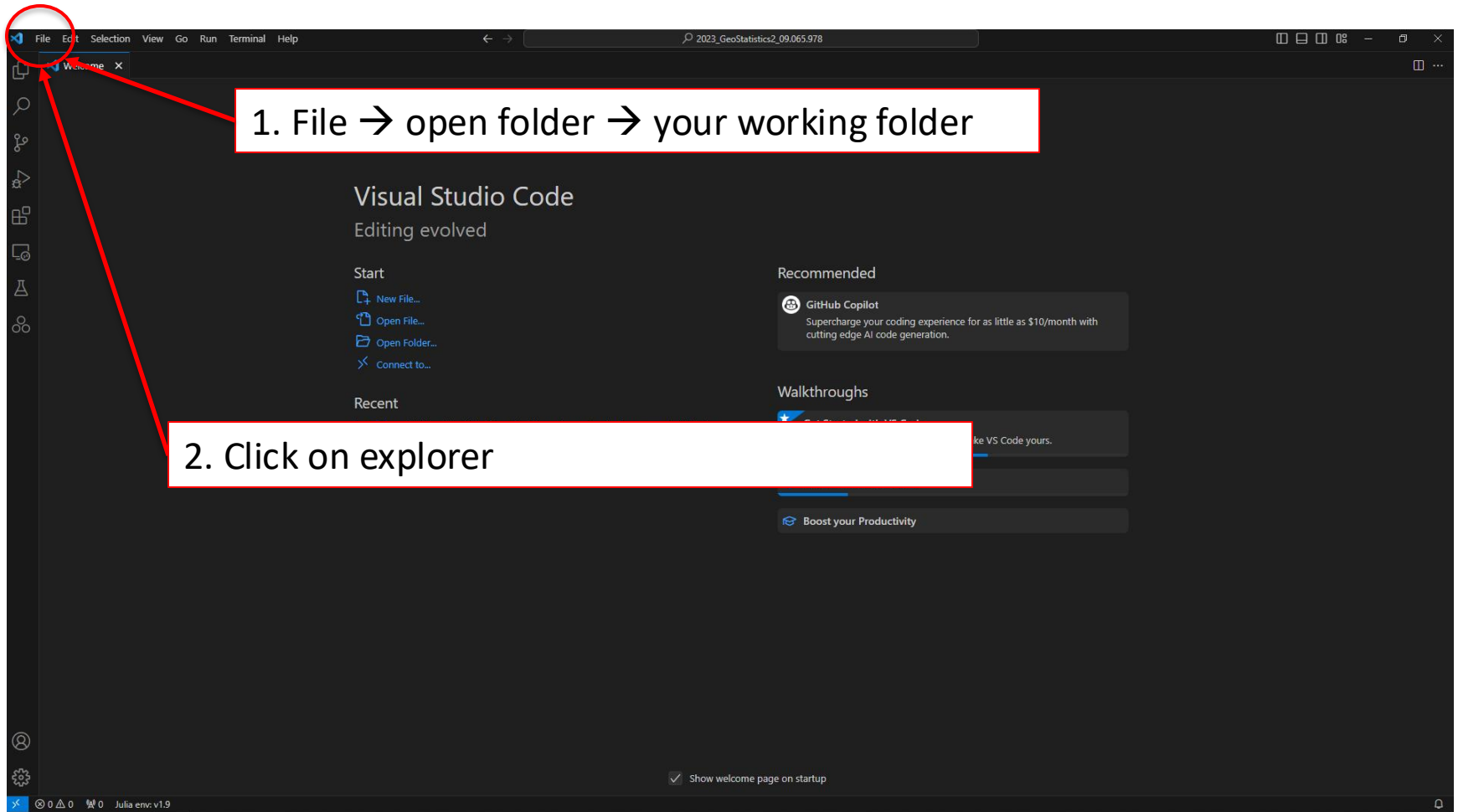


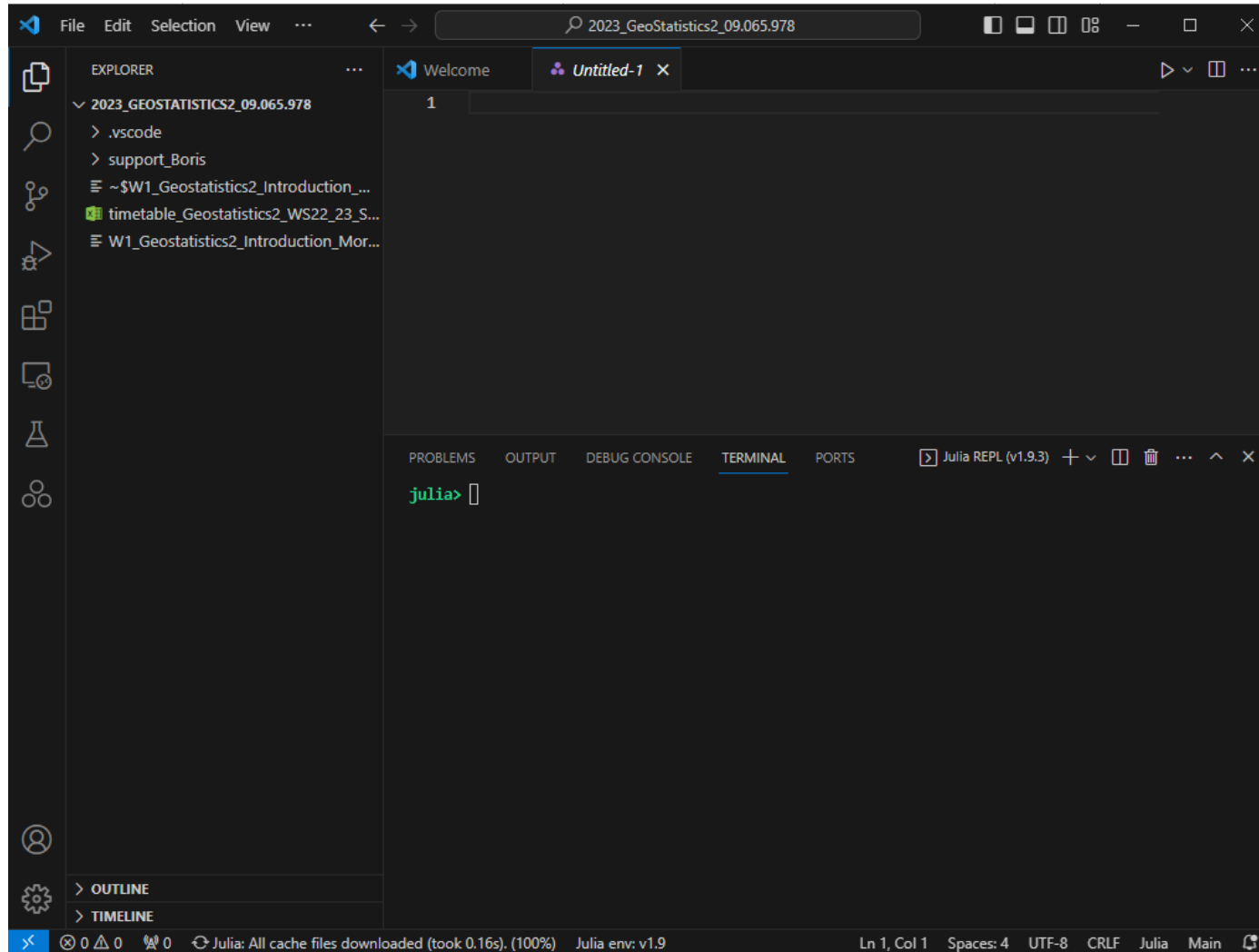
Julia introduction

17-21 02 2025 Heidelberg
Nicolas Riel - nriel@uni-mainz.de

VS Code



Julia REPL (demonstration)

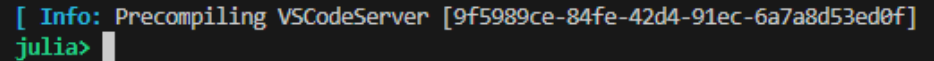


Julia REPL (read-eval-print loop)

Terminal or prompt pasting

- Julia terminal

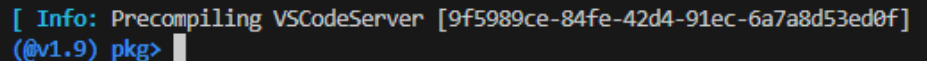
computation space, execute scripts...



```
[ Info: Precompiling VSCodeServer [9f5989ce-84fe-42d4-91ec-6a7a8d53ed0f]  
julia> |
```

-] package manager

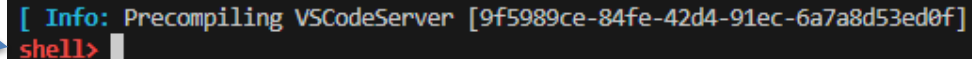
add/update packages



```
[ Info: Precompiling VSCodeServer [9f5989ce-84fe-42d4-91ec-6a7a8d53ed0f]  
(@v1.9) pkg> |
```

- ; shell

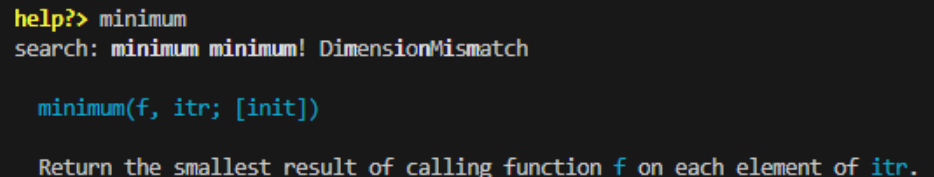
Changing directories



```
[ Info: Precompiling VSCodeServer [9f5989ce-84fe-42d4-91ec-6a7a8d53ed0f]  
shell> |
```

- ? help

provide help with functions



```
help?> minimum  
search: minimum minimum! DimensionMismatch  
  
minimum(f, itr; [init])  
  
Return the smallest result of calling function f on each element of itr.
```

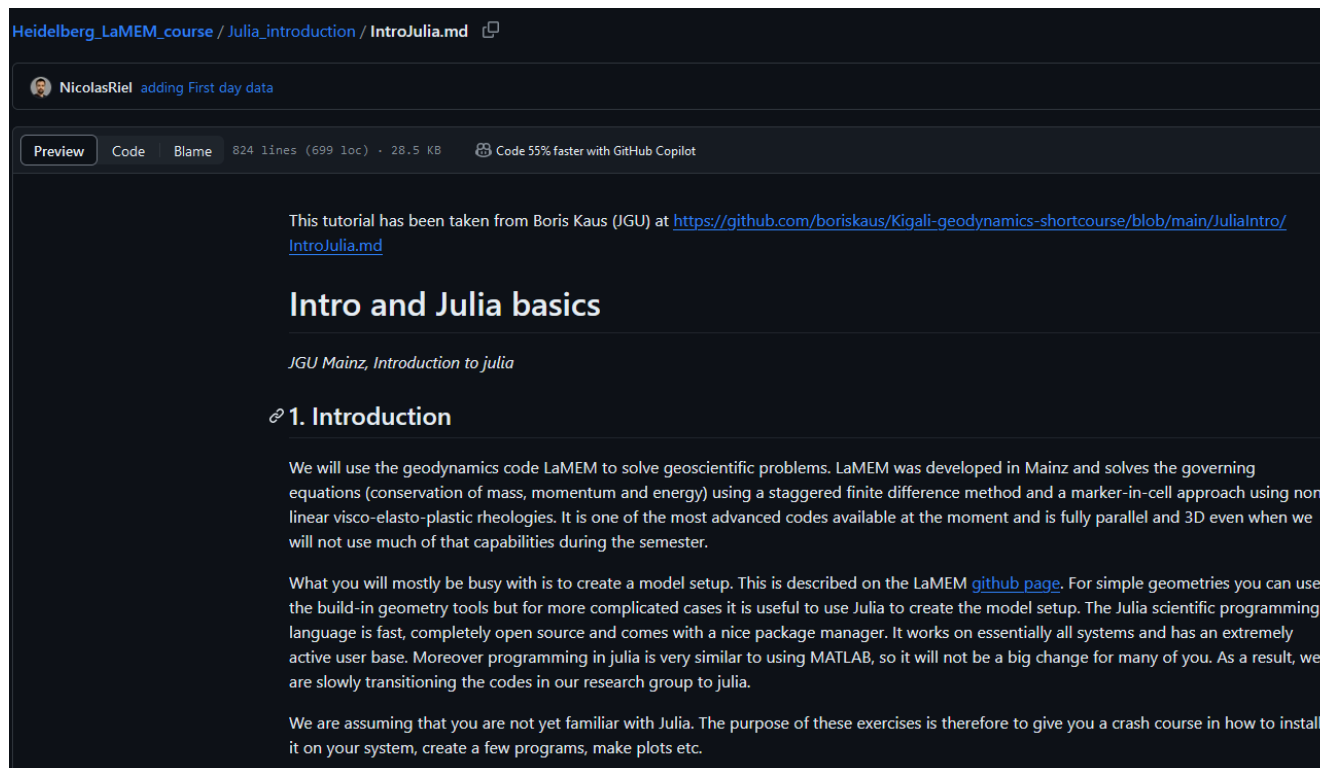
- Backspace

back to julia terminal

Julia introduction

- Access the following link and complete the Julia introduction

https://github.com/NicolasRiel/Heidelberg_LaMEM_course/blob/main/Julia_introduction/IntroJulia.md



The screenshot shows a GitHub repository page for 'Heidelberg_LaMEM_course' with the file 'IntroJulia.md' selected. The page header indicates the file was added by 'NicolasRiel'. Below the header, there are tabs for 'Preview', 'Code', and 'Blame'. The 'Preview' tab is active, showing the content of the file. The content starts with a reference to Boris Kaus (JGU) and a link to the original source. It then has a section titled 'Intro and Julia basics' followed by a subtitle 'JGU Mainz, Introduction to julia'. The main section is titled '1. Introduction' and contains two paragraphs of text explaining the purpose of the tutorial and the use of the LaMEM code.

Heidelberg_LaMEM_course / Julia_introduction / IntroJulia.md

NicolasRiel adding First day data

Preview Code Blame 824 lines (699 loc) · 28.5 KB Code 55% faster with GitHub Copilot

This tutorial has been taken from Boris Kaus (JGU) at <https://github.com/boriskaus/Kigali-geodynamics-shortcourse/blob/main/JuliaIntro/IntroJulia.md>

Intro and Julia basics

JGU Mainz, Introduction to julia

1. Introduction

We will use the geodynamics code LaMEM to solve geoscientific problems. LaMEM was developed in Mainz and solves the governing equations (conservation of mass, momentum and energy) using a staggered finite difference method and a marker-in-cell approach using non-linear visco-elasto-plastic rheologies. It is one of the most advanced codes available at the moment and is fully parallel and 3D even when we will not use much of that capabilities during the semester.

What you will mostly be busy with is to create a model setup. This is described on the LaMEM [github page](#). For simple geometries you can use the build-in geometry tools but for more complicated cases it is useful to use Julia to create the model setup. The Julia scientific programming language is fast, completely open source and comes with a nice package manager. It works on essentially all systems and has an extremely active user base. Moreover programming in julia is very similar to using MATLAB, so it will not be a big change for many of you. As a result, we are slowly transitioning the codes in our research group to julia.

We are assuming that you are not yet familiar with Julia. The purpose of these exercises is therefore to give you a crash course in how to install it on your system, create a few programs, make plots etc.

“for” loops

- Open new Julia file and reproduce the following lines of codes

```
1  # example of for loops
2
3  n_iteration = 10
4  total      = 0.0
5
6  for i=1:n_iteration
7      total = total + 1
8  end
9
10 print("total = $total\n")
11
12 # \n -> line break
13 # \t -> tabulation
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
julia> total      = 0.0
0.0
```

```
julia>
```

```
julia> for i=1:n_iteration
        total = total + 1
        end
```

```
julia>
```

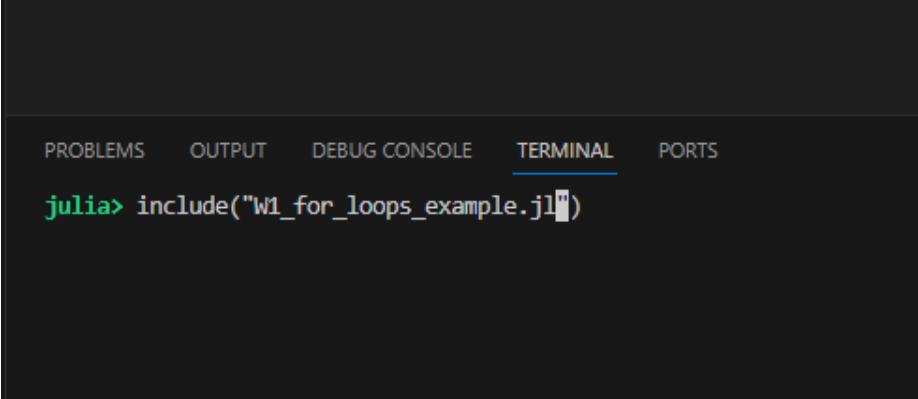
```
julia> print("total = $total\n")
total = 10.0
```

```
julia>
```

- Paste it in the terminal

“for” loops

- CTRL + L to clear the terminal
- Save the file as “meaningful_name.jl”
- Execute the scripts using “include”



A screenshot of a Julia REPL terminal window. The window has a dark background and a light-colored text area. At the top, there are tabs for "PROBLEMS", "OUTPUT", "DEBUG CONSOLE", "TERMINAL", and "PORTS". The "TERMINAL" tab is selected and highlighted with a blue underline. Below the tabs, the text "julia> include("w1_for_loops_example.jl")" is displayed in a green monospace font. A cursor is visible at the end of the line.

```
julia> include("w1_for_loops_example.jl")
```

First bug?

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
julia> include("W1_for_loops_example.jl")
Warning: Assignment to `total` in soft scope is ambiguous because a global variable by the same name exists: `total` will be treated as a new local. Disambiguate by using `local total` to suppress this warning or `global total` to assign to the existing global variable.
@ E:\Nextcloud\RESEARCH\USB_DATA\7_teaching\2023_Mainz\2023_GeoStatistics2_09.065.978\W1_for_loops_example.jl:7
ERROR: UndefVarError: `total` not defined
Stacktrace:
 [1] top-level scope
      @ E:\Nextcloud\RESEARCH\USB_DATA\7_teaching\2023_Mainz\2023_GeoStatistics2_09.065.978\W1_for_loops_example.jl:7
julia> 
```

- Here Julia does not know the “scope” of the variable `total`.
- Scope can be seen as the region of a code where the variable is visible/exists
- The main file here can be thought as an “open space”

→ The right way to solve the issue is to work in a “closed space” i.e. using functions!

```
1  # example of for loops
2
3  n_iteration = 10
4  total      = 0.0
5
6  for i=1:n_iteration
7      total = total + 1
8  end
9
10 print("total = $total\n")
11
12 # \n -> line break
13 # \t -> tabulation
```

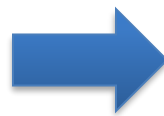

Functions

- Take one or multiple entries as arguments
- Return one of multiple variables

```
terminal  Help
Welcome  W1_demonstration.jl  W1_for_loops_example_function.jl X
W1_for_loops_example_function.jl > ...
1  # example of for loops
2
3
4  function sum_iteration(n_iteration)
5      total = 0.0
6
7      for i=1:n_iteration
8          total = total + 1
9      end
10         total = total + 1
11     return total
12 end
13
14 total = sum_iteration(n_iteration)
15
16
17 print("total = $total\n")
18
19 # \n -> line break
20 # \t -> tabulation
```

```
julia> include("W1_for_loops_example_function.jl")
total = 10.0
```

```
julia> █
```



Solve the “scope” problem

Function file

- When developing a code is it useful to put the functions in other files to keep the main file readable
- Create a “functions_file.jl” and copy/paste the For loop example in it
- Add comment before the function definition as

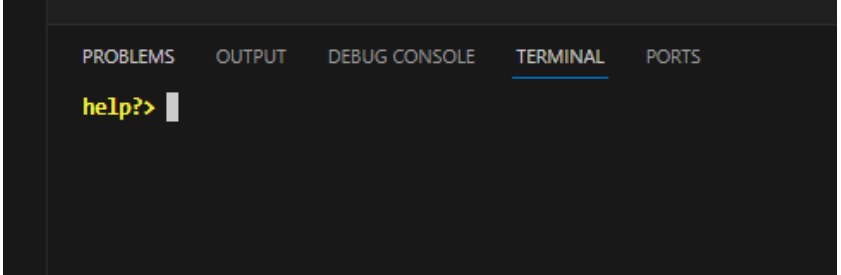
```
1  """
2  |    sum_iteration(n_iteration)
3  |    where n_iteration is an integer
4  |    return sum(n_iteration)
5  |    """
```

```
W1_functions_file.jl > ...
1  """
2  |    test function sum_iteration(n_iteration)
3  |    where n_iteration is an integer
4  |    return sum(n_iteration)
5  |    """
6  function sum_iteration(n_iteration)
7      total = 0.0
8
9      for i=1:n_iteration
10         total = total + 1
11     end
12
13     return total
14 end
15
```

1. The “""" mark the start and ending of the comment
2. The first lines should be the function first lines copied and pasted
3. Subsequent lines describe the input/output and role of the function

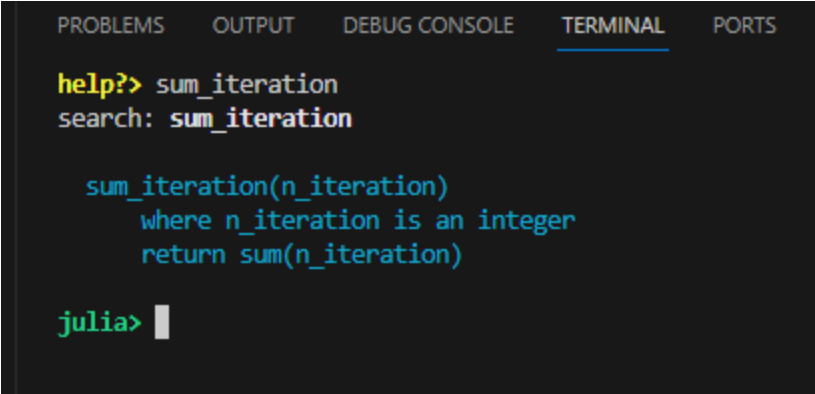
Forgot what the function(s) does?

- Open the help in the REPL (terminal) with ?



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
help?> 
```

- Type `sum_iteration` and enter (or `sum_i` +DOUBLE TAB)



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
help?> sum_iteration
search: sum_iteration

    sum_iteration(n_iteration)
      where n_iteration is an integer
      return sum(n_iteration)

julia> 
```



This displays the comment of your function. Very useful:

- When your code becomes big (many files)
- When you did not use your code for a while
- When you share your code!

“while” loops

- Generalization of the for loop: loops until the looping condition is broken
- Reproduce the following code in the functions_file.jl (after sum_iteration function)
- In the REPL include “W1_functions_file.jl” and define M and n_iteration then call the function

```
julia> include("W1_functions_file.jl")
exercise_3


julia> n_iteration = 10
10

julia> M = 50
50

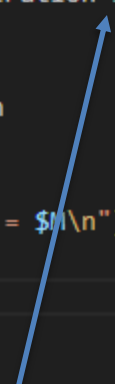
julia> M = while
while      while_loop
julia> M = while_loop(n_iteration,M)
```

Or using default values

```
julia> M = while_loop()
```



```
23  """
24  while_loop(n_iteration,M)
25
26      where n_iteration is an integer, M is an integer
27      return M + sum( (n_iteration-1)*10 )
28  """
29  function while_loop(n_iteration=10,M=50)
30
31      i=1
32      while i < n_iteration
33          M += 10
34          i -= 1
35          print("i = $i, M = $M\n")
36      end
37
38      return M
39  end
```




Note that default values can be defined as such:

Will this program stop or run forever?

“if elseif else” statement

- Check the state of a condition (true or false) and perform different computation depending on the case
- Reproduce code in “functions_file.jl”
- And test it
(re-include “W1_functions_file.jl”)

```
21
22  M = check_M(M)
23
24  print("check_M(M) = $M\n")
25
```



```
37
38  """
39  |   check_M(M)
40  |   where...
41  |
42  |   """
43  function check_M(M)
44  |
45  |   if M < 10
46  |   |   M += 25
47  |   |   elseif M == 10
48  |   |   |   M = M^2
49  |   |   else
50  |   |   |   M = 0
51  |   |   end
52  |
53  |   return M
54  end
```

Arrays (Vector)

- There is several different ways to declare a Vector in Julia for instance:

```
julia> array = [0,1,2,3,4]
5-element Vector{Int64}:
 0
 1
 2
 3
 4
```



This creates and fill the Vector with given values

```
julia> zeros(5)
5-element Vector{Float64}:
 0.0
 0.0
 0.0
 0.0
 0.0
```



This creates a Vector of length 5 and fill it with zeros

```
julia> zeros{Int64}(5)
5-element Vector{Int64}:
 0
 0
 0
 0
 0
```



The type can be specified

Note: you can also use:
`ones()`

Access/Modify Arrays (Vector)

- Access

```
julia> array = [0,1,2,3,4]
5-element Vector{Int64}:
 0
 1
 2
 3
 4
```

- Modify

```
julia> array[1] = 4
4
```

```
julia> array[1]
0
```

By position

```
julia> array[1:4]
4-element Vector{Int64}:
 0
 1
 2
 3
```

Using range

```
julia> array[[1,2,3,4]]
4-element Vector{Int64}:
 0
 1
 2
 3
```

Using vector of positions

```
julia> array[1:4] .= 4
4-element view(::Vector{Int64}, 1:4) with eltype Int64:
 4
 4
 4
 4
```

Don't forget the dot
(pointwise)

```
julia> array[[1,2,3,4]] .= 4
4-element view(::Vector{Int64}, [1, 2, 3, 4]) with eltype Int64:
 4
 4
 4
 4
```

Arrays (Matrix)

- There is several different ways to declare an array in Julia for instance:

```
julia> array2D = [0 1 2 3 4; 5 6 7 8 9]
2x5 Matrix{Int64}:
 0  1  2  3  4
 5  6  7  8  9
```



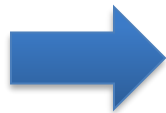
This creates and fill the matrix with given values

```
julia> zeros(2,5)
2x5 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
```



This creates a Matrix of 2 lines and 5 columns filled with 0.0

```
julia> zeros(Float64,2,5)
2x5 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
```



As before the type can be specified

Access/Modify Arrays (Matrix)

- Access

```
julia> array2D = [0 1 2 3 4; 5 6 7 8 9]
2x5 Matrix{Int64}:
 0  1  2  3  4
 5  6  7  8  9
```

- Modify

2D arrays are row major

```
julia> array2D[2,4]
8
```

By position

```
julia> array2D[2,4] = 4
4
```

```
julia> array2D[1,2:4]
3-element Vector{Int64}:
 1
 2
 3
```

Using range

```
julia> array2D[1,2:4] .= 4
3-element view(::Matrix{Int64}, 1, 2:4) with eltype Int64:
 4
 4
 4
```

Don't forget the dot
(pointwise)

```
julia> array2D[1,[2,3,4]]
3-element Vector{Int64}:
 1
 2
 3
```

Using vector of positions

```
julia> array2D[1,[2,3,4]] .= 4
3-element view(::Matrix{Int64}, 1, [2, 3, 4]) with eltype Int64:
 4
 4
 4
```

Why does type matters?

- There is three main types in Julia: Int64, Float64 and String
- To find the type of any variable use “typeof”
- Using some of the previous array definition, this gives:

```
julia> array2D = [0 1 2 3 4; 5 6 7 8 9]
2x5 Matrix{Int64}:
 0  1  2  3  4
 5  6  7  8  9

julia> typeof(array2D)
Matrix{Int64} (alias for Array{Int64, 2})
```

```
julia> array = [0 1 2 3 4 5]
1x6 Matrix{Int64}:
 0  1  2  3  4  5

julia> typeof(array)
Matrix{Int64} (alias for Array{Int64, 2})
```

```
julia> array = zeros(Float64,2,5)
2x5 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0

julia> typeof(array)
Matrix{Float64} (alias for Array{Float64, 2})
```

```
julia> array = [0, 1, 2, 3, 4, 5]
6-element Vector{Int64}:
 0
 1
 2
 3
 4
 5

julia> typeof(array)
Vector{Int64} (alias for Array{Int64, 1})
```

```
julia> i = 1
1

julia> typeof(i)
Int64
```

Arrays with push!

- Another way to create array is to declare it empty and push elements to it

```
julia> r = []  
Any[]  
  
julia> push!(r,1.0)  
1-element Vector{Any}:  
 1.0  
  
julia> push!(r,2.0)  
2-element Vector{Any}:  
 1.0  
 2.0  
  
julia> push!(r,3.0)  
3-element Vector{Any}:  
 1.0  
 2.0  
 3.0
```



Type Any
(can contain anything)

The type can be enforced too:

```
julia> r = Int64[]  
Int64[]  
  
julia> push!(r,1.0)  
1-element Vector{Int64}:  
 1  
  
julia> push!(r,2.0)  
2-element Vector{Int64}:  
 1  
 2  
  
julia> push!(r,3)  
3-element Vector{Int64}:  
 1  
 2  
 3
```



Note that even if a Float is
pushed, an integer is stored

This can also be done for
Vectors (and Matrix)

```
julia> r = []  
Any[]  
  
julia> push!(r,[1,2,3])  
1-element Vector{Any}:  
 [1, 2, 3]  
  
julia> push!(r,[1,2,3])  
2-element Vector{Any}:  
 [1, 2, 3]  
 [1, 2, 3]
```



And converted to matrix using
mapreduce function

```
julia> tmat = mapreduce(permutedims, vcat, r)  
2x3 Matrix{Int64}:  
 1 2 3  
 1 2 3
```

Working with arrays

- Creates 2 arrays as follow:

```
julia> array1 = [4,3,2,1]
4-element Vector{Int64}:
 4
 3
 2
 1
```



Here no comma is used and the Vector is created using Integers

```
julia> array2 = [4,3,2,1.0]
4-element Vector{Float64}:
 4.0
 3.0
 2.0
 1.0
```



Here a single comma is used and the Vector is created using Float

- Go through the values of the array using a “for” loop

```
1 for i=1:length(array1)
2     ⚡ val = array1[i]
3
4     ... # display value of array at position i, and the value of array at position val
5     ... print("val: $(val) array1[val]: $(array1[val])\n")
6 end
```

What happens if we use array2 instead?

Working with arrays

- Position in an array has to be access with integers (positional index)

```
julia> for i=1:length(array2)
    val = array2[i]

    # display value of array at position i, and the value of array at position val
    print("val: $(val) array1[val]: $(array1[val])\n")
end
ERROR: ArgumentError: invalid index: 4.0 of type Float64
Stacktrace:
 [1] to_index(i::Float64)
   @ Base .\indices.jl:300
 [2] to_index(A::Vector{Int64}, i::Float64)
   @ Base .\indices.jl:277
 [3] _to_indices1(A::Vector{Int64}, inds::Tuple{Base.OneTo{Int64}}, I1::Float64)
   @ Base .\indices.jl:359
 [4] to_indices
   @ .\indices.jl:354 [inlined]
 [5] to_indices
   @ .\indices.jl:345 [inlined]
 [6] getindex(A::Vector{Int64}, I::Float64)
   @ Base .\abstractarray.jl:1296
 [7] top-level scope
   @ .\REPL[60]:5
```

- Luckily Floats (including Vector{Float64}) can be converted to integer as:

```
julia> array2 = Int64.(array2)
4-element Vector{Int64}:
 4
 3
 2
 1
```

First Julia plot

- First add a plotting package “Plots”

] add Plots

“]” opens the package manager

```
(@v1.9) pkg> add Plots  
Resolving package versions...
```

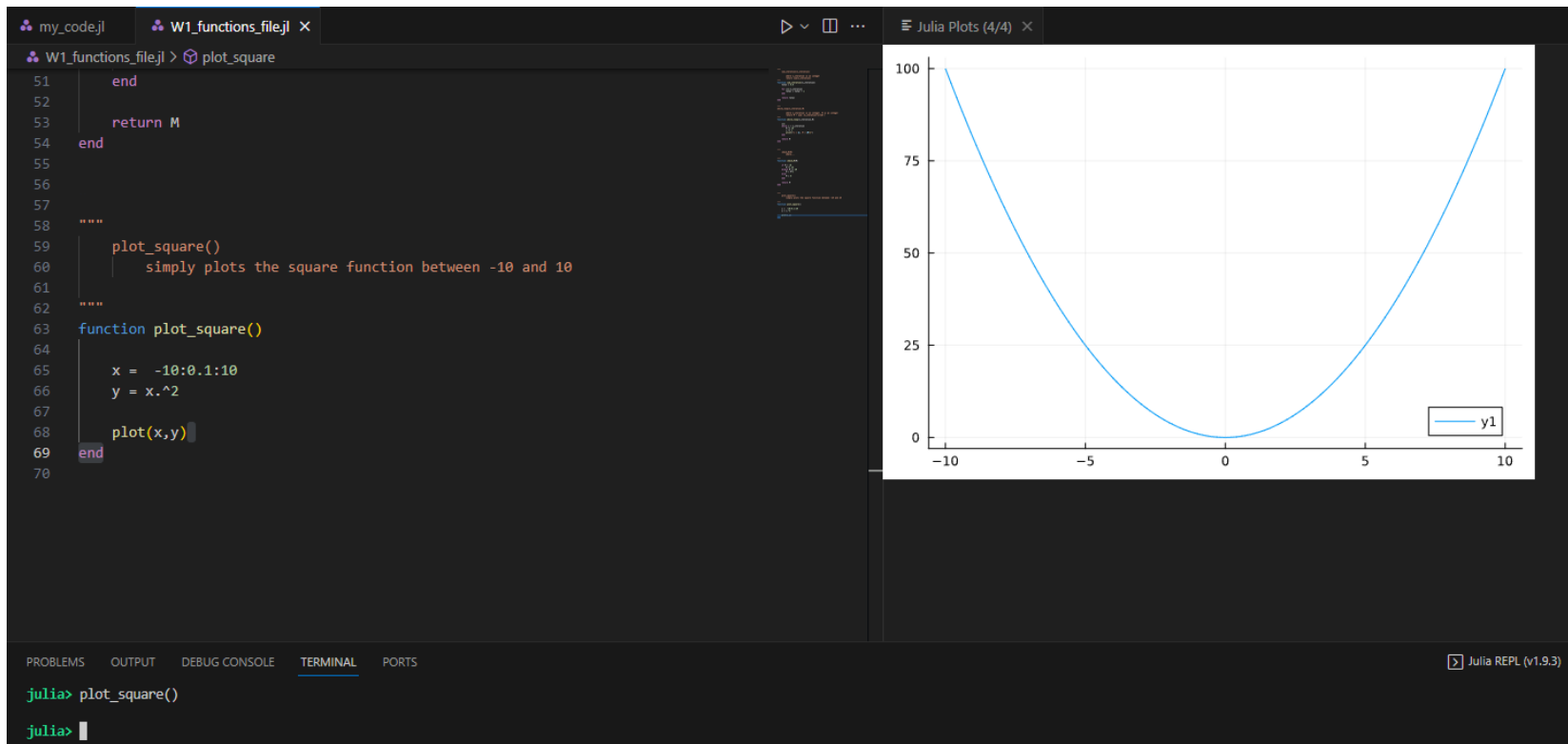
It can take a few minutes as other default packages are updated

- Create a plot_square function (still in functions_file.jl)

```
57  
58  
59     plot_square()  
60     |    simply plots the square function between -10 and 10  
61  
62     ""  
63     function plot_square()  
64  
65         x = -10:0.1:10  
66         y = x.^2  
67  
68         plot(x,y)  
69     end  
70
```

First Julia plot

- Execute “my_code.jl”
- In the REPL (terminal) type: `plot_square()`



→ The figures are displayed within VS code in a dedicated window

Add title and label

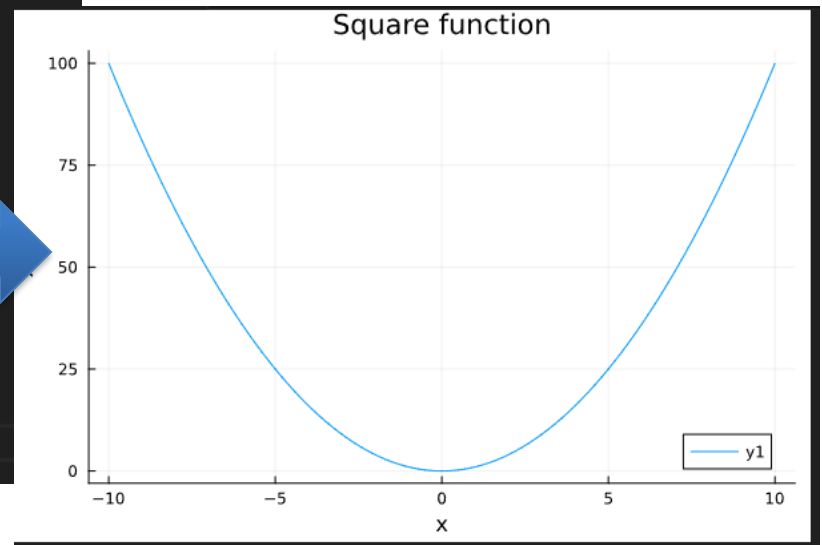
- Modify the function as

```
"""
plot_square()
    simply plots the square function between -10 and 10
"""

function plot_square()

    x = -10:0.1:10
    y = x.^2

    plot(x,y)
    title!("Square function")
    xlabel!("x")
    ylabel!("x^2")
end
```



→ Here the "!" means that the title, x- and y-labels will be added to previously declared plot.

Adding another curve to existing plot

- Modify the function as

```
"""
    plot_square()
    simply plots the square function between -10 and 10
"""

function plot_square()

    x = -10:0.1:10
    y = x.^2
    yp = x.^3

    plot(x,y)
    plot(x,yp)
    title!("Square function")
    xlabel!("x")
    ylabel!("x^2")
end
```



What happens here?

Adding another curve to existing plot

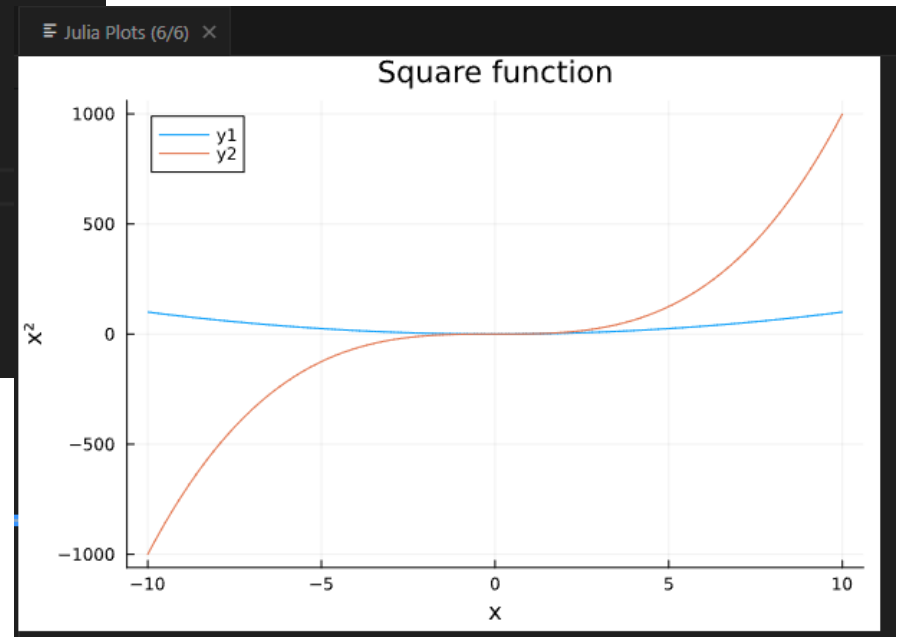
- The “!” is also used to add curves to an existing plot

```
"""
    plot_square()
    simply plots the square function between -10 and 10
"""

function plot_square()

    x = -10:0.1:10
    y = x.^2
    yp = x.^3

    plot(x,y)
    plot!(x,yp)
    title!("Square function")
    xlabel!("x")
    ylabel!("x^2")
end
```



Exercise 1a

Assume we have an initial amount of money on the bank ($M=1000$) and each year we save some money ($dM=100$)

1. What will the amount of money be after 25 years? Create Julia function for this.
2. Plot the total amount of money (M) versus time in years.

Example of solution

```
exercice_1(M, dM, n_years)
    Assumes we have an initial amount of money on the bank (M=1000)
    and each year we save some money (dM=100)
    M is starting_money
    dM is yearly_savings
    n_years is the number fo years of savings
```

```
function exercice_1(M, dM, n_years)
```

```
    Mtotal = zeros(n_years)
```

```
    for i=1:n_years
```

```
        M = M + dM;
```

```
        Mtotal[i] = M
```

```
    end
```

```
    plot(Mtotal)
```

```
    title!("Money trend in the bank")
```

```
    xlabel!("time (years)")
```

```
    ylabel!("money \$")
```

```
end
```

```
help?> exercice_1
search: exercice_1
```

```
exercice_1(M, dM, n_years)
```

```
    Assumes we have an initial amount of money on the bank (M=1000)
    and each year we save some money (dM=100)
```

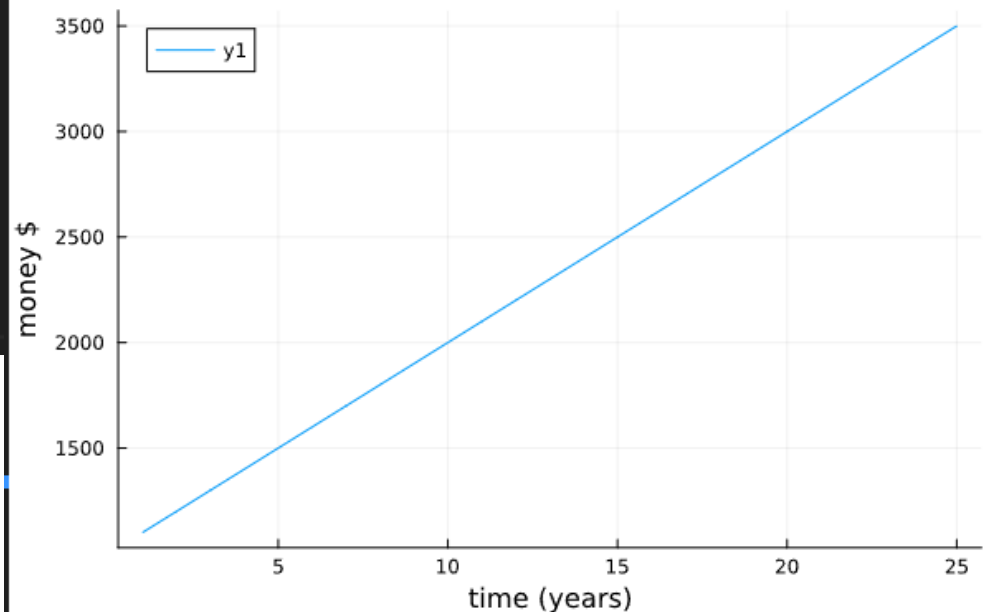
```
    M is starting_money
```

```
    dM is yearly_savings
```

```
    n_years is the number fo years of savings
```

Julia Plots (7/7) X

Money trend in the bank



```
julia> include("w1_functions_file.jl")
exercice_1
```

```
julia> exercice_1(1000, 100, 25)
```

Exercise 1b

Most banks give an interest on the money you have and pay that at the end of the year. Let's assume that the interest rate is 10%.

1. Duplicate the previous function and call it `exercice_1b`.
2. Modify the function to account for interest rate

Example of solution

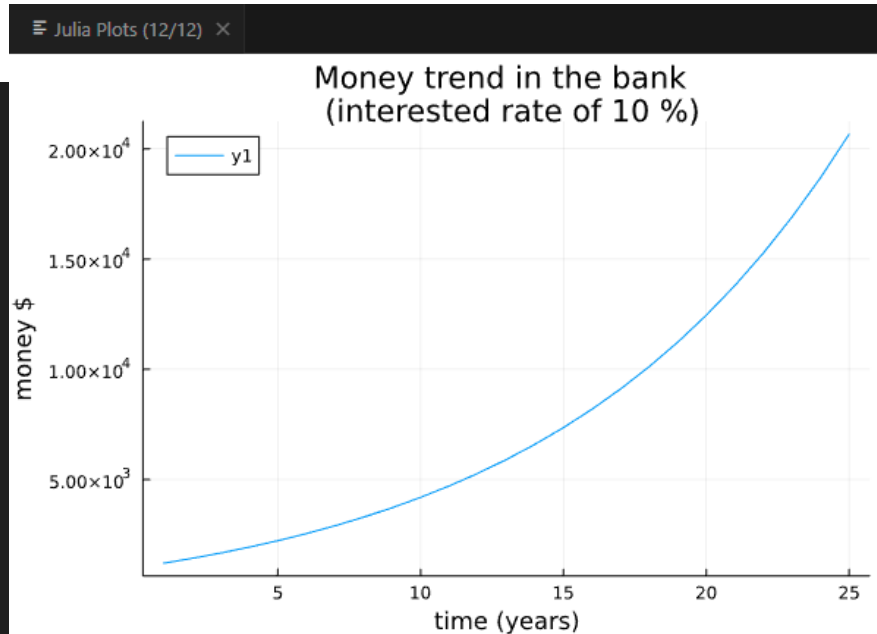
```
"""
exercice_1b(M, dM, n_years, interest_rate)
    Assumes we have an initial amount of money on the bank (M=1000)
    and each year we save some money (dM=100)
    M is starting_money
    dM is yearly_savings
    n_years is the number fo years of savings
    interest_rate is the interest rate of the bank in in %
"""

function exercice_1b(M, dM, n_years, interest_rate)

    Mtotal = zeros(n_years)

    for i=1:n_years
        M = M + M*interest_rate/100
        M = M + dM;
        Mtotal[i] = M
    end

    plot(Mtotal)
    title!("Money trend in the bank \n (interested rate of $interest_rate %)")
    xlabel!("time (years)")
    ylabel!("money \$")
end
```



```
julia> include("w1_functions_file.jl")
exercice_1b

julia> exercice_1b(1000, 100,25, 10)
```

Exercise 1c

1. Explore plotting options (Plots.jl)

use online doc:

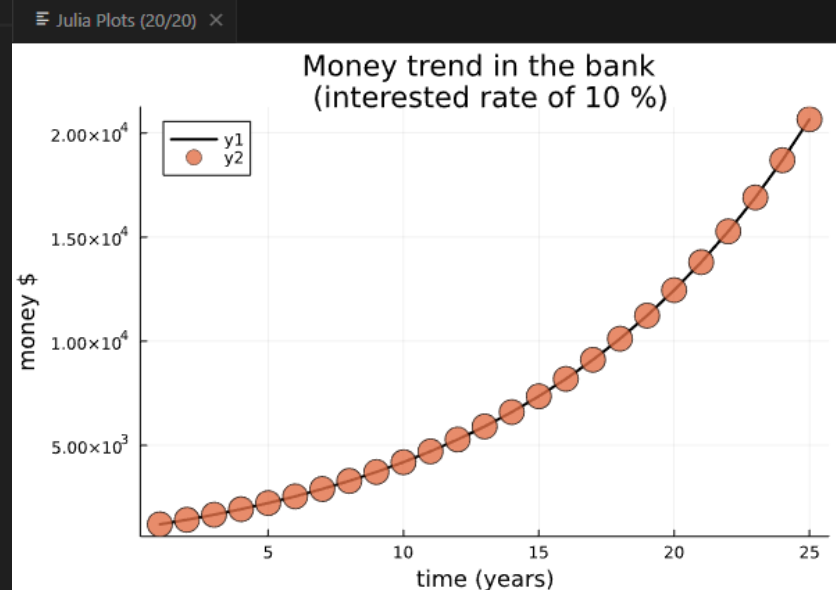
<https://docs.juliaplots.org/latest/tutorial/>

1. Improve “exercise_1b()” -> “exercise_1c()”

1. Change colors of the curve
2. Display points on top of the curve function (scatter)

Example of solution

```
129 """
130 exercise_1c(M, dM, n_years, interest_rate)
131     Assumes we have an initial amount of money on the bank (M=1000)
132     and each year we save some money (dM=100)
133     M is starting_money
134     dM is yearly_savings
135     n_years is the number fo years of savings
136     interest_rate is the interest rate of the bank in in %
137 """
138 function exercise_1c(M, dM, n_years, interest_rate)
139
140     Mtotal = zeros(n_years)
141
142     for i=1:n_years
143         M = M + M*interest_rate/100
144         M = M + dM;
145         Mtotal[i] = M
146     end
147
148     plot(Mtotal, lc=:black, lw=2)
149     scatter!(Mtotal, lc=:red, ms=10, ma=0.8)
150     title!("Money trend in the bank \n (interested rate of $interest_rate %)")
151     xlabel!("time (years)")
152     ylabel!("money \$")
153 end
```



Save the plot to a file as:

```
julia> savefig("save_figure_test.png")
```

(The file is saved in the working directory)

To go further

- Exercise 1d: fit an equation through the datapoints

Copy function “exercise_1c()” to “exercise_1d()”

Use CurvFit.jl to fit the points
(julia>] add CurveFit)

Use online doc and provided examples

<https://juliapackages.com/p/curvefit>

Try different polynomial exponent and plot the results
using Plots.jl

```
using CurveFit

x = 0.0:0.02:2.0
y0 = @. 1 + x + x*x + randn()/10
fit = curve_fit(Polynomial, x, y0, 2)
y0b = fit.(x)
```