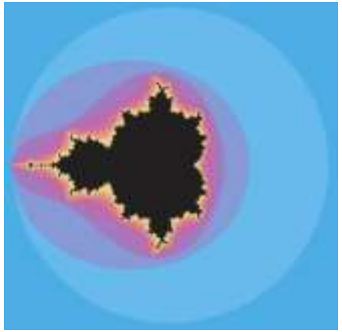


An introduction to the julia language for researchers

Universidad de Sevilla
Oct 2023

Raul Llamas

raulcarlos.llamas@universidadeuropea.es

Introduction

Course repository in Github

What is Julia?

A language for Scientific Computing

Two language problem (▶What makes Julia so awesome?)

Speed comparison

Why is it called Julia?

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

Why it may be interesting for you?

Speed

Easy to learn and use

Extremely elegant language

Some specific characteristics of Julia

Multiple Dispatch

Composability

LLVM

Mainly oriented to Functional Programming, not object oriented

Language comparison:

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

  https://cheatsheets.quantecon.org/

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

	Matlab	Python	Julia
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

A Python killer?

Enter Mojo

Mandelbrot in Julia vs Mojo

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

```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()
```

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

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

    return iters


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

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

        @parameter
        fn compute_vector[simd_width: Int](col: Int):
            """Each time we 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

Boeing

COVID

Black Holes

First hexaflop dynamic language

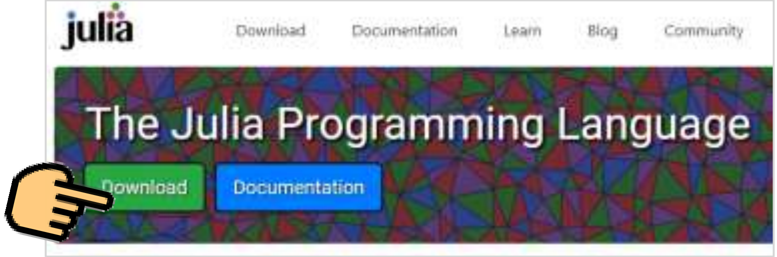
Installation

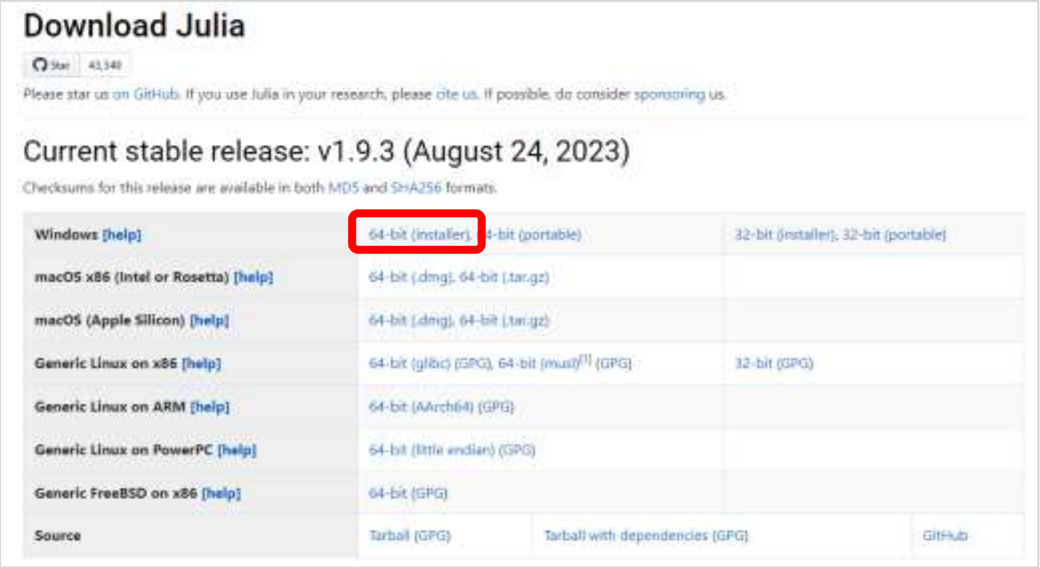
Installing Julia

[How to install Julia and Visual Studio Code]

1 Go to [The Julia Programming Language (julialang.org)]

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





Download Julia

Star 43,348

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

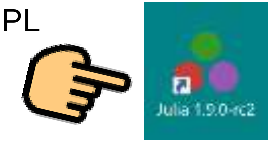
Current stable release: v1.9.3 (August 24, 2023)

Checksums for this release are available in both MD5 and SHA256 formats.

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

3 Accept all the proposed settings during the installation.

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



REPL and package manager

The Julia REPL

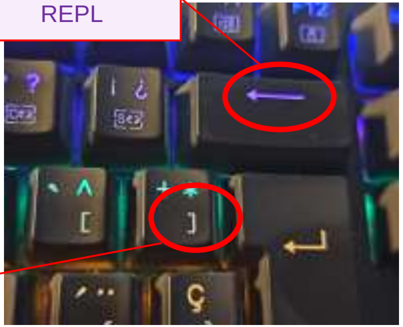
"REPL" stands for Read Evaluate Print Loop and it's where you are just after starting Julia.

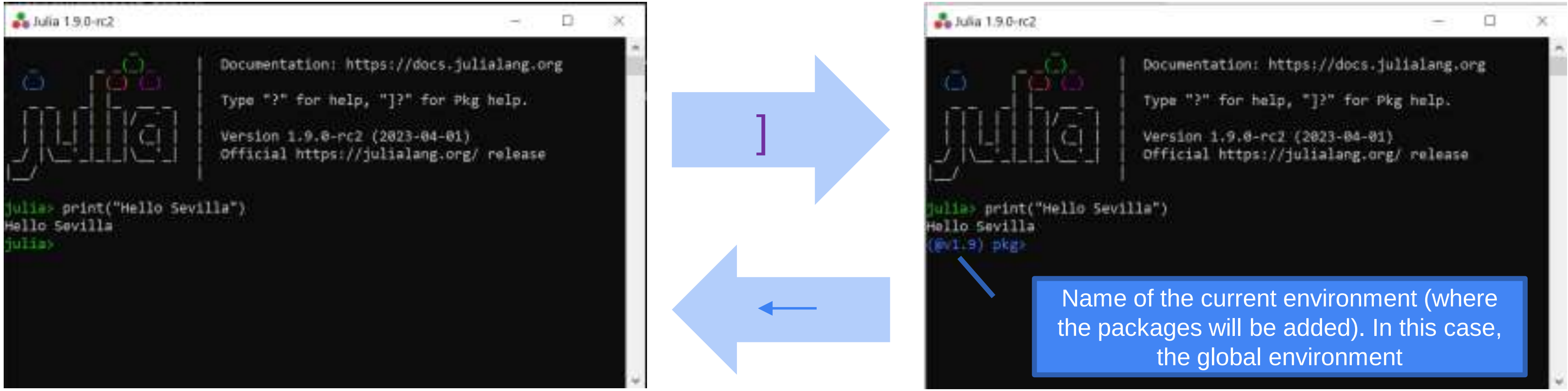
It is the most basic way in which Julia can be written using the "console"

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

The Julia package manager enables to install “packages” (libraries of code), update them, delete them and some 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 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 several 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 add them.

```
add Flux BenchmarkTools Images
```

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

Installing and starting Pluto

Julia + Pluto

First Experiments

Exploring Pluto

Help, reading files, downloading files, status, export options, sample notebooks

Getting help in Julia

Getting Help in Julia

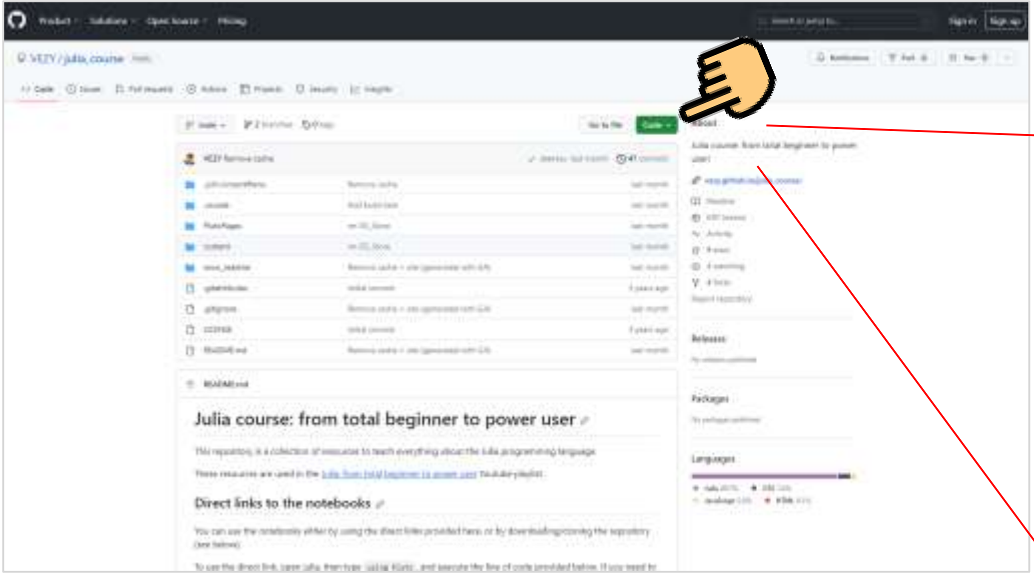
First experiments with Julia

Plots

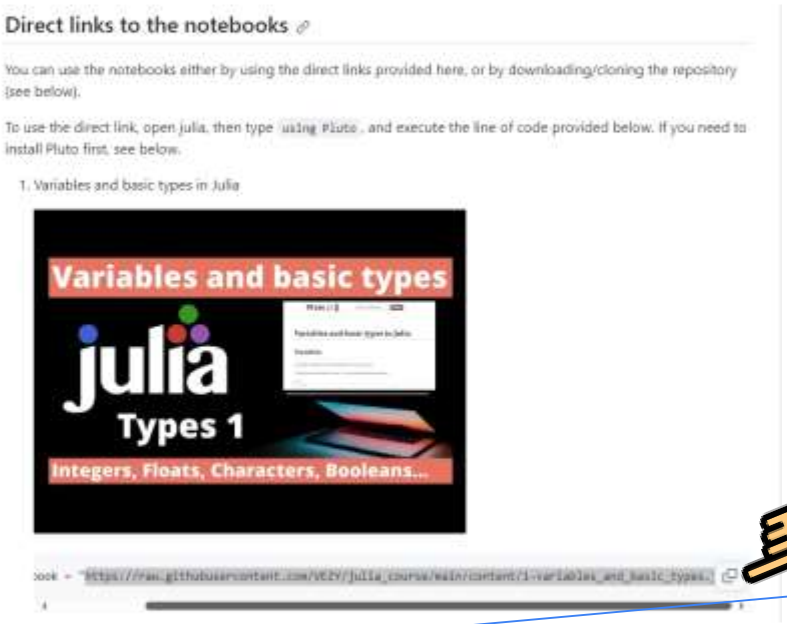
emojis

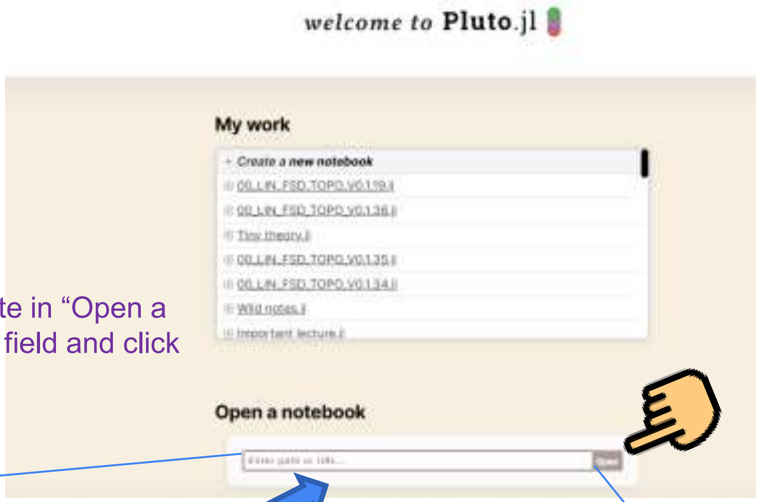
Course notebooks (Rémy Vezy)

[Github repository for this course]



Opening the course notes in Pluto





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)

The 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

Special symbols and idiosyncrasy

$ interpolate value of a variable into a string of text
: define symbol, range or "all values in array dimension"
begin … end
@ macro
; don't print result, separator for keyword arguments in functions, separator for vectors in an array definition
comment
""" …. """ comment block of code
( ) Function parameters
[ ] Array indices
{ } Variable type
c:\\file.txt equivalent to c:\file.txt in Windows
md" " Markdown
|> pipe operator
⟹ Dictionary assignment
… splat operator
(x,) definition of tuple
:: Type definition
! Mutating function (convention) ("Bang!")
. Broadcasting operator ("dot operator")
Julia is 1-indexed
Greek letters and symbols *** tab

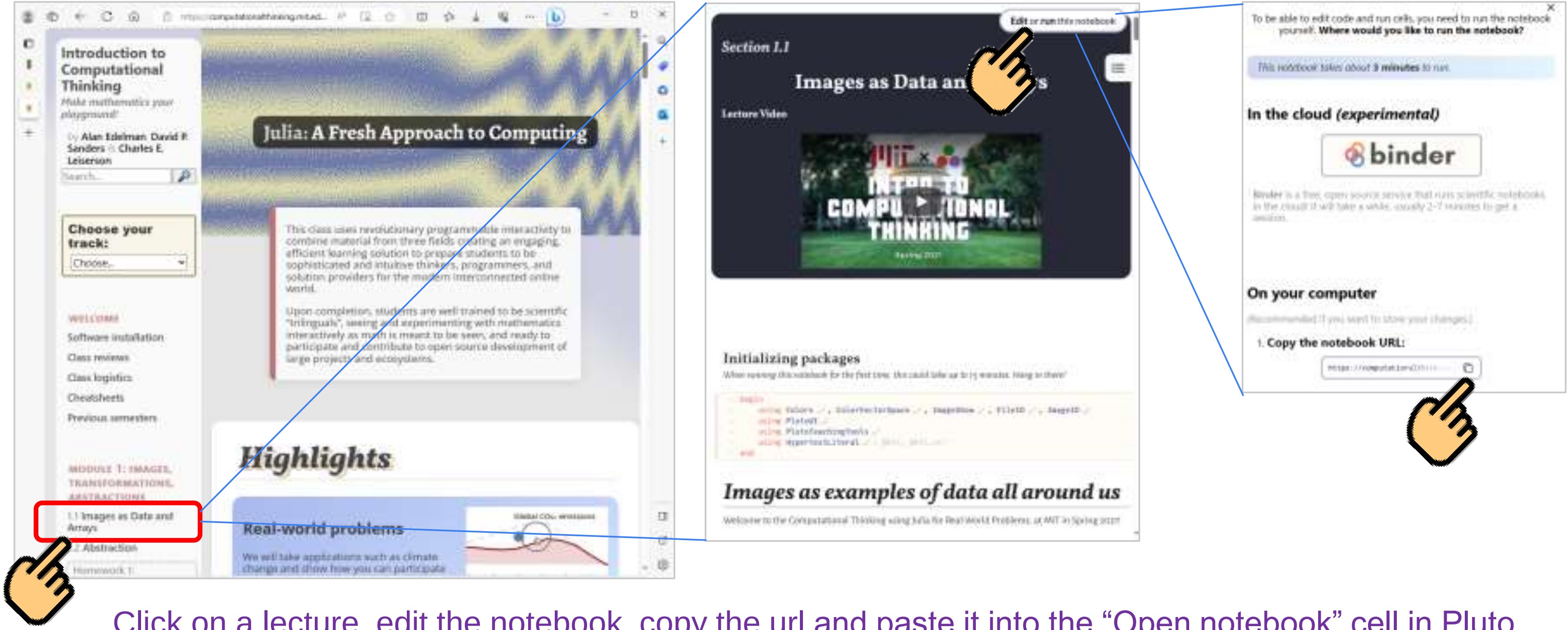
Error messages

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*

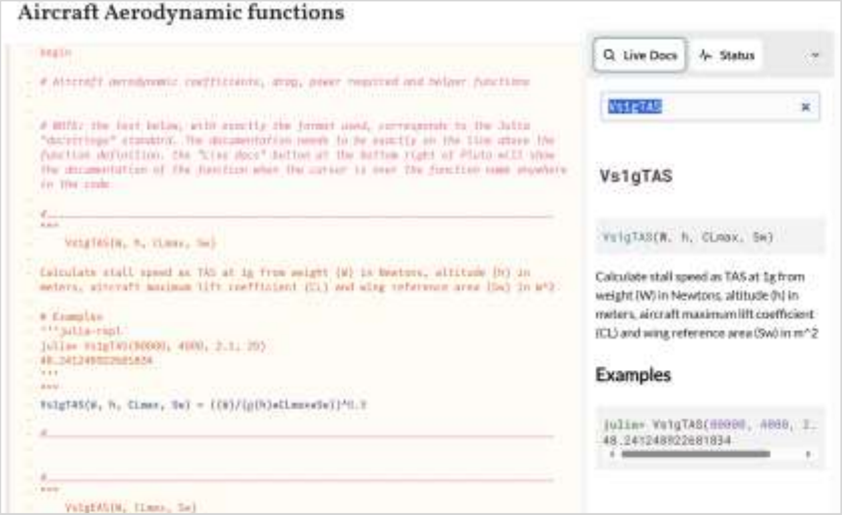
Interactive notebooks for education

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

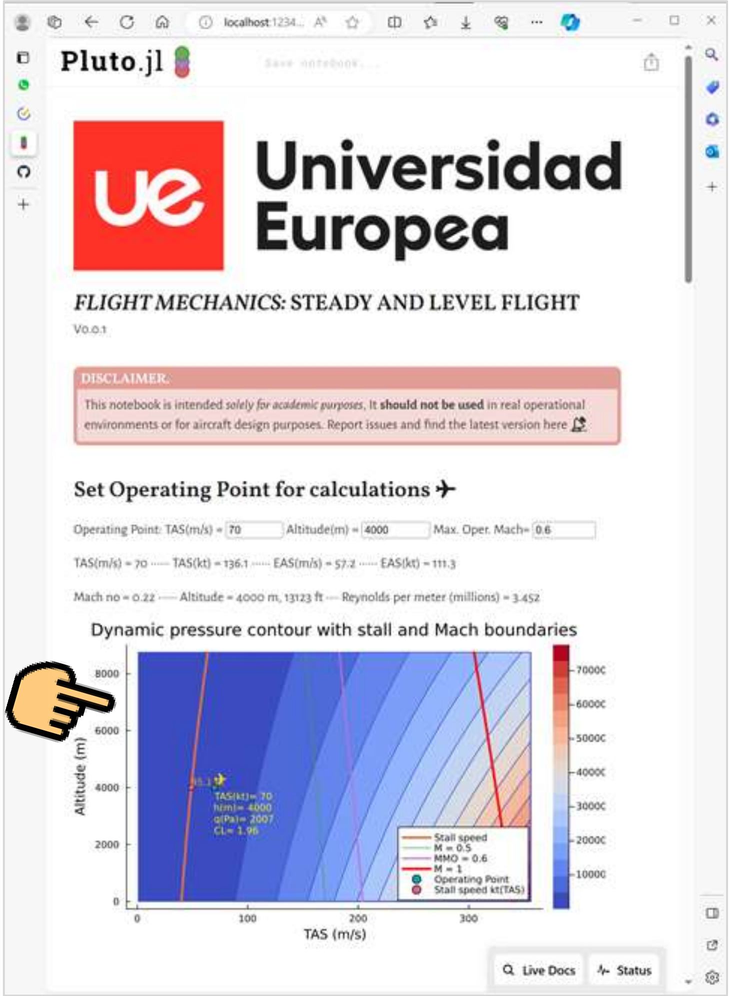
This Pluto notebook is a simple example of an interactive resource to teach aircraft performance.

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



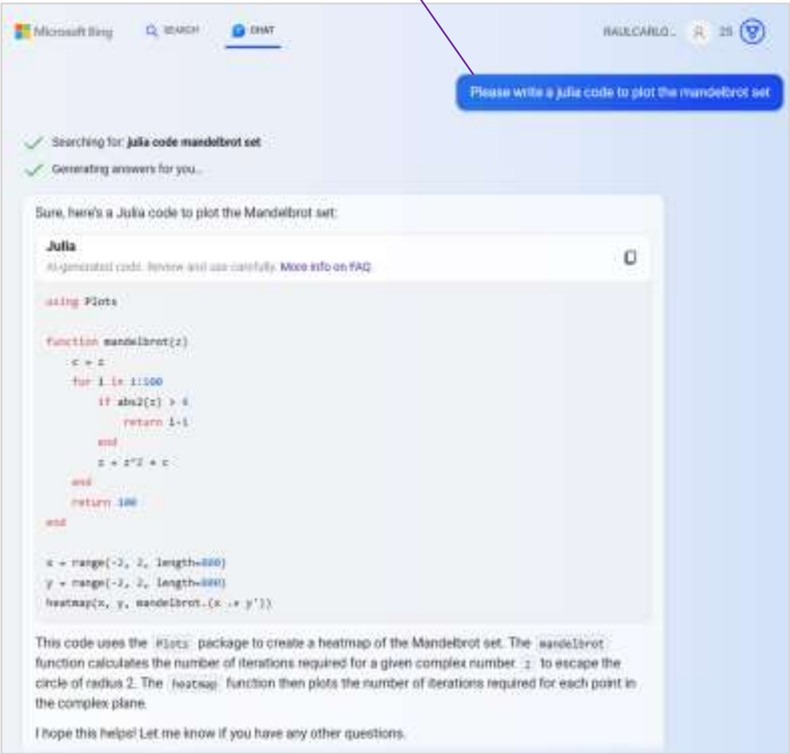
Enter chat GPT (or Bing chat or…)

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

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

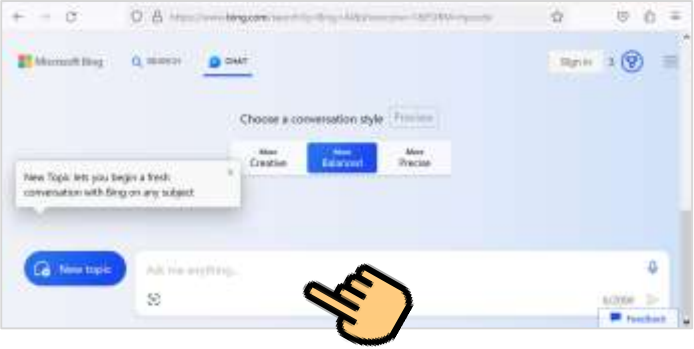
However, be prepared to edit the code manually or keep asking for refinements (“Regenerate”) until it works.



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

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

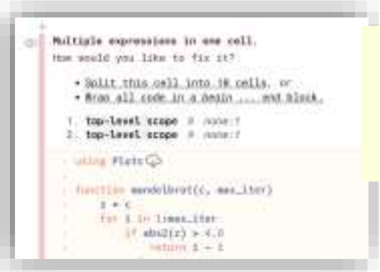
https://www.bing.com/search?q=Bing+AI&showconv=1&FORM=hpcodx



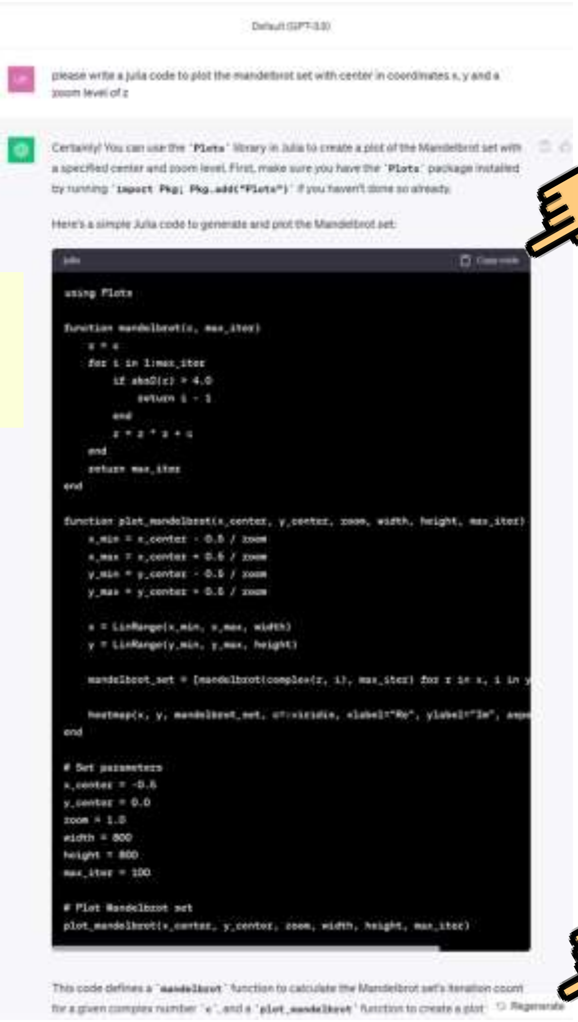
Mandelbrot set – Raw GPT3.5 code

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)
```

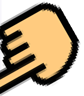


When you get this message in Pluto, click on “Split this cell into…”



Click to copy the code

If you don’t like the code, regenerate it



A refined version of Mandelbrot starting from GPT code

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

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

```
1000
```

```julia
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)
```

```julia
# 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
```

A refined version of Mandelbrot

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

```julia
# 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
```

A refined version of Mandelbrot

Set zoom 10^zoom = -0.4

```
md"Set zoom 10^zoom = $(@bind zoom Slider(-.6:.01:7 ;default=-.4, show_value=true) ) "
```

δ = 2.511886431509580130949643717030994594097137451171875

```
δ = BigFloat(1/(10^zoom))  # Convert the "zoom" into the width of the interval
```

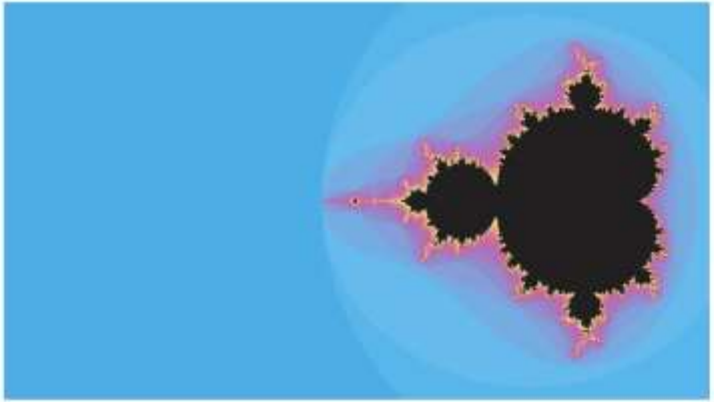
xcenter = -1.75 ycenter = 0.0

```

md"
xcenter = $(@bind x_center Slider(-3:.01:1; default = xcvideo , show_value=true) )
ycenter = $(@bind y_center Slider(-1:.01:1; default = ycvideo , show_value=true) )
    "

```

108.18144970414201

```
begin

#setprecision(BigFloat, 600)  # Enable this line to set arbitrary precision on the BigFloat type. Default value is 128 bits.

# The array to be plotted is the return value of the generate_mandelbrot function, note that the array needs to be transposed in order to obtain the usual orientation of the Mandelbrot set
array_to_plot = transpose(generate_mandelbrot(
                                        BigFloat(x_center),
                                        BigFloat(y_center),
                                        δ,
                                        width, aspect_ratio, max_iter))

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

end
```

A refined version of Mandelbrot



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

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

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
```

Coordinates from this video https://www.youtube.com/watch?v=aSgzDb3jF_4

```
begin
    # Default coordinates in the complex plane of the center of the image
    xcvideo =
    BigFloat(-1.7499576837068935036022145060706997072711087972626287793024283782028606
    80829728048872186727844317008311886445076556695313797475419999999995)

    ycvideo =
    BigFloat(0.0000000000000000002787937065633794021782947537900944364927985054600163081
    3790439306501893868497662021604774705522013257723324547269999999995)

    md" Coordinates from this video  https://www.youtube.com/watch?v=aSgzDb3jF_4"
end
```

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

Lotka-Volterra differential equations using Euler solver

```julia
using Plots
```

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

```julia
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)
```

```julia
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)
```

```julia
# Parameters
α, β, γ, δ = 0.1, 0.02, 0.1, 0.01
```

```
u0 = [100.0, 20.0]
```

```julia
# Initial conditions
u0 = [100.0, 20.0]
```

```
T = 1000.0
```

```julia
# Time span and step size
T = 1000.0
```

```
dt = 0.1
```

```julia
dt = 0.1
```

```
(0.0:0.1:1000.0, 10001×2 Matrix{Float64}:)
                  100.0          20.0
```

```julia
# Solve using Euler method
t_values, u_values = lotka_volterra_euler(α, β, γ, δ, u0, T, dt)
```

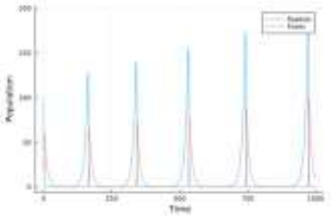
The **Lotka–Volterra equations**, also known as the **Lotka–Volterra predator–prey model**, are a pair of first-order nonlinear differential equations, frequently used to describe the dynamics of biological systems in which two species interact, one as a predator and the other as prey. The populations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

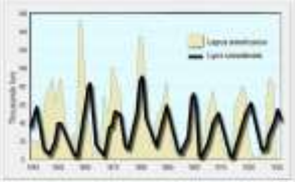
$$\frac{dx}{dt} = \alpha x - \beta x y,$$
$$\frac{dy}{dt} = \delta x y - \gamma y,$$

[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.





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.

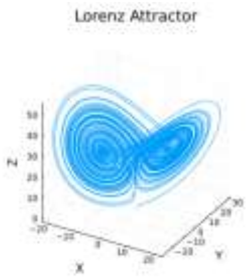
Lorentz Attractor – Euler method

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

The Lorenz system

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sigma y - \sigma x,$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = \rho x - xz - y,$$
$$\frac{\mathrm{d}z}{\mathrm{d}t} = xy - \beta z.$$

The 3 governing equations of the Lorenz system



Getting serious with Julia

Type system

Multiple dispatch: Functions and Methods

- Type safety @warn_code

Vectors and arrays

Type declaration

Columns first

Static arrays

Broadcasting, map, reduce etc…

LLVM

Macros

Inline functions

The reason why Julia can be faster than C

Visual Studio Code

The Julia ecosystem

Packages

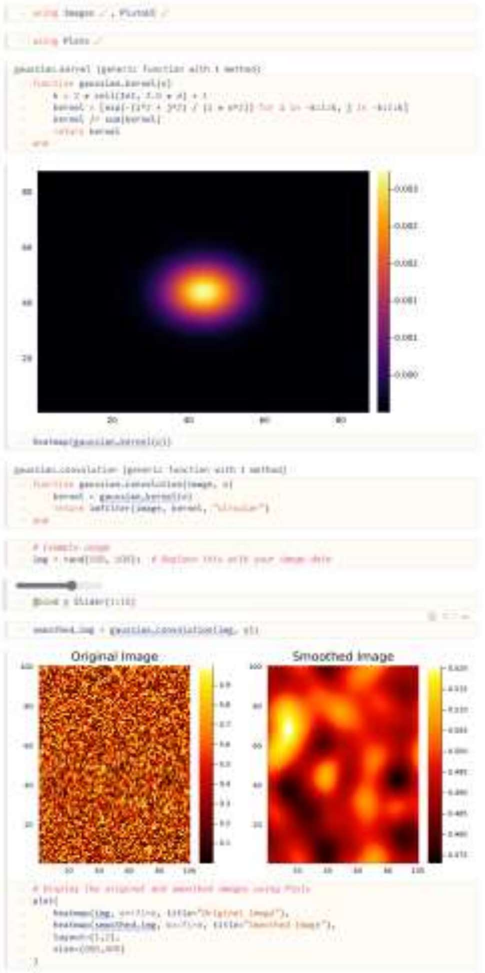
Packages documentation in Julia

Creating local environments

In this case we are adding the “Pluto” package

Image convolution using Images

- Convolution 3blueone..



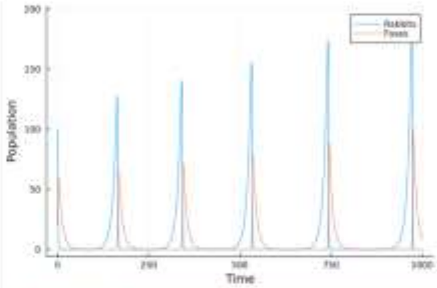
https://github.com/flt-acdesign/Mathematical_tourism_with_Julia/blob/main/gaussian_convolution_GOOD2_withlayout.jl

Jump

Differential Equations

Lotka-Volterra Euler

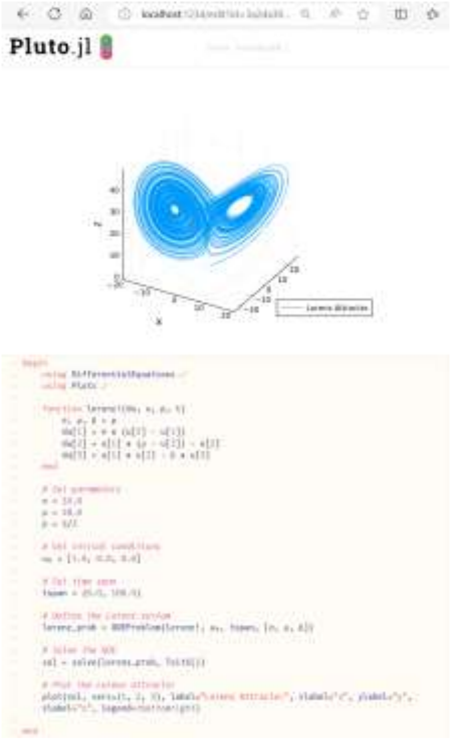
```julia
using Plots

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)
```




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

Lotka-Volterra DifferentialEquations

Lorenz attractor – DifferentialEquations package



Machine Learning

[Welcome · Flux (fluxml.ai)](https://fluxml.ai)

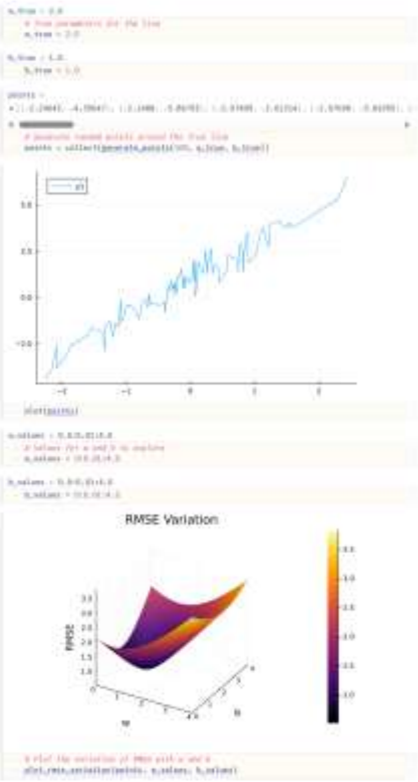
- What is a neuron and a neural network?

Linear regression “loss” space

[Fitting a Line · Flux (fluxml.ai)](https://fluxml.ai)

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 (and after “regenerating” 3 times)

```julia
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
```



Learning a simple function with a simple Neural Network

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

```
1×100 Matrix{Float32}:
 -0.427811  -0.938948  0.0349867  -2.65148  …  1.38439  0.386605  1.02273  -0.344687
```

```julia
begin
    # Generate training data
    x_train, y_train = generate_data(100)  # obtain the data for training (100 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)
```

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

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

```julia
# 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
```

10

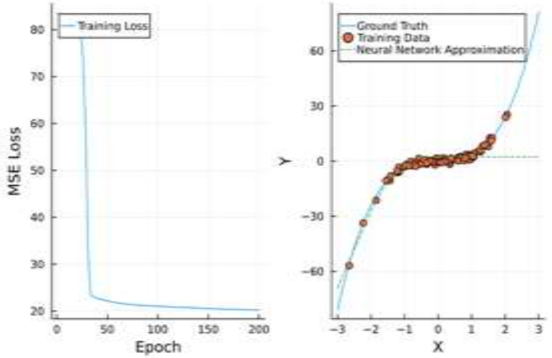
```julia
@bind n_neurons Slider(1:20, default = 10, show_value = true )
```

```
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
)
```

Learning a simple function with a simple Neural Network

```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 = 200     # number of training iterations
    training_loss = zeros(num_epochs) # allocate a vector to store the loss history
end
```




```julia
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
```

```julia
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

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

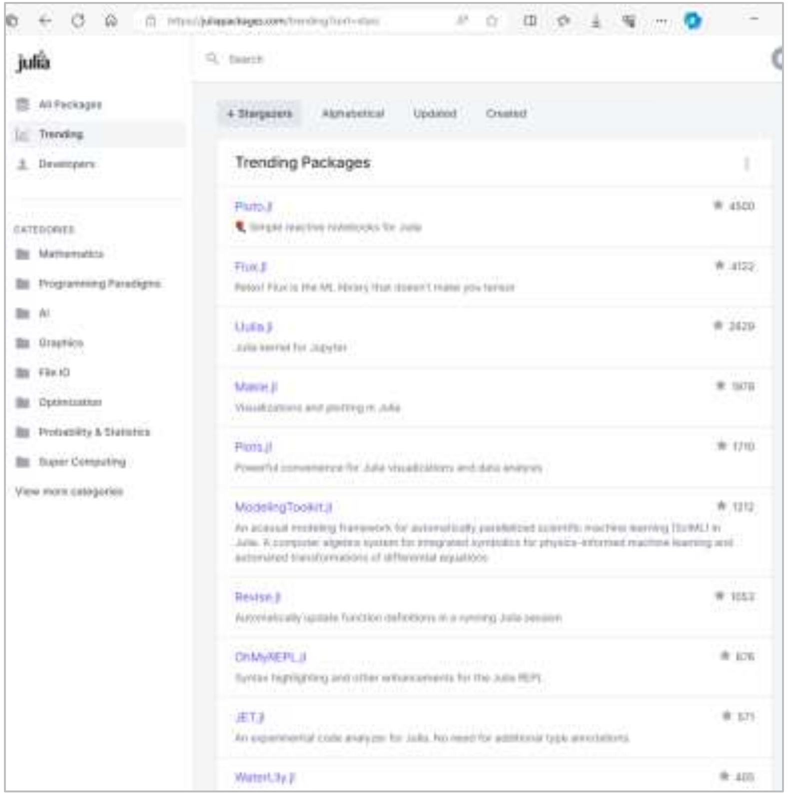
100%

100%

DataFrames

Other notable packages

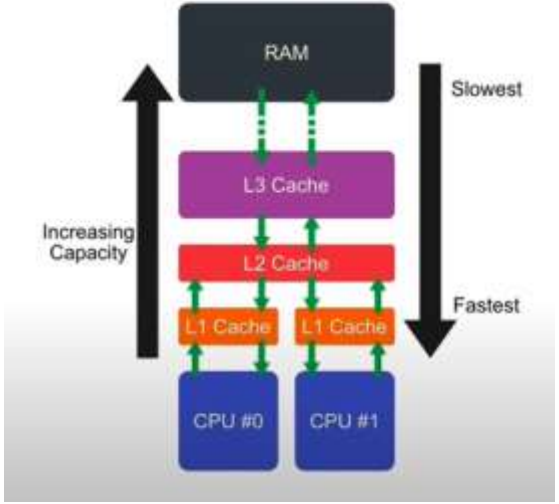
Gridap Solving Partial Differential Equations (PDE), including FE, CFD, electromagnetism, etc…



High performance Julia

High performance

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



Measuring time

Profilers

Code Optimization

Parallelization CPU

generic

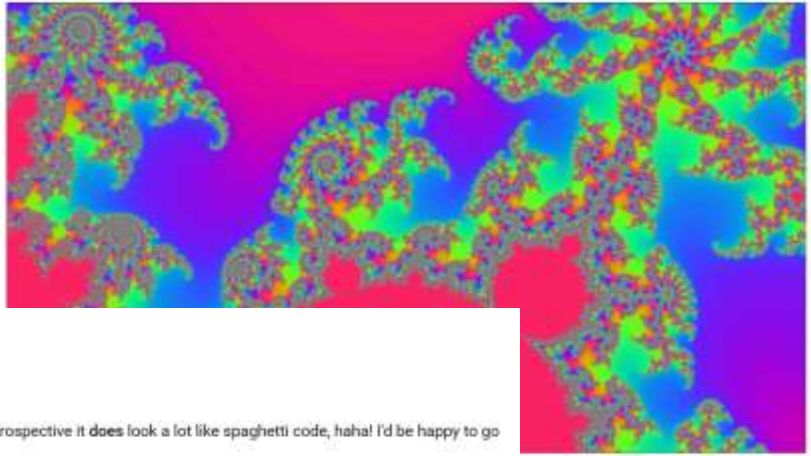
Parallelization GPU

kernels

Mandelbrot GPU

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

```julia
begin
    # Using the CUDA.jl occupancy API, from the example at
    #
https://github.com/JuliaGPU/CUDA.jl/blob/cb4a6b03ff443a062212c7a8ead260cfbf134410/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 LLM

Certainly! This Julia code appears to be part of a CUDA kernel for computing the Mandelbrot set on the GPU. Let's break down the code step by step:

1. `id = (blockIdx().x - 1) * blockDim().x + threadIdx().x`: This line calculates a unique thread identifier (`id`) based on the thread indices and block indices. It's a common way to map 2D or 3D thread indices to a linear index.
2. `stride = blockDim().x * gridDim().x`: The `stride` is calculated to determine the step size for each thread to iterate through the elements of the array. It represents the total number of threads in the grid.
3. `Nx, Ny = size(Z)`: Extracts the dimensions of the array `Z`.
4. `cind = CartesianIndices((Nx, Ny))`: Creates a set of Cartesian indices for a 2D array of size `(Nx, Ny)`.
5. `for k=id:stride:Nx*Ny`: This loop iterates over the elements of the array `Z`, assigning each thread a unique range of indices.
6. `i = cind[k][1]` and `j = cind[k][2]`: Extracts the row and column indices from the linear index `k`.
7. `x = ax*j + bx` and `y = ay*i + by`: Applies an affine transformation from screen space to world space. The transformation involves scaling (`ax` and `ay`) and translation (`bx` and `by`).
8. `c = x + y*im`: Constructs a complex number `c` based on the transformed coordinates.
9. `z = c`: Initializes a variable `z` with the value of `c`.
10. `iterations = 0`: Initializes a counter for the number of iterations.
11. `while CUDA.abs2(z) < 400.0 && iterations < max_iter`: Executes a loop to calculate the Mandelbrot set. The loop continues as long as the square of the absolute value of `z` is less than 400.0 and the number of iterations is less than the specified maximum (`max_iter`).
12. `z = z^2 + c`: Updates the value of `z` based on the Mandelbrot iteration formula.
13. `iterations += 1`: Increments the iteration counter.
14. `@inbounds Z[i, j] = iterations`: Updates the corresponding element in the Mandelbrot set array `Z` with the number of iterations. The `@inbounds` macro is used to indicate that array bounds checking is not necessary.
15. `return nothing`: This function appears to be a GPU kernel, and it doesn't explicitly return a value. The result is stored in the array `Z`.

In summary, this code is part of a GPU kernel that calculates the Mandelbrot set for a specified range of indices using an affine transformation from screen space to world space. The result is stored in the array `Z`.

Explaining code using LLM

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.

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

More GPU code for Mandelbrot

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

UL

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

```batch
@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.