Intro

Contents

Note

Disclaimer: While writing 1st draft of the book I was planning to entitle it in a tongue-in-cheek way, *Dating Julia as a macroeconomist*. Back then I was not aware of gender neutrality of the `Julia` language explicitly stated in the community standards. As a supporter of the gender inclusivity I changed the title but at the same time I kept this information to amplify this message even more.

The purpose of this notebook is to capture, organize, and categorize a knowledge repository of my `Julia` workflow. At this moment, I write it mainly for myself to be able to get back to `Julia` after short breaks. That being said, it might be of interest to other economists wanting to learn `Julia` or to improve their own workflow. People who want to start their journey with the language absolutely from scratch should start with a great intro by Sargent and Stachursky, which can be found here, or Julia Express by Bogumił Kamiński.

Numeric types

First, let's focus on different and most basic (and useful for daily use) numerical types of objects in `Julia`.

Type	Desc.	Smallest number	Largest number
`Int8`	signed integers	-2^7	$2^7 - 1$
`UInt8`	unsigned integers (*read:* natural numbers)	0	2^8
`Int16`	signed integers	-2^{15}	$2^{15} - 1$
`UInt16`	unsigned integers (*read:* natural numbers)	0	2^{16}
`Int64`	signed integers	-2^{63}	$2^{63} - 1$
`UInt64`	unsigned integers (*read:* natural numbers)	0	2^{64}

There is also an alias `Int` for integers with the default number of bits for your architecture (32 or 64).

```
x = 1
```

```
1
```

For quiet execution of line `x = 1`, we should add `;` at the end of the line. This way the assignment will be made without printing the result.

```
typeof(x)
```

```
Int64
```

We can explicitly force `Julia` to set the precision of our integers:

Note

In `Julia` we can use special unicode characters for naming objects, like `α` or even `αₖʲ`. In VS Code, to get `α` you have to type `\alpha` and press `<tab>`. `αₖʲ` can be obtained by typing combination `\alpha`, `<tab>`, `\_k`, `<tab>`, `\^j`, `<tab>`.

```
α = Int8(2)
```

```
2
```

```
Int8(2^7-1)
```

```
127
```

However, we have to remember about the ranges of used types:

```
Int8(2^8) #Error message
```

```
InexactError: trunc(Int8, 256)

Stacktrace:
 [1] throw_inexacterror(f::Symbol, #unused#::Type{Int8}, val::Int64)
   @ Core ./boot.jl:602
 [2] checked_trunc_sint
   @ ./boot.jl:624 [inlined]
 [3] toInt8
   @ ./boot.jl:639 [inlined]
 [4] Int8(x::Int64)
   @ Core ./boot.jl:749
 [5] top-level scope
   @ In[5]:1
 [6] eval
   @ ./boot.jl:360 [inlined]
 [7] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
   @ Base ./loading.jl:1094
```

Unsigned integers are not super useful due to their default hexadecimal notation:

```
@show UInt(1018);
```

```
UInt(1018) = 0x00000000000003fa
```

Another type of numbers in `Julia` are floating-point values

Type	Number of bits
`Float16`	16
`Float32`	32
`Float64`	64

Similarly to integers, `float` refers to floating-point values with the default number of bits for your architecture (32 or 64).

Notice the difference in letter capitalization: `float` vs. `Int`!

```
w = .2
```

```
0.2
```

```
typeof(w)
```

```
Float64
```

Basic arithmetic operations

On the surface, arithmetic operations (`+`, `-`, `*`, `/`) in `Julia` are very similar to the ones known from such languages as `Matlab`:

```
x = 1
y = 2
x + y
```

```
3
```

For multiplication we do not have to use `*` for unambiguous cases. `2*x` and `2x` wille give the same results. However, I do not recommend it because in my opinion it makes the code a bit less legible.

For presenting results of the operation macro `@show` can be quite useful:

```
x = 1
y = 2
@show x + y + .2
```

```
x + y + 0.2 = 3.2
```

```
3.2
```

You can make several assignments in one line using separator `,`. Then, instead of three lines:

```
x = 12.3;
y = 1;
🤩 = -10;
```

you can just use one line:

```
x, y, 🤩 = 12.3, 1, -10
```

```
(12.3, 1, -10)
```

Yes, you can use emojis for naming objects in `Julia`. Nonetheless, you have to make sure that emojis are installed on the computer. This is rather the case for your own laptop but it is not so obious for UN*X distributions used on HPC clusters, which you may use at some point. Moreover, there is a documented problem with using emojis in Julia for VS Code.

```
@show 🤩
@show x
@show y
@show y, x
```

```
🤩 = -10
x = 12.3
y = 1
(y, x) = (1, 12.3)
```

```
(1, 12.3)
```

Operations (`+`, `-`, `*`, `/`, `\`, `÷`, `%`, `^`) where one object is on both sides of the assignment operator (*e.g.*, `x=x+1`) can be shortened:

```
@show x;
@show x += 1;
@show x *= 2;
@show x ^= 2;
```

```
x = 12.3
x += 1 = 13.3
x *= 2 = 26.6
x ^= 2 = 707.5600000000001
```

If we combine two different numeric types, `Julia` will try to coerce the less general object to more general one:

```
x = Int(42)
y = float(.964)
@show z = x+y

typeof(z)
```

```
z = x + y = 42.964
```

```
Float64
```

You can use also `Lisp`-like syntax (which we find in `R` as well. Sometimes this can be very useful, and other times, it can be quite scary):

```
+(x, y)
```

```
42.964
```

```
@show  ^(*( 3, +(1, 2) ), 5)
```

```
(3 * (1 + 2)) ^ 5 = 59049
```

```
59049
```

There are several ways to divide numbers. The most natural one is by using `/`:

```
@show 1/2
```

```
1 / 2 = 0.5
```

```
0.5
```

If for some reason, we want to use common fractions instead of decimals, then we have to use `//` instead:

```
1//2
```

```
1//2
```

Within common fractions, we can perform arithmetic operations:

```
@show 1//2+1//3;
```

```
1 // 2 + 1 // 3 = 5//6
```

```
@show (1//2)^10;
```

```
(1 // 2) ^ 10 = 1//1024
```

But due to coercion in operations over different types, we might lose it:

```
@show 1//2 + .25;
@show 1//2 + π;
```

```
1 // 2 + 0.25 = 0.75
1 // 2 + π = 3.641592653589793
```

We perform integer division by using `div` or `÷` (to get this symbol type `\div` and `<tab>`):

```
@show div(5,3);
@show   ÷(5,3);
@show     5÷3;
```

```
div(5, 3) = 1
5 ÷ 3 = 1
5 ÷ 3 = 1
```

We get remainder by using `rem` or `%`:

```julia
@show rem(5,3);
@show   %(5,3);
@show     5%3;
```

```
rem(5, 3) = 2
5 % 3 = 2
5 % 3 = 2
```

Data structures

Arrays

Arrays are one of the most important data structures for economists. They are equivalent of vectors and matrices in `Matlab` or `R`. Arrays are multi-dimensional objects containing numbers (or different objects).

A vector is a one-dimensional array. To get a row vector you use the following convention:

```julia
A = [1 2 3]
```

```
1×3 Matrix{Int64}:
 1  2  3
```

You get column vectors by using `;` as the separator of elements:

```julia
B = [1; 2; 3]
```

```
3-element Vector{Int64}:
 1
 2
 3
```

All elements of arrays must be of the same type. In case of type incompatibility across elements, coercion is performed:

```julia
@show c = [1 π 2.3]

@show typeof(c) #It's Float64 despite the first element being an integer.
```

```
c = [1 π 2.3] = [1.0 3.141592653589793 2.3]
typeof(c) = Matrix{Float64}
```

```
Matrix{Float64} (alias for Array{Float64, 2})
```

```julia
X = [1 2 3;
     3 4 5]
```

```
2×3 Matrix{Int64}:
 1  2  3
 3  4  5
```

It is a good practice to initialize arrays before assigning values to their elements. To create an empty one-dimensional array with `K` elements of type `TYPE` you use `Array{TYPE}(undef, K)`. Below an illustrating example:

```julia
Array{Float64}(undef, 10)
```

```
10-element Vector{Float64}:
 3.5e-323
 8.4e-323
 9.4e-323
 1.1e-322
 1.2e-322
 1.24e-322
 1.43e-322
 1.5e-322
 1.53e-322
 5.0e-324
```

Argument `undef` means that nothing particular is assigned to all elements. Instead, existing values stored previously in the memory are assigned.

To create an array of dimensions $K \times N \times S \times W$ you use `Array{TYPE}(undef, K, N, S, W)`. Below an illustrating example:

```
Array{Int}(undef, 2, 3, 2, 2)
```

```
2×3×2×2 Array{Int64, 4}:
[:, :, 1, 1] =
 4563792624  4563792752  4563815568
 4563792688  4563814960  4563815056

[:, :, 2, 1] =
 4563792784  4563792880  4563815248
 4563792848  4649217888  4527567872

[:, :, 1, 2] =
 4563792976  4563793072  4527566656
 4563793040  4563815440  4646453648

[:, :, 2, 2] =
 4527568096  0  0
          0  0  0
```

To get an array with all elements constant, you use command `fill`:

```
A = fill(π^2, 4, 3)
```

```
4×3 Matrix{Float64}:
 9.8696  9.8696  9.8696
 9.8696  9.8696  9.8696
 9.8696  9.8696  9.8696
 9.8696  9.8696  9.8696
```

To get an object of the same size and data type, function `similar` can be useful. Elements of the new objects will contain information previously allocated in the memory.

```
B = similar(A)
```

```
4×3 Matrix{Float64}:
 2.20884e-314  2.20884e-314  2.20885e-314
 2.20884e-314  2.20885e-314  2.20885e-314
 2.20884e-314  2.20885e-314  2.20885e-314
 2.20884e-314  2.20885e-314  2.20489e-314
```

Arrays with zero elements can be constructed with `zeros`:

```
zeros(3,2)
```

```
3×2 Matrix{Float64}:
 0.0  0.0
 0.0  0.0
 0.0  0.0
```

To get an identity square matrix of dimension K function `I` from package `LinearAlgebra` can be used:

To load package to the memory, you use `using PackageName`. To install packages you use `Pkg` package: `using Pkg;`

```
Pkg.add("PackageName")
```

```
using LinearAlgebra
I(5)
```

```
5×5 Diagonal{Bool, Vector{Bool}}:
 1  ⋅  ⋅  ⋅  ⋅
 ⋅  1  ⋅  ⋅  ⋅
 ⋅  ⋅  1  ⋅  ⋅
 ⋅  ⋅  ⋅  1  ⋅
 ⋅  ⋅  ⋅  ⋅  1
```

`I` generates a matrix with boolean elements (with `true` and `false` values) but it can be used normally with arrays of other numeric types thanks to coercion.

Simple array operations

Except square-bracket notations, accessing arrays in `Julia` is almost one-to-one analogy to `Matlab`:

```
A = [1 2 3 4 5;
     6 7 5 9 12]
```

```
2×5 Matrix{Int64}:
 1  2  3  4   5
 6  7  5  9  12
```

```
size(A)
```

```
(2, 5)
```

```
A[1,:] #1st row
```

```
5-element Vector{Int64}:
 1
 2
 3
 4
 5
```

```
A[:,1] #1st column
```

```
2-element Vector{Int64}:
 1
 6
```

```
A[:,[1,3]] #1st and 3rd column
```

```
2×2 Matrix{Int64}:
 1  3
 6  5
```

```
A[:,3:end] #from 3rd to the last columns
```

```
2×3 Matrix{Int64}:
 3  4   5
 5  9  12
```

```
A[1,3:end] #1st row and from 3rd to the last columns
```

```
3-element Vector{Int64}:
 3
 4
 5
```

Just like with scalars, arithmetic operations on arrays of *the same dimensions* work in an analogous way:

```
A = [1 2;3 4];
B = I(2);
```

```
A + B
```

```
2×2 Matrix{Int64}:
 2  2
 3  5
```

```
A*A
```

```
2×2 Matrix{Int64}:
  7  10
 15  22
```

```
A
```

```
2×2 Matrix{Int64}:
 1  2
 3  4
```

```
A' #transposition
```

```
2×2 adjoint(::Matrix{Int64}) with eltype Int64:
 1  3
 2  4
```

```
A^(-1) #invert matrix
```

```
2×2 Matrix{Float64}:
 -2.0   1.0
  1.5  -0.5
```

```
A^(-1)*A
```

```
2×2 Matrix{Float64}:
 1.0          0.0
 2.22045e-16  1.0
```

```
kron(A,I(3))
```

```
6×6 Matrix{Int64}:
 1  0  0  2  0  0
 0  1  0  0  2  0
 0  0  1  0  0  2
 3  0  0  4  0  0
 0  3  0  0  4  0
 0  0  3  0  0  4
```

Nonetheless, if we try to add an object of incompatible dimensions (including scalars), we will get an error message, just like in an example below:

```
A + 1
```

```
MethodError: no method matching +(::Matrix{Int64}, ::Int64)
For element-wise addition, use broadcasting with dot syntax: array .+ scalar
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...) at operators.jl:560
  +(::T, ::T) where T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128, UInt16,
UInt32, UInt64, UInt8} at int.jl:87
  +(::Rational, ::Integer) at rational.jl:288
  ...

Stacktrace:
 [1] top-level scope
   @ In[26]:1
 [2] eval
   @ ./boot.jl:360 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
   @ Base ./loading.jl:1094
```

This can be perceived as a particularly bizzare and excessively formal thing. That said, at the development stage there were reasons for this. Furthermore, notice that in Mathematics this type of operations, $A + 1$, in most cases is also forbidden. One (extremely inefficient but didactically useful) solution to this problem is to increase the size of the smaller object by using `repeat`. This function constructs an array by repeating the input *array* a given number of times in each out of the provided dimension.

Thus, to compute $A + 1$, we have to increase the size of 1 in such a way that the new object fits the size of A. As we remember the size of A is 2×2. Just to double check, let's use function `size`.

```
size(A)
```

```
(2, 2)
```

The first argument of `repeat` is the input array (note that we need to write the array form `[1]` fort this reason that a scalar object `1` is not accepted by the function), next arguments tell us how many times in each dimension the object should be repeated.

Below you can find `repeat` at play:

```
repeat([1], 3)
```

```
3-element Vector{Int64}:
 1
 1
 1
```

```
repeat([1], 3, 2)
```

```
3×2 Matrix{Int64}:
 1  1
 1  1
 1  1
```

```
repeat([1], 4, 3, 2)
```

```
4×3×2 Array{Int64, 3}:
[:, :, 1] =
 1  1  1
 1  1  1
 1  1  1
 1  1  1

[:, :, 2] =
 1  1  1
 1  1  1
 1  1  1
 1  1  1
```

```
repeat([1; 2], 2, 3) #input argument: column vector
```

```
4×3 Matrix{Int64}:
 1  1  1
 2  2  2
 1  1  1
 2  2  2
```

```
repeat([1 2], 2, 3) #input argument: row vector
```

```
2×6 Matrix{Int64}:
 1  2  1  2  1  2
 1  2  1  2  1  2
```

Consequently, we are in the position to compute $A + 1$:

```
A + repeat([1], 2, 2)
```

```
2×2 Matrix{Int64}:
 2  3
 4  5
```

An addition of two arrays of different sizes can be conducted analogously:

```
A + repeat([0; 1], 1, 2)
```

```
2×2 Matrix{Int64}:
 1  2
 4  5
```

Note

That being said, by no means use this approach in your daily use. A *much more recommended* and **efficient** way to operate on objects of different sizes is to use broadcasting or mapping. I discuss it in more details in one of the further secions.

Assignment operator (=) and arrays: easy for Pythonistas, confusing for Matlabians

What can be quite confusing at first for people with the `Matlab` background is how the assignment operator (`=`) works with arrays in `Julia`. In `Matlab` if we want to create a matrix `V1` that is a copy of another matrix `V0` it is enough to write:

```
V1=V0
```

Henceforth, there are two *separate* matrices in the memory, `V1` and `V0`. Any changes in either of those will not affect the other one. In `Julia` this works in a very different way. Exactly like in `Python`, running `V1=V0` will create a new *pointer* to the memory allocated for `V0`. Both objects, `V1` and `V0`, share the same information. Consequently, any changes in the memory through accessing `V1` will affect the data pointed by `V0`, and vice versa. Below you can find an illustration of this language feature.

```
@show V₀ = [1 2 3];
```

```
V₀ = [1 2 3] = [1 2 3]
```

```
@show V₁ = V₀;
```

```
V₁ = V₀ = [1 2 3]
```

Running `V₁ = V₀` will create a new pointer to the memory associated with `V₀`. Now changes in either of those objects will affect both pointers:

```
@show V₁[1] = 420;
@show V₁;
@show V₀;

@show V₀[3] = 128;
@show V₁;
@show V₀;
```

```
V₁[1] = 420 = 420
V₁ = [420 2 128]
V₀ = [420 2 128]
V₀[3] = 128 = 128
V₁ = [420 2 128]
V₀ = [420 2 128]
```

Note to myself: explain it using better words.

This feature of `Julia` can be especially confusing in iterative procedures with updating values, such as value function iterations. In this case, trying to create a copy of one array but making rather a new pointer instead will result in meeting the convergence criterion immediately after the first iteration (*terrible sentence*). Obviously, this will not be correct.

A remedy to this issue can be simply addressed by using `deepcopy`, which creates a *new object in the memory* that contains the same information as the copied object.

```
@show V₀ = [1 2 3];
@show V₁ = deepcopy(V₀);
```

```
V₀ = [1 2 3] = [1 2 3]
V₁ = deepcopy(V₀) = [1 2 3]
```

Now changes in either of those objects *will not* affect values pointed by the other object:

```
@show V₁[1] = 420;
@show V₁;
@show V₀;

@show V₀[3] = 128;
@show V₁;
@show V₀;
```

```
V₁[1] = 420 = 420
V₁ = [420 2 3]
V₀ = [1 2 3]
V₀[3] = 128 = 128
V₁ = [420 2 3]
V₀ = [1 2 128]
```

An important thing that should be also stressed is that this feature works only in the presence of assignments without additional operations. If, for instance, code `V₁ = 2*V₀` involves two operations: (1) multiplication `*(2,V₀)`; and (2) making pointer `V₁` to memory containing *results* from computing `*(2,V₀)` in the past. This means that memory spaces pointed by `V₁` and by `V₀` are two different things.

```
@show V₀ = [1 2 3];
@show V₁ = 2*V₀;
```

```
V₀ = [1 2 3] = [1 2 3]
V₁ = 2V₀ = [2 4 6]
```

```
@show V₁[1] = 420;
@show V₁;
@show V₀;

@show V₀[3] = 128;
@show V₁;
@show V₀;
```

```
V₁[1] = 420 = 420
V₁ = [420 4 6]
V₀ = [1 2 3]
V₀[3] = 128 = 128
V₁ = [420 4 6]
V₀ = [1 2 128]
```

Dictionaries

Dictionaries are very useful objects that are particularly useful in my daily workflow. They contain different objects and data structures of those types can be very different. Dictionaries are initialized with `Dict` and the list of the elements take arguments in the form `"name_of_the_object" => object`. An illustration below:

```
my_first_dict = Dict("X" => 2,
                     "Y" => 1:10)
```

```
Dict{String, Any} with 2 entries:
  "Y" => 1:10
  "X" => 2
```

Alternatively you can use a bit different notation, which I personally prefer (and I am not sure why): `Dict([ ("name_of_the_object1", object1) ])`. For example, if we want to create an object containing all parameters of our model, dictionaries are good candidates for it:

```
params = Dict([
                ("β" , (1/1.04)^(1/12)),
                ("b" , [.55 0]) ,
                ("vec_states" , 1:2) ,
                ("π", [1:10 20:29])
            ])
```

```
Dict{String, Any} with 4 entries:
  "b"          => [0.55 0.0]
  "π"          => [1 20; 2 21; … ; 9 28; 10 29]
  "vec_states" => 1:2
  "β"          => 0.996737
```

Accessing elements of dictionaries has some similarities with arrays. The main difference is that we have to use names in quotates instead of indices:

```
@show params["β"]
params["π"]
```

```
params["β"] = 0.9967369426185624
```

```
10×2 Matrix{Int64}:
  1  20
  2  21
  3  22
  4  23
  5  24
  6  25
  7  26
  8  27
  9  28
 10  29
```

We can normally work on objects inside dictionaries. Suppose we want to increase values of all rows but the last one of the first column of array `π` by one we can do it by:

```
params["π"][1:(end-1),1]  .+= 1;
params["π"]
```

```
10×2 Matrix{Int64}:
  2  20
  3  21
  4  22
  5  23
  6  24
  7  25
  8  26
  9  27
 10  28
 10  29
```

We can also add new elements to existing dictionaries. It is super simple:

```
params["Σ"] = 12
```

```
12
```

Now in in `params` there is a new object `Σ`:

```
params
```

```
Dict{String, Any} with 5 entries:
  "Σ"          => 12
  "b"          => [0.55 0.0]
  "π"          => [2 20; 3 21; … ; 10 28; 10 29]
  "vec_states" => 1:2
  "β"          => 0.996737
```

In my own workflow, dictionaries are super useful for putting all blocks of models into one object. I create a dictionary `economy` with all parameters, value functions, simulations inside. Thanks to this during analyses I can explore different parametrizations, counterfactual calibrations and what not in a very organized way.

Too many repetitions of dictionary names? `@unpack` should help!

Suppose that we have a dictionary `equation` containing a range of argumnents $\mathrm{x} = \left(1 + \frac{(i-1)}{10}\right)_{i=1}^{100}$ and values of coefficients $(a = 1, b = 12, c = \pi)$ characterizing a certain quadratic equation $ax^2 + bx + c$. Then, our dictionary will be of the following form:

```
equation  = Dict([
                    ("x" , collect(range(1, step=.1, length=100)) ),
                    ("a", 1),
                    ("b", 12),
                    ("c", π)
                ])
```

```
Dict{String, Any} with 4 entries:
  "c" => π
  "x" => [1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9  …  10.0, 10.1, 10.2…
  "b" => 12
  "a" => 1
```

Now suppose we want to evaluate the function calibrated with $(a = 1, b = 12, c = \pi)$ at x. One way to do it is as follows:

Here I broadcast functions using `@.` macro. Broadcasting in general is discussed in this section, while `@.` macro in this subsection.

```
equation["y"] = @. equation["a"]*equation["x"]^2 + equation["b"]*equation["x"] +
equation["c"]
```

```
100-element Vector{Float64}:
  16.141592653589793
  17.551592653589793
  18.981592653589793
  20.431592653589796
  21.90159265358979
  23.391592653589793
  24.9015926535898
  26.431592653589792
  27.981592653589797
  29.55159265358979
  31.141592653589793
  32.751592653589796
  34.38159265358979
   ⋮
 216.78159265358983
 219.95159265358978
 223.14159265358978
 226.35159265358976
 229.58159265358978
 232.8315926535898
 236.10159265358982
 239.39159265358978
 242.70159265358978
 246.03159265358974
 249.38159265358982
 252.7515926535898
```

The code works but it does not have the nicest look and might be difficult to change. In this case, macro `@unpack` from package `Parameters` can be helpful in making the code more legible. This macro has the following syntax: `@unpack object_1, object_2 = dictionary_1`. `@unpack` extracts objects with names `object_1` and `object_2` from `dictionary_1` and makes them available in the current local scope. `dictionary_1` might have other objects inside but `@unpack` extracts only those explicitly mentioned objects. Below you can see how it works

Note to myself: in the future you need to write a section on scopes.

```
@show a #object a is only defined in dictionary equation, so you cannot access it
```

```
UndefVarError: a not defined

Stacktrace:
 [1] top-level scope
   @ show.jl:955
 [2] eval
   @ ./boot.jl:360 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, 
filename::String)
   @ Base ./loading.jl:1094
```

```
using Parameters
@unpack a, b, c, x = equation
@show a #now it works
```

```
a = 1
```

```
1
```

Now to evaluate our quadratic function at x we can do it by:

```
equation["y"] = @. a*x^2 + b*x + c
```

```
100-element Vector{Float64}:
  16.141592653589793
  17.551592653589793
  18.981592653589793
  20.431592653589796
  21.90159265358979
  23.391592653589793
  24.9015926535898
  26.431592653589792
  27.981592653589797
  29.55159265358979
  31.141592653589793
  32.751592653589796
  34.38159265358979
   ⋮
 216.78159265358983
 219.95159265358978
 223.14159265358978
 226.35159265358976
 229.58159265358978
 232.8315926535898
 236.10159265358982
 239.39159265358978
 242.70159265358978
 246.03159265358974
 249.38159265358982
 252.7515926535898
```

You have to remember that in `@unpack`, the assignment (`=`) operator works normally... for `Julia`. This means that if `object_1` is a scalar while `object_2` is an array, working with those objects can be very different. This topic was discussed in this section. Below you can find an example illustrating those differences:

```
@show equation["a"];
@show a;

@show a += 1;

@show equation["a"]; # `a += 1;` didn't change  the value of `equation["a"]`
@show a;             # `a += 1;`        changed the value of `a`
```

```
equation["a"] = 1
a = 1
a += 1 = 2
equation["a"] = 1
a = 2
```

```
2
```

```
@show equation["x"];
@show x;


@show x .+= 1;

@show equation["x"]; # `x .+= 1;`        CHANGED the value of `equation["x"]`
@show x;             # `x .+= 1;`        changed the value of `x`
```

```
equation["x"] = [1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0, 2.1, 2.2,
2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8,
3.9, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5.0, 5.1, 5.2, 5.3, 5.4,
5.5, 5.6, 5.7, 5.8, 5.9, 6.0, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 7.0,
7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 8.0, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6,
8.7, 8.8, 8.9, 9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9, 10.0, 10.1, 10.2,
10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9]
x = [1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0, 2.1, 2.2, 2.3, 2.4,
2.5, 2.6, 2.7, 2.8, 2.9, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4.0,
4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5.0, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6,
5.7, 5.8, 5.9, 6.0, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 7.0, 7.1, 7.2,
7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 8.0, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8,
8.9, 9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9, 10.0, 10.1, 10.2, 10.3,
10.4, 10.5, 10.6, 10.7, 10.8, 10.9]
x .+= 1 = [2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.0, 3.1, 3.2, 3.3,
3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9,
5.0, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 6.0, 6.1, 6.2, 6.3, 6.4, 6.5,
6.6, 6.7, 6.8, 6.9, 7.0, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 8.0, 8.1,
8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7,
9.8, 9.9, 10.0, 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9, 11.0, 11.1,
11.2, 11.3, 11.4, 11.5, 11.6, 11.7, 11.8, 11.9]
equation["x"] = [2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.0, 3.1, 3.2,
3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8,
4.9, 5.0, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 6.0, 6.1, 6.2, 6.3, 6.4,
6.5, 6.6, 6.7, 6.8, 6.9, 7.0, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 8.0,
8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6,
9.7, 9.8, 9.9, 10.0, 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9, 11.0,
11.1, 11.2, 11.3, 11.4, 11.5, 11.6, 11.7, 11.8, 11.9]
x = [2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.0, 3.1, 3.2, 3.3, 3.4,
3.5, 3.6, 3.7, 3.8, 3.9, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5.0,
5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 6.0, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6,
6.7, 6.8, 6.9, 7.0, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 8.0, 8.1, 8.2,
8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8,
9.9, 10.0, 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9, 11.0, 11.1, 11.2,
11.3, 11.4, 11.5, 11.6, 11.7, 11.8, 11.9]
```

```
100-element Vector{Float64}:
  2.0
  2.1
  2.2
  2.3
  2.4
  2.5
  2.6
  2.7
  2.8
  2.9
  3.0
  3.1
  3.2
  ⋮
 10.8
 10.9
 11.0
 11.1
 11.2
 11.3
 11.4
 11.5
 11.6
 11.7
 11.8
 11.9
```

While `a+=1` *did not* have impact on `equation["a"]`, `x .+= 1` affected `equation["x"]`. It is important to remember this feature of the language.

Ranges

Ranges are quite intuitive objects. They collect elements from a certain range of values. They can be created either by using `:` operator (just like in `R` and `Matlab`) or by using `range` function. For instance, if we want to create a range from 5 through 120, we can use two options:

```
@show 5:120;
@show range(5, stop=120);
```

```
5:120 = 5:120
range(5, stop = 120) = 5:120
```

Notice that you are not allowed to use `range(5,120)` because at least one of `length`, `stop`, or `step` must be specified explicitely.

If we want, we can set the size step. Again, it can be done in two ways:

```
@show 5:.5:120;
@show range(5, stop=120, step=.5);
```

```
5:0.5:120 = 5.0:0.5:120.0
range(5, stop = 120, step = 0.5) = 5.0:0.5:120.0
```

You can perform normal arithmetic operations:

```
(1:5)*2
```

```
2:2:10
```

However, be careful about the order of operations. Arithmetic operations have a higher priority in `Julia`. For `Matlab` users that is the same, but it might be problematic for `R` users, where the order is different. In `Julia`if you type `1:5+1`, you will give you `1:6`, while in `R` you will get `2:6`.

```
1:5+1
```

```
1:6
```

Another important difference between other languages and `Julia` is the fact that arrays and ranges are not compatible with each other. This means that arithmetic operations will give you an error message, even if objects are of the same size:

```
1:3 + [1; 2; 3]
```

```
MethodError: no method matching +(::Int64, ::Vector{Int64})

For element-wise addition, use broadcasting with dot syntax: scalar .+ array

Closest candidates are:

  +(::Any, ::Any, ::Any, ::Any...) at operators.jl:560

  +(::T, ::T) where T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128, UInt16,
UInt32, UInt64, UInt8} at int.jl:87

  +(::Union{Int16, Int32, Int64, Int8}, ::BigInt) at gmp.jl:534

  ...



Stacktrace:

 [1] top-level scope

   @ In[293]:1

 [2] eval

   @ ./boot.jl:360 [inlined]

 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)

   @ Base ./loading.jl:1094
```

You can extract all element from a range and create an array by using `collect`:

```
collect(1:3)
```

```
3-element Vector{Int64}:
 1
 2
 3
```

In this case, arithmetic operations are allowed:

```
collect(1:3) + [1; 2; 3]
```

```
3-element Vector{Int64}:
 2
 4
 6
```

Tuples

To be written. An example of a tuple:

```
(1,2,3)
```

```
(1, 2, 3)
```

Sets

To be written

Broadcasting: arrays meet scalars

In the section describing arrays, we faced one seemingly bizzare obstacle in operating with objects of different data structures. In our discussed example we tried to add a scalar number, $1,$ to a matrix A by running $\mathrm{A}+1$. In `Julia` it is not possible. While trying running `A+1` you get an error message. We addressed this problem by using function `repeat`. Here I present a much more recommended approach and to this end I will use function `broadcast`.

Broadcasting applies a certain function `f` over the sequence of elements of the provided input objects. In our example, we want to broadcast function `+()` over elements of `A` and `1`:

```
A = [1 2; 3 4] #our matrix

output = broadcast(+, A , 1)
```

```
2×2 Matrix{Int64}:
 2  3
 4  5
```

The broadcasting procedure adds `A[1,1]` and `1` and assings the result to element `output[1,1]`. Next, it adds `A[2,1]` and `1` and assings the result to element `output[2,1]`. In total, the operation is conducted four times and as a result you get a new object `output` of size 2×2.

Broadcasting allows to use not only arrays and scalars but also arrays of different sizes in a very similar way:

```
broadcast(+, A , [1 2])
```

```
2×2 Matrix{Int64}:
 2  4
 4  6
```

Dot (`.`) notation for broadcasting

The number of input objects depends on the number of required inputs of the broadcasted function. To illustrate this, suppose that we want to create an exponential grid of n elements from k_1 to k_n. In a grid of this type, the elements are defined as follows:

$$\forall_{i \in \{1, \dots, n\}} \ln k_i = \ln k_1 + (i-1)d,$$

where $d = \frac{\ln k_n - \ln k_1}{n-1}$. Then, the grid on the logarithmic scale $(\ln k_i)_{i=1}^{n}$ can be computed as follows:

```
k₁    =  1;
kₙ    = 12;
n     = 1000;
d     = (kₙ-k₁)/(n-1);
ln_k  = collect( range(log(k₁), stop=log(kₙ), length=n) );
```

Equipped with $(\ln k_i)_{i=1}^{n}$, we would like to recover the grid with true values of $(k_i)_{i=1}^{n}$. We can do it by exponentiating each element of `ln_k`. `broadcast` might be very useful for this:

```
k = broadcast(exp, ln_k)
```

```
1000-element Vector{Float64}:
  1.0
  1.0024904901749636
  1.004987182891239
  1.007490093596194
  1.0099992377756684
  1.0125146309540696
  1.0150362886944675
  1.0175642265986928
  1.0200984603074312
  1.0226390055003225
  1.0251858778960556
  1.0277390932524673
  1.0302986673666386
  ⋮
 11.676115152945892
 11.705194403016046
 11.734346074672795
 11.76357034828139
 11.792867404656281
 11.822237425062225
 11.851680591215429
 11.881197085284661
 11.910787089892368
 11.940450788115827
 11.97018836348827
 12.0
```

Notice that the distance between two subsequent elements of `ln_k` is constant and equal to $d = \frac{\ln k_n - \ln k_1}{n-1}$, while the distance on `k` is increasing with each element. Check this by comparing vectors of distances, `ln_k[2:end] - ln_k[1:end-1]` and `k[2:end] - k[1:end-1]`.

Procedure `broadcast` has a convenient short dot (`.`) notation. Instead of using `broadcast(f, As)`, we can do exactly the same thing by adding `.` at the end of function `f`, *i.e.* `f.(As)`. In our example it will be:

```
exp.(ln_k)
```

```
1000-element Vector{Float64}:
  1.0
  1.0024904901749636
  1.004987182891239
  1.007490093596194
  1.0099992377756684
  1.0125146309540696
  1.0150362886944675
  1.0175642265986928
  1.0200984603074312
  1.0226390055003225
  1.0251858778960556
  1.0277390932524673
  1.0302986673666386
  ⋮
 11.676115152945892
 11.705194403016046
 11.734346074672795
 11.76357034828139
 11.792867404656281
 11.822237425062225
 11.851680591215429
 11.881197085284661
 11.910787089892368
 11.940450788115827
 11.97018836348827
 12.0
```

Arithmetic operations such as `+`, `-`, `^`, `/`, `*` can be broadcasted by adding `.` *in front of* operators:

Sometimes `.` operator generates some ambiguity. For example, in `2.+A` `Julia` does not know whether `.` refers to `2.` or `.+`. In this case, you need to make a space in a proper place: `2 .+A` or `2. +A`.

```julia
A .+ [1 2]
```

```
2×2 Matrix{Int64}:
 2  4
 4  6
```

Don't use `repeat` for what can be done with `broadcast`

In section on arrays, I discussed a method involving `repeat` instead of `broadcast`. Now, I will show how inefficient it is in comparison to broadcasting. For this, I am going compare both approaches in adding 1 to each element of an array of size $100,000 \times 10,000$. In the exercise macro `@btime` from `BenchmarkTools` is used to measure the execution time of both methods.

```julia
using BenchmarkTools

X = repeat(collect(1:100), 1000, 10000);
@show size(X)
```

```
size(X) = (100000, 10000)
```

```
(100000, 10000)
```

```julia
@btime X .+ 1;
```

```julia
@btime X + repeat([1], 100000, 10000);
```

```
  15.541 s (7 allocations: 14.90 GiB)
```

The broadcasted (execution time: 3.308 s) function is almost five times faster than using `repeat` (execution time: 15.541 s). The main reason for this is that `repeat([1], 100000, 10000)` generates a new large object, while in `broadcast` this step is skipped.

Too many dots? `@.` is the answer!

Suppose that we have vector $\mathbf{x} = (1, 2, 3, 4, 5)'$ and we would like to perform the following operation for each element of the vector:

$$\forall_{i \in \{1, \ldots, 5\}} \frac{x_i^2 - 1}{3}.$$

Using `broadcast`, the code will look as below:

```
x = 1:5;

broadcast( /, broadcast( - , broadcast(^, x, 2), 1), 3)
```

```
5-element Vector{Float64}:
 0.0
 1.0
 2.6666666666666665
 5.0
 8.0
```

As can be seen, a relatively simple operation is written in a very complicated way making the code extremely illegible. With `.` notation it can be slightly improved:

```
(x.^2 .- 1)./3
```

```
5-element Vector{Float64}:
 0.0
 1.0
 2.6666666666666665
 5.0
 8.0
```

The code looks much better now but still the number of `.`'s might be overwhelming for some people. This issue can be addressed by applying `@.` at the beginning of the line. This macro converts *all* functions in the line to broadcasted functions (actually `@.` can be put in any place of the line. The scope of the macro is from symbol `@.` to the the end of the current line). In our example we will have:

```
@. (x^2 - 1)/3
```

```
5-element Vector{Float64}:
 0.0
 1.0
 2.6666666666666665
 5.0
 8.0
```

Some caveats on thoughtless usage of `@.`

Suppose that we have vector $\mathbf{x} = (1, 2, 3, 4, 5)'$ and we would like to compute the variance of its elements by using a textbook formula:

$$\mathrm{Var}(\mathbf{x}) = \frac{\sum_{i=1}^{n} \left(x_i - \frac{\sum_{i=1}^{n} x_i}{n} \right)^2}{n - 1}.$$

To avoid multiple `.`'s, we might be tempted to use `@.` notation. However, the result can be quite surprising:

```
@. sum( (x - sum(x)/length(x))^2 )/(length(x) - 1)
```

```
5-element Vector{Float64}:
 NaN
 NaN
 NaN
 NaN
 NaN
```

To understand what happens, recall that macro `@.` broadcasts *all* functions in the line. In particular, it means that `sum` and `lenght` are broadcasted too. This is not what we want. `sum.(x)` computes for each element s the sum of all elements, *i.e.* $\sum_{i=s}^{s} x_i$, which is nothing else than x_s itself. In a similar fashion, `length.(x)` computes for each element s the length of this element, which is always 1. As a result we have:

```
@show sum.(x);
@show length.(x);
```

```
sum.(x) = [1, 2, 3, 4, 5]
length.(x) = [1, 1, 1, 1, 1]
```

For illustration reasons, In this section, intentionally I use neither `mean` nor `var` from package `Statistics`.

To compute the variance we need to use `.` notation only for `-()` and `^()`:

```
sum( (x .- sum(x)/length(x)).^2 )/(length(x) - 1)
```

```
2.5
```

`broadcast` vs. `map`

to be written

```
@show map(+, A, 1);
@show broadcast(+, A, 1);
```

```
map(+, A, 1) = [2]
broadcast(+, A, 1) = [2 3; 4 5]
```

Functions

A Function *by Any Other Name*

Functions by `function`

Functions in `Julia` can be defined in several ways. One and the most typical is simply by using `function`. The structure for using `function` is as follows:

```
    function function_name(argument_1, argument_2)

        some_operations
        output = some_operations

        return output
    end
```

An example of defining a very simple functions and its execution:

```
function add_numbers(x, y)
    output = x + y
    return output
end
```

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

```
@show add_numbers(5, 12)
```

```
add_numbers(5, 12) = 17
```

```
17
```

```julia
@show w = add_numbers(10,10)
```

```
w = add_numbers(10, 10) = 20
```

```
20
```

Functions do not need arguments:

```julia
function add_first_ten_numbers()
    Σ = sum( 1:10 )
    return Σ
end

add_first_ten_numbers()
```

```
55
```

Functions can return many outpus:

```julia
function add_and_multiply(x, y)
    out1 = x+y
    out2 = x*y

    return out1, out2

end


@show res1, res2 = add_and_multiply(2,3);
```

```
(res1, res2) = add_and_multiply(2, 3) = (5, 6)
```

```julia
res1
```

```
5
```

```julia
res2
```

```
6
```

“Assignment-form” functions

There is a compact alternative for `function`: `f(args) = some operations`. Example below:

```julia
f(x,y,z) = (x^2 + y)/z
@show f(2,1,3)
```

```
f(2, 1, 3) = 1.6666666666666667
```

```
1.6666666666666667
```

Functions in this form can contain multiple lines by using `begin`/`end` block:

```julia
g(x) = begin
        y = x^2
        y += 1
        return(y)
    end
```

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

Anonymous functions

Sometimes we do not need to create named functions. Anonymous functions allow to create a normal function without names. Their structure is `arg -> operations` with one argument or `(arg1, arg2, arg3) -> operations` with many arguments:

```
x -> x^2
(x,y) -> x+y
```

```
#3 (generic function with 1 method)
```

We can evaluate functions at some values in the following way:

```
(x -> x^2)(3)
```

```
9
```

`(x -> x^2)` is the function and `3` is the argument of this function.

Just like assignment-form functions, anonymous functions can be defined within multiple lines using `begin`/`end` blocks:

```
x -> begin
        x = x + 1
        x = 2*x
        return(x)
    end
```

```
#7 (generic function with 1 method)
```

`map` and anonymous functions

Recall our example from here. We can use anonymous functions as the argument of `map` function to compute $\frac{x_i^2-1}{3}$ for each element of vector $\mathbf{x}$:

```
x = 1:5;

map(xᵢ -> (xᵢ^2-1)/3, x)
```

```
5-element Vector{Float64}:
 0.0
 1.0
 2.6666666666666665
 5.0
 8.0
```

Function `map(f, c)` transform elements of `c` by applying `f` (which can be an anonymous but also a named function) to each element. In most cases this function provides very similar results to `broadcast` discussed before. In my own workflow when I work with anonymous functions I use `map`, while I use `broadcast` only in the dot (`.`) convention.

Scopes: minor digression

Objects defined outside the function, by default, are not available and their values cannot be changed.

```
σ = 12

function modify_σ()
    σ = σ + 1;
    return(σ)
end

modify_σ()
```

```
UndefVarError: σ not defined

Stacktrace:
 [1] modify_σ()
   @ Main ./In[14]:4
 [2] top-level scope
   @ In[14]:8
 [3] eval
   @ ./boot.jl:360 [inlined]
 [4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, 
filename::String)
   @ Base ./loading.jl:1094
```

Sometimes, if we have a *good* reason, we may want to change objects created outside the function and not passed as arguments. To do it we need to point which objects were created beyond the scope of the function. This can be done with the use of `global`:

```
σ = 12

function modify_σ()
    global σ
    σ = σ + 1;
    return(σ)
end

modify_σ()
```

```
13
```

Not only does this function access `σ` but also changes its value in the global scope:

```
@show σ
```

```
σ = 13
```

```
13
```

It is convenient to use dictionaries as arguments of functions. Dictionaries passed as the arguments can be unpacked using the macro `@unpack`, which we already discussed here. This way, inside the function we can write a clean code without cluttering our global space. Supposed that we want to have a function that evaluates the function at x with parameters *twice as large* as the input arguments (a, b, c).

```
equation  = Dict([
                    ("x" , collect(range(1, step=.1, length=100)) ),
                    ("a", 1),
                    ("b", 12),
                    ("c", π)
                ])
```

```
Dict{String, Any} with 4 entries:
  "c" => π
  "x" => [1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9  …  10.0, 10.1, 10.2…
  "b" => 12
  "a" => 1
```

```
using Parameters

function multiply_coefficients(X)
    @unpack  a, b, c  = X;

    a *= 2;
    b *= 2;
    c *= 2;


    output = deepcopy(X)

    #instead of writing:
    #output["y"] = @. X["a"]*X["x"]^2 + X["b"]*X["x"] + X["c"]
    #we have:

    output["y"] = @. a*x^2 + b*x + c

    output["a"] = a;
    output["b"] = b;
    output["c"] = c;



    return output

end
```

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

```
new_result = multiply_coefficients(equation)
```

```
Dict{String, Any} with 5 entries:
  "c" => 6.28319
  "x" => [1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9  …  10.0, 10.1, 10.2…
  "b" => 24
  "a" => 2
  "y" => 32.2832
```

The result of executing function is assigned to `new_result`. Inside function `multiply_coefficients`, we were referring to `a`, `b`, `c` normally. However, after function execution unpacked arguments are still unavailable in the global scope:

```
@show a
```

```
UndefVarError: a not defined

Stacktrace:
 [1] top-level scope
   @ show.jl:955
 [2] eval
   @ ./boot.jl:360 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, 
filename::String)
   @ Base ./loading.jl:1094
```

Caveat on passing arrays and scalars in functions

Consider the following code:

```julia
using LinearAlgebra

x = 2

y = [1 2;3 4]

set_to_1(scal, matrix)= begin
                scal     = 1;
                matrix  .= 1;
                return scal, matrix
            end

@show x
@show y

@show set_to_1(x, y)

@show x
@show y;
```

```
x = 2
y = [1 2; 3 4]
add_1(x, y) = (1, [1 1; 1 1])
x = 2
y = [1 1; 1 1]
```

Function `set_to_1` did not modify scalar `x` but modified array `y`. It is because of how the assignment operator works in `Julia`. Notice that `matrix .= 1` has an assignment operator `=`, which is applied to an array. This mean that each element *allocated in the memory* pointed by `matrix` has a new value. To understand it better recall our earlier discussion on the assignment operator with arrays. The result of this operation preserves after the execution of the function is finished.

Note

The behavior of the assignment operator inside functions is quite different from other languages. There is some logic behind it but you have to remember it, especially if you are used to other languages as well.

One way to write the function without changing the input array is as follows:

```julia
using LinearAlgebra

x = 2

y = [1 2;3 4]

set_to_1(scal, matrix)= begin
                scal     = 1;
                matrix_op = deepcopy(matrix) #new line
                matrix_op  .= 1; #we use `matrix_op` instead of `matrix`
                return scal, matrix_op
            end

@show x
@show y

@show set_to_1(x, y)

@show x
@show y;
```

```
x = 2
y = [1 2; 3 4]
add_1(x, y) = (1, [1 1; 1 1])
x = 2
y = [1 2; 3 4]
```

Multi dispatch

to be written

```
function Fun1(x::Float64,y::Float64)
    return(x-y)
end
```

```
Fun1 (generic function with 2 methods)
```

```
function Fun1(x::Array{Float64},y::Array{Float64})
    return(x .- y)
end
```

```
Fun1 (generic function with 3 methods)
```

```
Fun1([1; 2], [3; 2])
```

```
2-element Vector{Int64}:
 -2
  0
```

```
Fun1(3,2)
```

```
1
```

Control flow

Boolean type

`true`

Loops

```
for i in [1 2 3 4]
    println("$i=i")
end
```

```
1=i
2=i
3=i
4=i
```

```
A = [1 2; 3 4]
```

```
2×2 Matrix{Int64}:
 1  2
 3  4
```

```
for i in A
    println("i=$i")
end
```

```
i=1
i=3
i=2
i=4
```

Random numbers

RNG

Drawing from arbitrary distributions

Monte Carlo

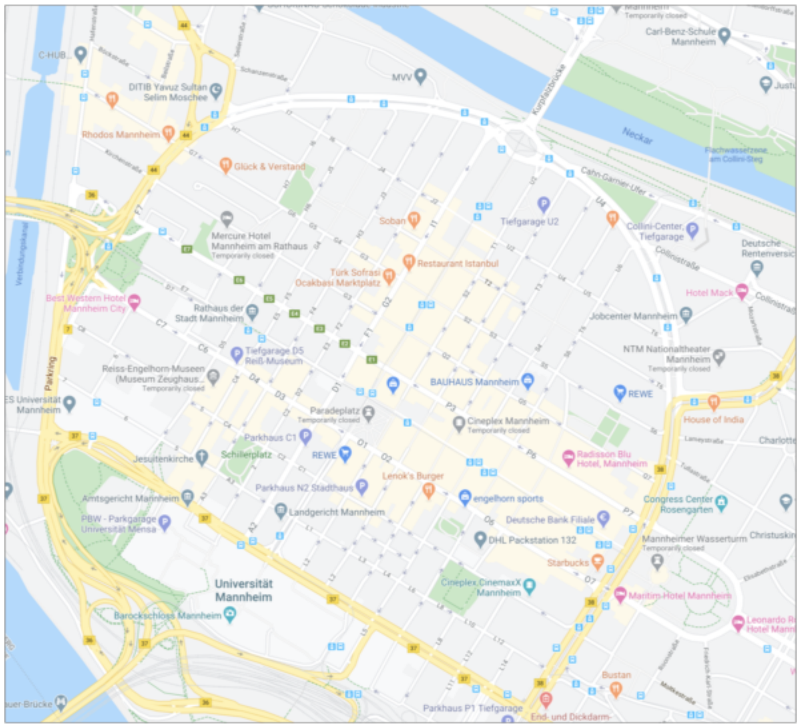
Application: Monte Carlo in Mannheim

tbd

```
Pkg.add(["FileIO", "ImageMagick", "ImageIO"])
```

```
UndefVarError: Pkg not defined

Stacktrace:
 [1] top-level scope
   @ In[1]:1
 [2] eval
   @ ./boot.jl:360 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
   @ Base ./loading.jl:1094
```

```
using Images, FileIO
img_path = "Mannheim_quadrate.png"
img = load(img_path)
```



```
typeof(img)
```

```
Matrix{RGBA{N0f8}} (alias for Array{RGBA{Normed{UInt8, 8}}, 2})
```

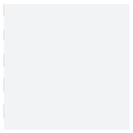
```
size(img)
```

```
(2156, 2374)
```

```
[234, 1413,1]
```

```
3-element Vector{Int64}:
  234
 1413
    1
```

```
img[500,500]
```



```
using Random
```

```
img = rand(5,5)
```

```
5×5 Matrix{Float64}:
 0.112738   0.883755   0.289751  0.589134   0.717956
 0.945748   0.0812687  0.716711  0.7295     0.475783
 0.0930118  0.359034   0.14368   0.191692   0.96892
 0.413357   0.404343   0.537111  0.0702231  0.370256
 0.28802    0.148923   0.432021  0.569693   0.0366185
```

Pipes

```
exp.( collect( range(log(1), stop=log(42), length=73) ) )
```

```
73-element Vector{Float64}:
  1.0
  1.0532831317160591
  1.1094053555575891
  1.1685179472442655
  1.2307802429398609
  1.2963600687379486
  1.3654341930319522
  1.4381888029888847
  1.5148200064111028
  1.5955343603388272
  1.6805494278182591
  1.7700943643360474
  1.8644105355008187
  ⋮
 23.727548568154436
 24.991826663810592
 26.323469455763327
 27.726066345998433
 29.203397991080454
 30.7594464927957
 32.3984061317844
 34.12469467309464
 35.94296527413145
 37.85811902709873
 39.87531816974189
 42.00000000000001
```

```
using Pipe
@pipe range(log(1), stop=log(42), length=73) |> collect |> exp.(_)
```

```
73-element Vector{Float64}:
  1.0
  1.0532831317160591
  1.1094053555575891
  1.1685179472442655
  1.2307802429398609
  1.2963600687379486
  1.3654341930319522
  1.4381888029888847
  1.5148200064111028
  1.5955343603388272
  1.6805494278182591
  1.7700943643360474
  1.8644105355008187
  ⋮
 23.727548568154436
 24.991826663810592
 26.323469455763327
 27.726066345998433
 29.203397991080454
 30.7594464927957
 32.3984061317844
 34.12469467309464
 35.94296527413145
 37.85811902709873
 39.87531816974189
 42.00000000000001
```

```
@pipe range(log(1), stop=log(42), length=73) |>
                                     collect |>
                                     exp.(_) |>
            map(x -> 2*(x-1)/(42-1) - 1, _ ) #standardizing to [-1, 1]
```

```
73-element Vector{Float64}:
 -1.0
 -0.997400822843119
 -0.9946631533874347
 -0.991779612329548
 -0.9887424271736653
 -0.9855434112810757
 -0.9821739418033194
 -0.9786249364395666
 -0.974886828955556
 -0.970949543398106
 -0.9668024669356947
 -0.9624344212519002
 -0.9578336324145942
  ⋮
  0.10866090576363097
  0.17033300799076057
  0.23529119296406464
  0.3037105534633382
  0.3757755117600221
  0.45168031672174136
  0.5316295674041169
  0.6158387645412018
  0.7045348914210465
  0.7979570257121331
  0.8963569838898482
  1.0000000000000004
```

Bisection

```
using Roots  #required by find_zero
using Printf #required by @sprintf
using Plots
```

Root-finding procedures

Suppose that we want to find the root of the following function:

$$Fun_1(x) = x - x \cdot \sqrt{x+1} + 5$$

```
Fun1(x)=x-x*√(x+1)+5
```

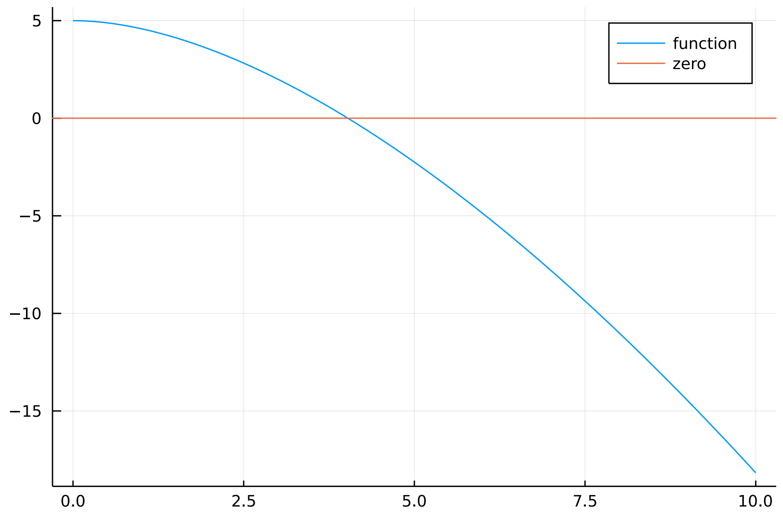
```
Fun1 (generic function with 1 method)
```

```
grid_points = 0:.1:10
fun_val=Array{Float64,1}(undef, length(grid_points))

for (count, arg)∈enumerate(grid_points)
    fun_val[count] = Fun1(arg)
end

map(Fun1,grid_points)
```

```
101-element Vector{Float64}:
   5.0
   4.995119115182985
   4.980910976997934
   4.957947372470259
   4.92671361735203
   4.887627564304205
   4.841053361559589
   4.7873116632716295
   4.726687370800101
   4.6594356123118805
   4.585786437626905
   4.505948557919162
   4.420112363097041
   ⋮
 -14.10319624614305
 -14.460498941515414
 -14.820252419368678
 -15.18244373771521
 -15.547060156739057
 -15.914089133602552
 -16.283518317437338
 -16.65533554451143
 -17.02952883356448
 -17.40608638130377
 -17.784996558053976
 -18.166247903553995
```

This function looks as follows:

```
plot(grid_points, fun_val, label="function")
hline!([0], label="zero")
```

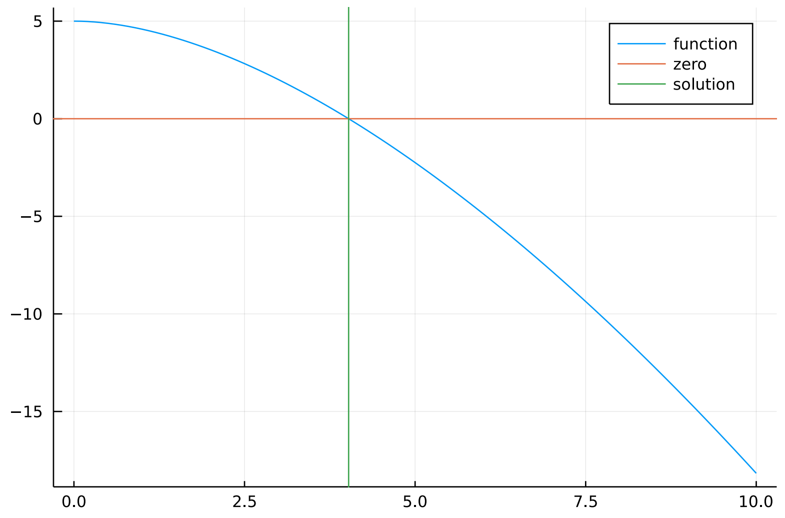


Bisection

```
xₗ  = 0
xᵣ = 10
mid_point  = (xᵣ + xₗ)/2
Fun1(  xₗ  )*Fun1( xᵣ ) #it's negative

for i ∈ 1:100

    mid_point  = (xᵣ + xₗ)/2
    if Fun1( mid_point ) * Fun1(  xₗ  ) < 0
        xᵣ = mid_point
    else
        xₗ = mid_point
    end
    println("i=$i, mid_point = $(round(mid_point, digits=3)),
Fun1($(round(mid_point, digits=3))) = $(round(Fun1(mid_point), digits=3))")

    if abs(Fun1(mid_point)) < .001 #Tolerance error
        println(" 😃   abs(Fun1(mid_point))=$(abs(Fun1(mid_point))), which less
than .001!!!")
        break
    end

end #i ∈ 1:100
```

```
i=1, mid_point = 5.0, Fun1(5.0) = -2.247
i=2, mid_point = 2.5, Fun1(2.5) = 2.823
i=3, mid_point = 3.75, Fun1(3.75) = 0.577
i=4, mid_point = 4.375, Fun1(4.375) = -0.768
i=5, mid_point = 4.062, Fun1(4.062) = -0.078
i=6, mid_point = 3.906, Fun1(3.906) = 0.254
i=7, mid_point = 3.984, Fun1(3.984) = 0.089
i=8, mid_point = 4.023, Fun1(4.023) = 0.006
i=9, mid_point = 4.043, Fun1(4.043) = -0.036
i=10, mid_point = 4.033, Fun1(4.033) = -0.015
i=11, mid_point = 4.028, Fun1(4.028) = -0.005
i=12, mid_point = 4.026, Fun1(4.026) = 0.0
 😃   abs(Fun1(mid_point))=0.0004735185227016103, which less than .001!!!
```

```
plot(grid_points, fun_val, label="function")
hline!([0], label="zero")
vline!([mid_point], label="solution")
```



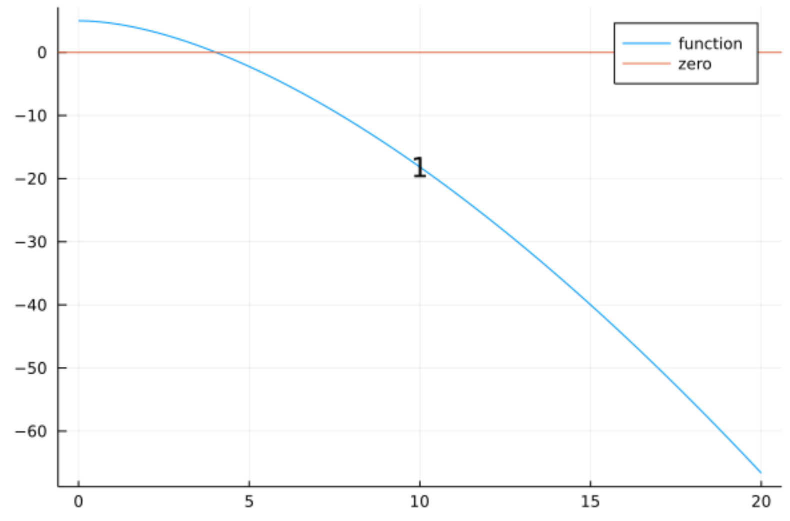
Ilustration of the algorithm

```julia
grid_points = 0:.1:20
fun_val=Array{Float64,1}(undef, length(grid_points))

for (count, arg)∈enumerate(grid_points)
    fun_val[count] = Fun1(arg)
end


plot(grid_points, fun_val, label="function")
hline!([0], label="zero")

xₗ  = Array{Float64, 1}(undef, 100)
xᵣ = Array{Float64, 1}(undef, 100)
mid_point = Array{Float64, 1}(undef, 100)

xₗ[1]    = 0
xᵣ[1]   = 20
mid_point[1] = (xₗ[1] +xᵣ[1] )/2


for i ∈ 1:99

    if Fun1(mid_point[i]) * Fun1(  xₗ[i]   ) < 0
         xᵣ[i+1] = mid_point[i]
         xₗ[i+1] = xₗ[i]
    else
         xₗ[i+1] = mid_point[i]
         xᵣ[i+1] = xᵣ[i]
    end

    mid_point[i+1] = (xₗ[i+1] + xᵣ[i+1] )/2


    println("i=$i, mid_point = $(round(mid_point[i+1], digits=3)),
Fun1($(round(mid_point[i+1], digits=3))) = $(round(Fun1(mid_point[i+1]),
digits=3))")

    if abs(Fun1(mid_point[i+1])) < .001 #Tolerance error
        println(" 😃   abs(Fun1(mid_point))=$(abs(Fun1(mid_point[i+1]))), which
less than .001!!!")

        break
    end

end #i ∈ 1:100
```

```
i=1, mid_point = 5.0, Fun1(5.0) = -2.247
i=2, mid_point = 2.5, Fun1(2.5) = 2.823
i=3, mid_point = 3.75, Fun1(3.75) = 0.577
i=4, mid_point = 4.375, Fun1(4.375) = -0.768
i=5, mid_point = 4.062, Fun1(4.062) = -0.078
i=6, mid_point = 3.906, Fun1(3.906) = 0.254
i=7, mid_point = 3.984, Fun1(3.984) = 0.089
i=8, mid_point = 4.023, Fun1(4.023) = 0.006
i=9, mid_point = 4.043, Fun1(4.043) = -0.036
i=10, mid_point = 4.033, Fun1(4.033) = -0.015
i=11, mid_point = 4.028, Fun1(4.028) = -0.005
i=12, mid_point = 4.026, Fun1(4.026) = 0.0
 😃   abs(Fun1(mid_point))=0.0004735185227016103, which less than .001!!!
```

```julia
anim = @animate for j∈1:10
    plot(grid_points, fun_val, label="function", xlim=[2.5 7.5])
    for i∈1:j
        annotate!([( mid_point[i], Fun1(mid_point[i]), "$i")])
    end
    hline!([0], label="zero")
end

gif(anim, "anim.gif", fps = 5)
```

```
┌ Info: Saved animation to
│   fn = /Users/pytka/Dropbox/My Mac (juprof-vwl06)/Documents/OwnLearning/julia-
intro/anim.gif
└ @ Plots /Users/pytka/.julia/packages/Plots/SVksJ/src/animation.jl:104
```



#BUILT-IN FUNCTIONS #Root finding procedure. #One of examples of such a procedure you should know: BISECTION.

```
solution = fzero(Fun1, 1)
```

```
4.026100199345813
```

```
Fun1(solution)
```

```
8.881784197001252e-16
```

Iterative root finding procedure

$$\begin{cases} Fun_1(x,y) = x - \sqrt{x+y} - 3 \\ Fun_2(x,y) = y - \sqrt{x + \frac{1}{y}} - 5 \end{cases}$$

```
Fun1(x,y)=x-√(x+y)-3
Fun2(x,y)=y-√(x+1/y)-5
```

```
Fun2 (generic function with 1 method)
```

```
y0 = .7
x0 = fzero( x -> Fun1(x,y0), 1)
Fun1(x0,y0)
```

```
0.0
```

```
#But
Fun2(x0,y0)
```

```
-6.929834998627596
```

```
y0 = fzero( y -> Fun2(x0,y), 1)

Fun2(x0,y0)
# Fun1(x0,y0)
```

```
0.0
```

```
Fun1(x0,y0)
```

```
-1.098452432583965
```

```
y0 = .7
for i∈1:9
    global x0 = fzero( x -> Fun1(x,y0), 1)
    println("Iteration no. $i. x₀=$x0, y₀=$y0,")
    println("Iteration no. $i. Fun1(x0,y0)=$(Fun1(x0,y0)),
Fun2(x0,y0)=$(Fun2(x0,y0))")
    println("---")
    global y0 = fzero( y -> Fun2(x0,y), 1)
    println("Iteration no. $i. x₀=$x0, y₀=$y0,")
    println("Iteration no. $i. Fun1(x0,y0)=$(Fun1(x0,y0)),
Fun2(x0,y0)=$(Fun2(x0,y0))")
    println("================================================================")
end

Fun1(x0,y0)
Fun2(x0,y0)
```

```
Iteration no. 1. x₀=5.487460691435179, y₀=0.7,
Iteration no. 1. Fun1(x0,y0)=0.0, Fun2(x0,y0)=-6.929834998627596
---
Iteration no. 1. x₀=5.487460691435179, y₀=7.371312241577558,
Iteration no. 1. Fun1(x0,y0)=-1.098452432583965, Fun2(x0,y0)=0.0
================================================================
Iteration no. 2. x₀=6.759035477189157, y₀=7.371312241577558,
Iteration no. 2. Fun1(x0,y0)=4.440892098500626e-16,
Fun2(x0,y0)=-0.2544631739539298
---
Iteration no. 2. x₀=6.759035477189157, y₀=7.624916086558346,
Iteration no. 2. Fun1(x0,y0)=-0.033582558383038474, Fun2(x0,y0)=0.0
================================================================
Iteration no. 3. x₀=6.797713766620498, y₀=7.624916086558346,
Iteration no. 3. Fun1(x0,y0)=4.440892098500626e-16,
Fun2(x0,y0)=-0.007357218192534987
---
Iteration no. 3. x₀=6.797713766620498, y₀=7.6322493687398625,
Iteration no. 3. Fun1(x0,y0)=-0.0009653637299931184, Fun2(x0,y0)=0.0
================================================================
Iteration no. 4. x₀=6.798825452905907, y₀=7.6322493687398625,
Iteration no. 4. Fun1(x0,y0)=4.440892098500626e-16,
Fun2(x0,y0)=-0.00021115812788430333
---
Iteration no. 4. x₀=6.798825452905907, y₀=7.6324598406132855,
Iteration no. 4. Fun1(x0,y0)=-2.770212906888503e-5, Fun2(x0,y0)=0.0
================================================================
Iteration no. 5. x₀=6.798857353783775, y₀=7.6324598406132855,
Iteration no. 5. Fun1(x0,y0)=4.440892098500626e-16,
Fun2(x0,y0)=-6.059131589353228e-6
---
Iteration no. 5. x₀=6.798857353783775, y₀=7.6324658800535685,
Iteration no. 5. Fun1(x0,y0)=-7.94902135670128e-7, Fun2(x0,y0)=8.881784197001252e-
16
================================================================
Iteration no. 6. x₀=6.798858269167314, y₀=7.6324658800535685,
Iteration no. 6. Fun1(x0,y0)=0.0, Fun2(x0,y0)=-1.7386426698351443e-7
---
Iteration no. 6. x₀=6.798858269167314, y₀=7.632466053352803,
Iteration no. 6. Fun1(x0,y0)=-2.280938415921696e-8, Fun2(x0,y0)=0.0
================================================================
Iteration no. 7. x₀=6.798858295433861, y₀=7.632466053352803,
Iteration no. 7. Fun1(x0,y0)=0.0, Fun2(x0,y0)=-4.988962132301822e-9
---
Iteration no. 7. x₀=6.798858295433861, y₀=7.632466058325552,
Iteration no. 7. Fun1(x0,y0)=-6.545062269935897e-10,
Fun2(x0,y0)=8.881784197001252e-16
================================================================
Iteration no. 8. x₀=6.79885829618757, y₀=7.632466058325552,
Iteration no. 8. Fun1(x0,y0)=4.440892098500626e-16,
Fun2(x0,y0)=-1.4315482133042678e-10
---
Iteration no. 8. x₀=6.79885829618757, y₀=7.632466058468243,
Iteration no. 8. Fun1(x0,y0)=-1.8780532684559148e-11, Fun2(x0,y0)=0.0
================================================================
Iteration no. 9. x₀=6.798858296209197, y₀=7.632466058468243,
Iteration no. 9. Fun1(x0,y0)=4.440892098500626e-16,
Fun2(x0,y0)=-4.107825191113079e-12
---
Iteration no. 9. x₀=6.798858296209197, y₀=7.632466058472337,
Iteration no. 9. Fun1(x0,y0)=-5.38680211548126e-13, Fun2(x0,y0)=0.0
================================================================
```

```
0.0
```

Price Search

```
# Parameters

α = .189 #price elasticity from Aguiar-Hurst (AER, 2007)
https://www.aeaweb.org/articles?id=10.1257/aer.97.5.1533
δ = .33
L = 40
R = 10
y = 1
β = .8867 #.997^L
```

```
0.8867
```

```
pR(cR) = (1/(1+cR/α))^(-α)
pW(cW) = ((1-δ)/(1+cW/α))^(-α)
cR(cW, cR) = √(β*pW(cW)/pR(cR)) * cW
```

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

```
cᴿ_value = .9 #Initial guess for retired consumption

#Implied value of working consumption given cᴿ_value:
cᵂ_value = fzero( cW -> pR(cR(cW,cᴿ_value))*cR(cW, cᴿ_value)*R + pW(cW)*cW*L-L*y,
1)

pR(cR(cᵂ_value,cᴿ_value))*cR(cᵂ_value, cᴿ_value)*R + pW(cᵂ_value)*cᵂ_value*L-L*y
```

```
0.0
```

```
#but cR(cᵂ_value , cᴿ_value) ≠ cᴿ_value --- contradiction
cR(cᵂ_value , cᴿ_value)
```

```
0.5523917350799546
```

```
println("-------------------------------------------")
println("👻 Iterative procedure STARTS:")
@time for i ∈ 1:7
    global cᵂ_value = fzero( cW -> pR(cR(cW,cᴿ_value))*cR(cW, cᴿ_value)*R +
pW(cW)*cW*L-L*y, 1)
    global cᴿ_value = cR(cᵂ_value, cᴿ_value)
    println("Iteration No. $i, cᴿ=$(@sprintf("%.5f", cᵂ_value)),
cᵂ=$(@sprintf("%.5f", cᴿ_value))")
end
println("-------------------------------------------")
```

```
-------------------------------------------
👻 Iterative procedure STARTS:
```

```
Iteration No. 1, cᴿ=0.58347, cᵂ=0.55239
Iteration No. 2, cᴿ=0.57968, cᵂ=0.56885
Iteration No. 3, cᴿ=0.57990, cᵂ=0.56790
Iteration No. 4, cᴿ=0.57989, cᵂ=0.56795
Iteration No. 5, cᴿ=0.57989, cᵂ=0.56795
Iteration No. 6, cᴿ=0.57989, cᵂ=0.56795
Iteration No. 7, cᴿ=0.57989, cᵂ=0.56795
  0.222149 seconds (308.59 k allocations: 12.947 MiB, 4.93% gc time, 100.20%
compilation time)
-------------------------------------------
```

```
pR(cR(cᵂ_value,cᴿ_value))*cR(cᵂ_value, cᴿ_value)*R + pW(cᵂ_value)*cᵂ_value*L-L*y
```

```
1.0590227361717552e-8
```

```
cR(cᵂ_value , cᴿ_value)
```

```
0.567948706299423
```

```
println("Summary statistics:")
println("===================================================")
println("Retirement to working prices      💵   : $(@sprintf("%.4f",
(pR(cR(cᵂ_value, cᴿ_value ))/pW(cᵂ_value)))).")
println("Retirement to working cons.       🍎   : $(@sprintf("%.4f",
(cR(cᵂ_value,cᴿ_value)/cᵂ_value))).")
println("Retirement to working cons. exp.  💵🍎  : $(@sprintf("%.4f",
(pR(cR(cᵂ_value,cᴿ_value))*cR(cᵂ_value,cᴿ_value)/(pW(cᵂ_value)*cᵂ_value)))).")
println("===================================================")
```

```
Summary statistics:
===================================================
Retirement to working prices      💵   : 0.9244.
Retirement to working cons.       🍎   : 0.9794.
Retirement to working cons. exp.  💵🍎  : 0.9053.
===================================================
```

Labor search

McCall search

Simulation of agents' histories

Standard incomplete market economy in continuous time

Packages

The list of packages that I use:

- `Parameters` - required by: `@unpack`
- `Dates` - required by: \`Dates.now\`\`
- `LinearAlgebra`
- `Printf` - required by: @sprintf
- `Statistics` - required by statistical functions (`mean`, `var`, etc.)
- `Plots`
- `IterTools`
- `JLD2, FileIO`
- `StatsBase` - required by `countmap`
- `FreqTables`
- `SpecialFunctions` - required by `zeta`
- `DataFrames`
- `Pipe`
- `Optim`
- `Optim` - `converged`, `maximum`, `maximizer`, `minimizer`, `iterations`
- `Crayons`
- `FIGlet`
- `ProgressLogging`
- `SparseArrays`
- `Base.Threads`
- `InvertedIndices`
- `CSV`
