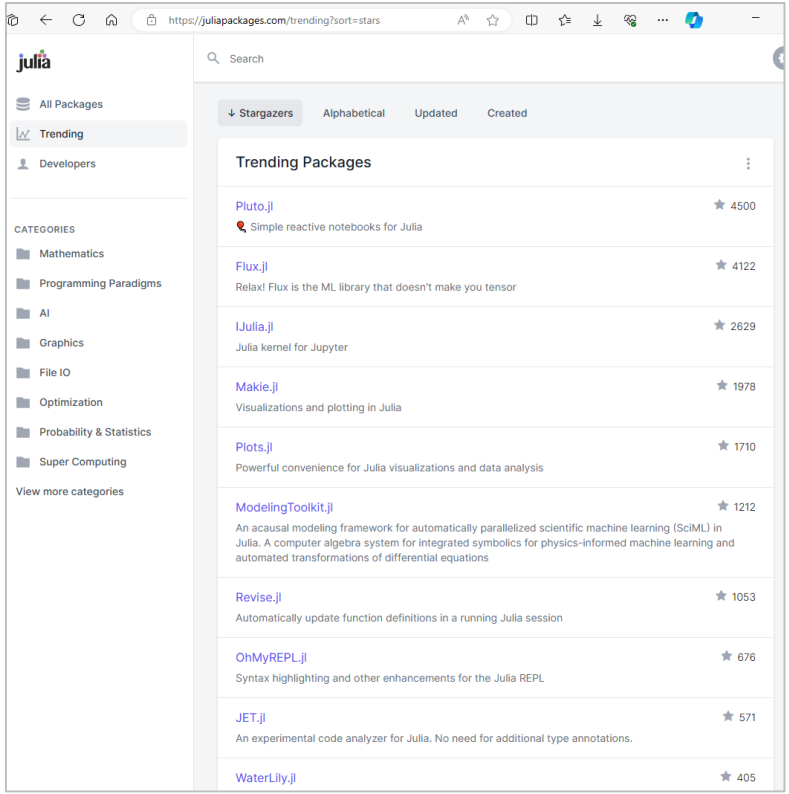
```
# Filtering data
filtered_data = filter(row -> row.Age > 30, data)
```

```
display(filtered_data)
```

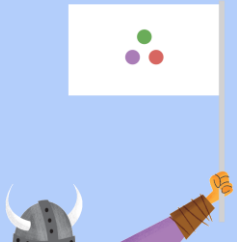
```
3×5 DataFrame
 Row │ ID     Name      Age    Score  Grade
     │ Int64  String    Int64  Int64  String
─────┼──────────────────────────────────────
   1 │     1  Person 1     31     86  C
   2 │     6  Person 6     32     63  C
   3 │     7  Person 7     38     25  B
```

Other notable packages

Gridap Solving Partial Differential Equations (PDE), including FE, CFD, electromagnetism, etc…



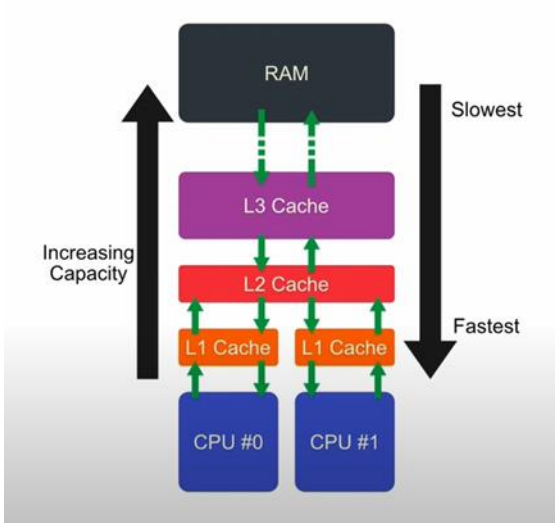
High performance Julia



High performance

Performance Tips · The Julia Language

Hardware & Software Basics (HPC in Julia 1/10) - YouTube



Measuring time

Profilers

Code Optimization

Vectors, arrays and “vectorization” (or “broadcasting”)

Type declaration

Columns first

Static arrays

Broadcasting, map, reduce etc…

Parallelization CPU - Multithreading

[08x03] Intro to Julia Multi-Threading | How to use the Threads.@threads macro - YouTube

generic

Parallelization GPU – Basic concepts

kernels

Parallelization GPU - Array (or generic) programming

kernels

Parallelization GPU – Kernel functions

kernels

Mandelbrot GPU

https://github.com/flt-acdesign/Mathematical_tourism_with_Julia/blob/main/Mandelbrot_GPU_CUDA_good

```
begin
    # Using the CUDA.jl occupancy API, from the example at
    #
https://github.com/JuliaGPU/CUDA.jl/blob/cb4a6b03ff443a062212c7a8aad260cfbf134410/src/
indexing.jl#L37
    Z2 = CuArray{Float64}(undef, height, width);
    # Manual invocation, with fixed set of threads:
    # CUDA.@sync @cuda threads=1024 kernel_mandelbrot!(Z2);

    kernel = @cuda name="mandelbrot" launch=false kernel_mandelbrot!(Z2)
    config = launch_configuration(kernel.fun)
    threads = Base.min(length(Z2), config.threads)
    blocks = cld(length(Z2), threads)
    # The @benchmark macro is optional (again, it causes an error. RCLOct23)
    CUDA.@sync kernel(Z2; threads=threads, blocks=blocks)
end
```

Fijado por doggo dot jl

@viniciusfreitasdealmeida3663 hace 7 meses

Yay, first time getting featured in a video! What an honour :D

I confess I don't remember really well how I implemented the transforms and interpolation expressions, but in retrospective it **does** look a lot like spaghetti code, haha! I'd be happy to go through it and refactor to make it clearer, tough. Also, feel free to give me any feedbacks on how to improve it!

Explaining code using Large Language Models (e.g. GPT)

UE

can you explain the following Julia code which uses the CUDA package? Z2 = CuArray{Float64}(undef, height, width);
Manual invocation, with fixed set of threads:
CUDA.@sync @cuda threads=1024 kernel_mandelbrot!(Z2);

kernel = @cuda name="mandelbrot" launch=false kernel_mandelbrot!(Z2)
config = launch_configuration(kernel.fun)
threads = Base.min(length(Z2), config.threads)
blocks = cld(length(Z2), threads)
# The @benchmark macro is optional (again, it causes an error. RCLOct23)
CUDA.@sync kernel(Z2; threads=threads, blocks=blocks)

Certainly! The provided Julia code involves the use of the CUDA.jl package to perform GPU-accelerated computations for generating the Mandelbrot set. Let's break down the code step by step:

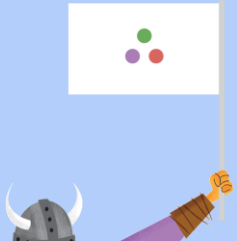
1. `Z2 = CuArray{Float64}(undef, height, width);`: This line creates an uninitialized CUDA array (`CuArray`) of type `Float64` with dimensions `height` by `width`. The Mandelbrot set will be stored in this array.
2. `kernel = @cuda name="mandelbrot" launch=false kernel_mandelbrot!(Z2)`: This line creates a CUDA kernel object named "mandelbrot" using the `@cuda` macro. The `launch=false` option is used to prevent immediate launching of the kernel.
3. `config = launch_configuration(kernel.fun)`: This line retrieves the launch configuration for the CUDA kernel. It is used to determine the number of threads and blocks that will be used to execute the kernel.
4. `threads = Base.min(length(Z2), config.threads)`: This line calculates the number of threads to be used. It takes the minimum of the total length of `Z2` (total number of elements in the array) and the number of threads obtained from the launch configuration.
5. `blocks = cld(length(Z2), threads)`: This line calculates the number of blocks to be used. It uses the `cld` function (short for "ceiling division") to ensure that there are enough blocks to cover all elements of the array.
6. `CUDA.@sync kernel(Z2; threads=threads, blocks=blocks)`: This line synchronously launches the CUDA kernel on the GPU with the specified number of threads and blocks. The `@sync` macro ensures that the execution on the GPU is complete before proceeding.

In summary, this code sets up a CUDA array, creates a CUDA kernel object for the Mandelbrot computation, determines the launch configuration, calculates the number of threads and blocks, and then launches the kernel on the GPU using the specified configuration. The result, the Mandelbrot set, is stored in the `Z2` array on the GPU.

More GPU code for Mandelbrot

aurelio-amerio/Mandelbrot.jl: Julia library to compute and plot nice and colorful Mandelbrot Fractals. Uses multi-threading and GPU. (github.com)

Learn Julia



Basic learning resources

Doggo, MIT Computational Thinking, Rackaukas, Julia conferences

Vectorization

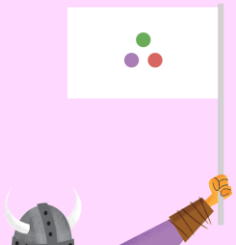
En español

Fast Track to Julia: Syntax summaries

More resources

Conclusions

Backup



UE

please write a windows script so that, using a shortcut in the desktop, I can launch the Julia executable and then make it execute import Pluto; Pluto.run()

Certainly! You can create a Windows script to achieve this using a batch file. Follow these steps:

1. **Create a new text file:**
   Right-click on your desktop, go to "New," and choose "Text Document." Name the file something like `RunPluto.bat`.
2. **Edit the batch file:**
   Right-click on the batch file you just created and select "Edit" to open it in Notepad or any text editor.
3. **Write the script:**
   Enter the following lines in the batch file:

batch — Copy code

```
@echo off
cd /d "C:\Path\To\Your\Julia\Executable"  REM Change this to the actual path

julia -e "import Pluto; Pluto.run()"
```

Make sure to replace `"C:\Path\To\Your\Julia\Executable"` with the actual path where your Julia executable is located.

1. **Save and close the file:**
   Save the changes and close the text editor.
2. **Create a shortcut:**
   Right-click on the batch file (`RunPluto.bat`) and choose "Create shortcut." You can rename the shortcut to something like "Run Pluto."
3. **Assign a shortcut key:**
   Right-click on the shortcut, go to "Properties," and in the "Shortcut" tab, you can set a shortcut key combination in the "Shortcut key" field. For example, you can use Ctrl + Alt + P.
4. **Run the script:**
   Now, you can use the shortcut key combination to launch Julia with the specified command.

Make sure that the path to your Julia executable is correct, and if the path contains spaces, you should enclose it in double quotes in the batch file. Also, note that you might need to have the Julia executable in your system's PATH or provide the full path to the Julia executable in the script.

- cmd /k “aslkjas/pluto.bat”
- Pin to start
- Julia – t auto –e “Import…”