1 Identify your problem

1.1 Define the scientific goal

Before writing any code you must be able to answer the following questions:

- What is my code suppose to do?
  If the code serves multiple puroses, split them in small independant units.
- What are the different ways to achieve this/theses goals?
  Temporal dynamic simulation, mathmatical optimisation, formal mathematical models, etc ...
- How reliable is my input data?
  Another way to put it: to what extend the code errors depend on things I have no control over.
- Is there a way for me to control:
  - the exactness of the results produced by the code
  - the determinism of the code (if any)

1.2 Choose the language

Various criteria must be taken into account in the choice of the programming language.

1.2.1 Open-source aspect

An open-source language is a language for which the content needed to improve or modify the language is available freely. Every one can contribute and use such a language.

These languages benefit from:

- A large community that can provide help
- The tracking of different versions and modifications of the language
- A pretty fast development cycle (time between two releases of the language)

Proprietary languages (in contrast with open-source languages) have the following drawbacks)

- Often very costly
- Code sharing made difficult because one must have bought the right tools
- Sometimles, partial documentation in order to protect an intellectual property

1.2.2 Running speed

Some languages are slower than other (sometimes up to 100 times slower). In many cases, slowness is the price to pay for a simpler syntax and fewer code lines. Regarding this running speed:

- It can often be **largely improved** using dedicated libraries.
- It depends on what you intend to do: different languages perfom best in different problems.
- It depends on the compiler/interpreter

Without using dedicated libraries, popular languages can be sorted out according to their typical running speed **approximately**:

1. C/C++
2. Julia
3. Matlab
4. Python
5. R

1.2.3 Versatility

Do you need to ...	Then: search for
Read/write files, manipulate strings	simple syntax
Interact with existing codes and softwares	dedicated libraries, large community
Store and retrieve easily a lot of data into memory	easy type specification, simple syntax
Perform a intensive computation	native language speed, dedicated libraries

Table 1: Versatility of the language: a simple guide

1.2.4 Understandable by other people

Your code must be understandle by people:

- Who work in the same field
- Whose training is similar to yours

note: using a rare language can prevent people from trying to help you.

1.3 Define a development plan

Software development is a full time activity. Software development on "spare time" between two other activites (for instance: experimental activites) is error prone. **It is less time consuming to develop in an organize way rather than correct lack-of-attention errors afterwards**.

Table 2: Proposal of development plan

Step	Prior thoughts	Actions	Test
1. Meet the the primary purpose of the code.	Data structure.	Reading/Declaration of input parameters. Main content of the code. Simplified presentation of results.	Using a test case: Are the results correct? Does the running time seem compatible with all the cases to be treated?
2. Structure the code	Possible interactions with the code. Aspect of results.	Definition of functions/classes/structures Errors handling, progress log.	Resilience to parameters change, easy results exploitation.
3. Optimize the code	I/O importance in the problem, memory indepandance.	Benchmark Speed up: algorithmic, variable types.	Better ressources use, with same code functionalities.
4. Document the code for your own interest	Sections of the code subjected to change	1. Short comments for difficult sections 2. Short description at the top of the file	Try to understand the code 2 weeks later.
5. Document for other users of the code	Typical use cases of the code	Writing documentation in an iterative process: 1. Important content 2. Optionnal content (see 3.3.1)	Ask somebody to try your code.

How to get organized:

- **Evaluate prior to any development** the time need for each development step.
  If you exceed this time, this might mean that you are out of the scope of the step.

- **Keep in mind the real aim of the code**
  This makes it possible to distinguish what is useful from what is detail.

2 Develop

2.1 Choose an IDE

2.1.1 Definition

An *Integrated development environment* is a GUI software that facilitates software development by:

- Communicating with the compiler/interpreter.
- Highlighting some of the development errors before compilation.
- Speeding up code writing using plugins and keyboard shortcuts.

Support for advanced features (version control (*git, svn*), code suggestion) are sometimes available.

2.1.2 Examples

Some IDE are language specific, other are multi-languages. Some famous IDE:

- Eclipse
- Visual Studio
- Spyder
- Netbeans
- PyCharm

Les languages fermés (ex: Matlab, VBA) viennent avec leur propre IDE.

2.1.3 Notes

Don't lose times on optionnal time consuming tasks! For instance, proper formatting of a file is done automatically by the IDE.

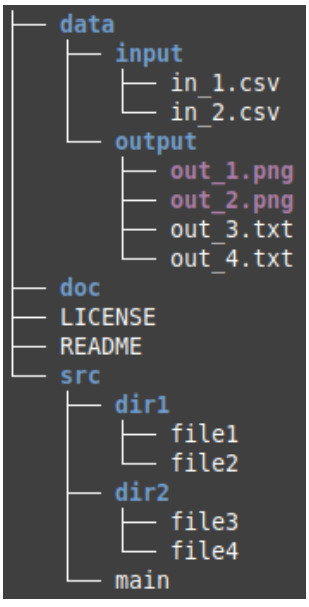
2.2 Organize your working space

A working space is made of:

- Source files and compiled files
- Input data: in your case, all the content with a scientific meaning, used by the program
- Data produced by the program
- Documentation: automatically generated files (html, texte, tex, ...)
- Configuration files: every fotware-related parameters needed for execution on your computer

Some important rules:

- Store all the source files together. Do not duplicate them.
- Separate data from source.
- Avoid file paths with accentuation and special characters.



Suggestion of working space

2.3 Universal development rules

Each language has some specific mandatory rules. Yet there exists come common good practices:

2.3.1 Language

Code must be written in English language since:

- The syntax of the language is English.
- Scientific community communicates using English.
- Development problems are described in English on dedicated forums.

2.3.2 Naming conventions

Some rules must be respected:

- In most of the languages, do not use any accentuation or special character.
- CamelCase or snake_case
  Same everywhere in the code.

camelCase

snake_case

nocase

Variables

In the common case, a variable name must be:

- Short ($\leq$ 25 characters)
- Explicit
- Without any references to its type
- Using plural form if it's a container-like

Incorrect	Correct
myvar	dyn_viscosity
a_very_long_var	input_paths
temporaryArraySTRING	initialConditions

Global variables are written
in UPPER_SNAKE_CASE.

In a software development process, an explicit variable name tells the user about the physical (or mathematical, etc ...) meaning of the variable. Each wisely-defined variable name represents a lot of saved time when using the code as it is understood with less effort. lit le code.

Functions and methods
The name of a function:

- **Starts with a verb** à l'impératif at the imperativ form.
- Does not mention the type of input or output data.

2.3.3 Indentation and spacing

Some languages have few constraints regarding indentation and spacing. In these cases:

- Do not exceed 100 characters as a length of code lines.
- Indent the same way instructions blocs that share the same structure, or let the IDE do it for you.

2.3.4 Nested code

Nested code is difficult to read. For instance, do not nest multiple calls to functions in a one-liner as follows:

```
Float32 solar_power = compute_solar_power(get_area(solar_panel),
            get_irradiance(
                        download_weather(get_actual_time())))
```

Instead, write as many lines as needed:

```
Float32 solar_area = get_area(solar_panel)
                     Time current_time = get_actual_time()
WeatherData Geneve_weather = download_weather(current_time)
Float32 Geneve_irradiance = get_irradiance(Geneve_weather)
Float32 solar_power = compute_solar_power(solar_area, Geneve_irradiance)
delete(solar_area, Geneve_weather, Geneve_irradiance)               # optional in most lan
```

2.3.5 Comments

Comments start with a special character or set of characters that is defined for each language. Yet, comment characters must not be inserted manually as:

- It is less readable (unwanted spaces at wrong places)
- It creates indentations error whenever you remove manually this comment sign
- Automatic removal by the IDE might not be possible

Instead, you must use the IDE shortcut for defining either line comments or block comments.

3 Comment and documentation

3.1 Differences

A comment is a short explanation in the code that helps any **developer** to understand a few instructions. A comment may be only temporary. A commented code is mush simpler to understand.

A documentation is an exhaustive explanation for the **user** of the code. The documented parts of the code are the one that constitute its API.

3.2 Comment a code

A comment must be inserted at least in the following cases:

1. Bug to be corrected
2. Local code improvement needed
3. The way a difficult bug was solved Exemple:

```
UInt64 simulation_timestep = get_timestep()  // UInt64 prevents overflow
```

4. Unusual instructions, ye tneeded. Exemple:

```
parse_request(parser=Parser.NO_DEFAULT, timeout=NULL) // default timeout (3 min)
                                                      // is too short
```

5. Reference to a scientific article or an important scientific methodology or result Exemple:

```
Array[Float32] fourier_coefs = adapted_fourier_transform() //cf DOI 04.52/j.fake.206.18
```

   This type of comments does not intend to replace an exhaustive scientific description of the code in a dedicated document.

6. Some parameters are specified or a section of code is made unavailable (i.e.: "commented out") Exemple:

```
Int8 thermal_diffusivity = 9 // get_thermal_properties() has to be fixed
```

Each IDE comes with specific keywords to make comments more readable. It is often the case of **fixme** (see 1) and **todo** (see 2).

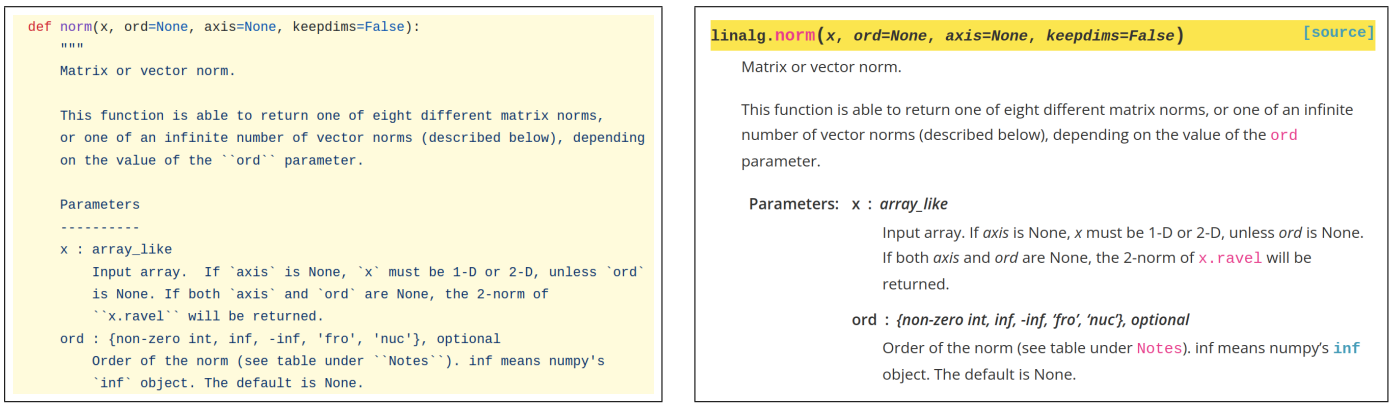
3.3 Document a code

3.3.1 Content of a documentation

The documentation section of a function must give some information about the following points, from most to less important:

1. What the function does
2. Input parameters
   - types
   - what they describe
   - the set of expected values, if any
   - the default value, if any
   - physical unit, if any
3. Same for output parameters
4. Related functions
5. Detail of the operations achieved by the function
6. Complete scientific references
7. Use case examples of the function

Points 1 to 3 are mandatory. The other are optional yet recommended, in particular 7.
Note that writting the documentation of a code is an iterative process: going through the entire code reveals some bug to fix, before going further into the documentation.



```
def norm(x, ord=None, axis=None, keepdims=False):
    """
    Matrix or vector norm.

    This function is able to return one of eight different matrix norms,
    or one of an infinite number of vector norms (described below), depending
    on the value of the ``ord`` parameter.

    Parameters
    ----------
    x : array_like
        Input array.  If `axis` is None, `x` must be 1-D or 2-D, unless `ord`
        is None. If both `axis` and `ord` are None, the 2-norm of
        ``x.ravel`` will be returned.
    ord : {non-zero int, inf, -inf, 'fro', 'nuc'}, optional
        Order of the norm (see table under ``Notes``). inf means numpy's
        `inf` object. The default is None.
```


Figure 1: Example of Python documentation: code side and HTML rendering.
(source: `https://numpydoc.readthedocs.io/en/latest/example.html`)

3.3.2 Documentation: howto

Documentation writing The documentation is written into the code. It is often located:

- Either at the line before the function definition
- Or in the function body

A particular syntax is adoptéed to reference a component of the code (variable, function, ...) or the language (types, ...).

Documentation generation Each documentation block is then rendered in an easy-to-read format:

- HTML: if documentation is hosted on the web `https://about.readthedocs.com/`
- Latex/PDF: if documentation is shared locally

The built document is called the "API reference" because it makes it possible to entirely use the software according to its API. Some famous documentation builders are:

- Javadoc (Java)
- Doxygen (C, C++)
- Sphinx (Python)

Other than the API reference, the user can find on the web some special documentations to learn a programming language or a specific library:

- Getting started : suggestions of code and important concepts that cover most of development needs
- Tutorials: achieve very specific things with the tool you learn

4 Example

This part shows the different development steps applied to a fictive math problem (parts 1, 2, 3) Let's solve the following differential equation:

$$(E) \quad \ddot{\theta} + w_0^2 \sin\theta = 0 \tag{1}$$

Given:

- θ a time dependant function, with $t \in [0, 10]$
- initial conditions:

$$\theta_0 = \theta(0) = \frac{\pi}{8} \tag{2}$$

$$\theta_0 = \theta'(0) = 0 \tag{3}$$

- w_0^2 takes one of the following values:

$$w_0^2 \in \{0.05, 0.5, 5, 50, 500\} \tag{4}$$

4.1 Identify one's problem

4.1.1 Define the scientific goal

Our code must be able to:

1. Read a parameter file
2. Solve a differential equation using the parameters specified by the file
3. Plot and save a diagram presenting the solution to the equation

A discretization on t makes it possible to get a solution close to the formal one. Conversely, a formal resolution would be difficult with the sin.

The input data (w_0^2) is reliable since these are independant parameters of typical amplitude regarding this type of equations.

The exactness of the result will be evaluated usng the approximation for small θ_0 values:

$$\theta : x \to \theta_0 \cos(w_0 x)$$

This problem is deterministic, and we shall check we get the same solution from one run to another either graphically or numerically.

4.1.2 Choose the language

This problem does not represent a heavy CPU task. Yet, discretization and numerically stable solving both require a dedicated library. *Julia* language is adaptated to analytic maths problems, using dedicated libraries such as *SciML* (solving) and *Plots* (plotting). This language is a high-level language, hence dealing with data input/output will be easy.

4.1.3 Define a development plan

Table 3 is a suggestion of development plan.
note: Parameters reading from the disk is not prioritary and could take place at the second step only.

Table 3: Proposal of development plan: exercise

Step	Actions	Duration
1. meet the the primary purpose of the code.	read and store parameters values Define and solve the equation. Perform aquick draft plot of the solution.	1 hour
2. Structure the code	Build an API. Display some progress information for the user. Customizz and save the results plot.	1 hour
3. Optimize the code	Profile and analyse the running time. Evaluate potential time savings. Speed up the code.	30 min
4. Document the code for your own interest	Description of every difficult instruction.	10 min
5. Document the code for other users of the code	Write documentation block of each function. Build as html.	30 min

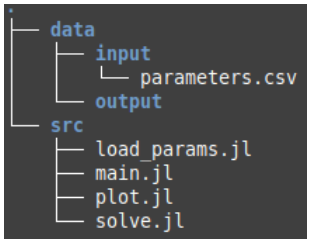
4.2 Develop

4.2.1 Preparation

Choose an IDE Regarding Julia language, the largest community IDE is Visual Studio Code with the dedicated plugin.

Organize your working space

For instance, let's create 3 source files, one for each basic functionality. All of them are supervised by the file *main.jl*.
Yet, at step 1, (see Table 3), all of the code is in *main.jl*. Code structuration befins at step 2.



Actual structure of the working dir.

4.2.2 Development

Step 1: meet the the primary purpose Equation 1 is transformed into 2 equations of first order:

$$\dot{\theta}(t) = \omega(t)$$
$$\dot{\omega}(t) = -w_0^2\theta(t)$$

Let's call u the result vector:

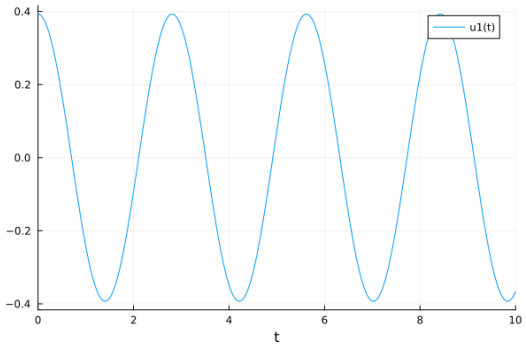
$$u = \begin{bmatrix} \theta \\ \omega \end{bmatrix}$$

Below, a first version of the code for step 1.

src > main.jl > ...

```julia
#= reading parameters
helper doc: https://csv.juliadata.org/stable/reading.html#CSV.read =#
using CSV
using DataFrames
parameters_path = joinpath(pwd(), "data", "input", "parameters.csv")

all_params = CSV.read(parameters_path, DataFrame, header=false)
all_params = all_params[!, "Column1"]


#= solving equation for one value of w_0_2
helper doc: https://docs.sciml.ai/DiffEqDocs/stable/types/ode_types/ =#
using DifferentialEquations
using Plots

w_0_2 = all_params[1]

function eq!(du, u, p, t)
    du[1] = u[2]
    du[2] = - p * u[1]
end
u0 = [π/8, 0]
tspan = (0.0, 10)
prob = ODEProblem(eq!, u0, tspan, w_0_2)

sol = solve(prob, Tsit5())


#= Plotting result =#
plot(sol, vars=1)
```

Code - step 1



θ as a function of time - step 1

Step 2: structure the code The following taks are achieved:

- Split up of the code in dedicated files.
- Use of functions
- Check of parameters compliance
- Display of log messages
- Custom of the plot, disk save

See figures 2 and 3.

Figure 2: Code - step 2

src > main.jl > ...

```julia
include("load_params.jl")
include("solve.jl")
include("plot.jl")


all_params = load_from_disk("parameters.csv")

results = Dict()
println()
for w_0_2 in all_params
    sol = solve_inplace(w_0_2)
    results[w_0_2] = sol
end

println("")
plot_custom(results, "first_parameters_set.png");
```

main.jl

src > load_params.jl > ...

```julia
#= reading parameters
helper doc: https://csv.juliadata.org/stable/reading.html#CSV.read =#
using CSV
using DataFrames

function load_from_disk(file_name::String)
    @info "Loading file" file_name
    parameters_path = joinpath(pwd(), "data", "input", file_name)
    all_params = CSV.read(parameters_path, DataFrame, header=false)
    if isempty(all_params)
        error("\nBad parameters: no values")
    end
    all_params = all_params[!, "Column1"]
    if typeof(all_params)!=Vector{Float64}
        error("\nBad parameters: type error, expected floats")
    end
    all_params
end
```

load_params.jl

src > solve.jl > ...

```julia
#= solving equation
helper doc: https://docs.sciml.ai/DiffEqDocs/
            stable/types/ode_types/ =#

using DifferentialEquations
using Plots


function solve_inplace(w_0_2::Float64)
    function eq!(du, u, p, t)
        du[1] = u[2]
        du[2] = - p * u[1]
    end
    u0 = [π/8, 0]
    tspan = (0.0, 10)
    @info "Solving problem" w_0_2
    prob = ODEProblem(eq!, u0, tspan, w_0_2)
    sol = solve(prob, Tsit5())
    return sol
end
```

solve.jl

src > plot.jl > ...

```julia
#= Plotting result =#
using Plots

function plot_custom(results, plot_name)
    @info "Building figures"
    results = collect(results)
    nbr_solutions = length(results)
    plots = []
    for (w_0_2, sol) in results
        pl = plot(sol, vars=1, label="w_0_2=$w_0_2", linewidth=1)
        push!(plots, pl)
    end
    nbr_cols = div(nbr_solutions, 2) + 1
    plot(plots..., layout=(2, nbr_cols), size=(800, 500), dpi=200)
    save_path = joinpath(pwd(), "data", "output", plot_name)
    savefig(save_path)
end
```

plot.jl

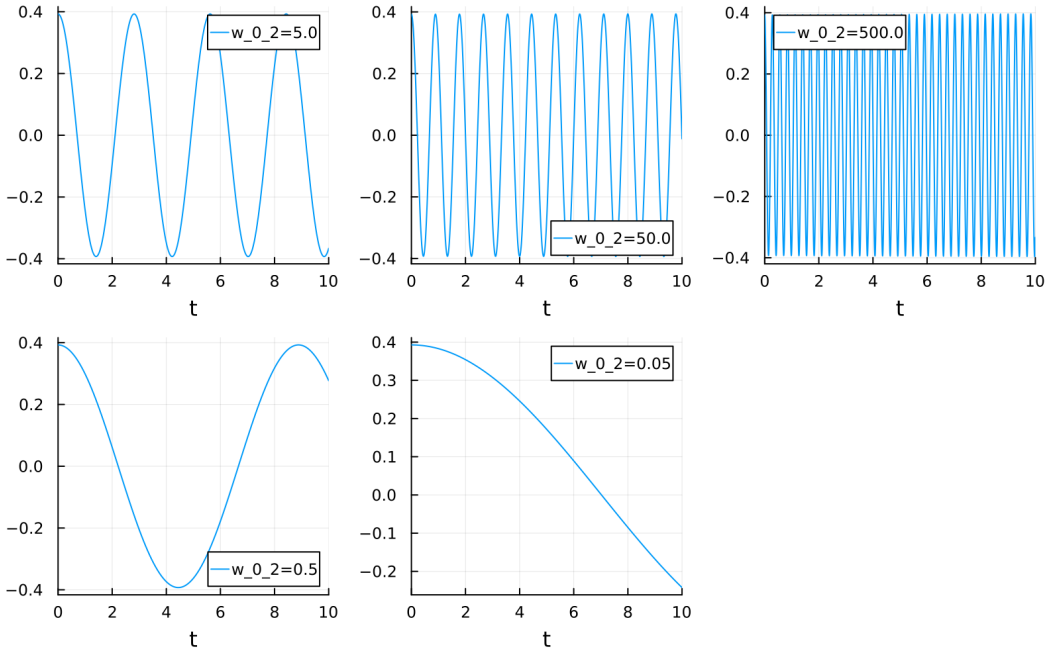


Figure 3: Visualization - step 2

Step 3: optimize the code

Measure the running time: overview

Every language, IDE and operating system comes with **benchmarking and profiling tools**. For this problem, the macro *@benchmark* is used. it returns a total execution duration and an estimation of memory usage.

```
BenchmarkTools.Trial: 15 samples with 1 evaluation.
 Range (min … max):  315.696 ms … 405.419 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     352.485 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   355.361 ms ±  29.252 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%

  316 ms           Histogram: frequency by time         405 ms <

 Memory estimate: 13.92 MiB, allocs estimate: 351244.
```

Results:

1. On average, the code runs takes 0.36s to run and uses 14 MiB of memory (*MiB=mebibyte*)
   → **Is-it too much?**

2. The standard deviation of running time is low.

Profile the running time

The macro *@profile* makes it possible to focus on time consuming sections of the code. Figure 4 shows a *FlameGraph* of this analysis.

Most of the running time comes from plotting and saving the results. This is an important part of the code that must be sped up. Two choices:

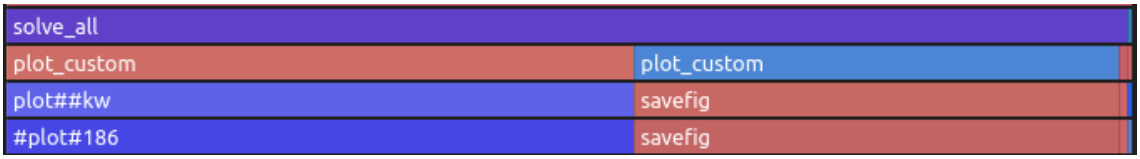
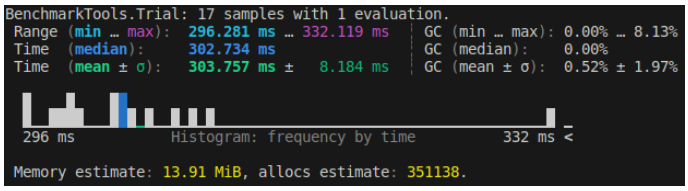
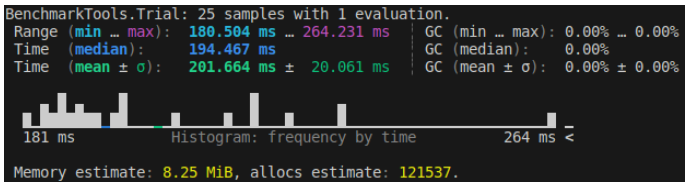


Figure 4: Running time decomposition

1. Specify some variable types to speed up code interpretation.
   - This is specific to Julia language
   - Saved time: 15%.
   - **does not alter functionalities of the code**.
2. Do not save the plot on disk.
   - Saved time: 30%.
   - **is a degradation of functionalities**.





Step 4: document the code for your own interest Will be seen on a Python case.

Step 5: document the code for other users of the code Will be seen on a Python case.