

3. Generic programming;

- [3. Generic programming;](#)
 - [1. Problem of Random Walk](#)
 - [2. Let's code it for one](#)
 - [3. Generalizing the function for 2D walker.](#)
 - [Exercises](#)
 - [1. Two walkers in 1D](#)
 - [Counting the number of times the walker passes a point](#)
 - [References](#)

In this chapter we will talk about Generic programming. The core idea is we want to avoid copying pasting code multiple times, and take functions as arguments of another function. Let's start with a simple problem of random walk.

1. Problem of Random Walk

Let's think about a problem called **random walk**. A random walk describes a **path** that consists of a succession of random steps on some mathematical spaces such as the integers.

An elementary example of a random walk is the random walk on the integer number line, \mathbb{Z} , which starts at 0 and at each step moves +1 or -1 with equal probability. So we can start thinking about a walker which will walk in this random path. Thinking initially about **only a single walker**, here is how could we **simulate** this. We will run following **algorithm**.

1. Initialise the walker

1. At each step:

1. move the walker

2. store the current position

2. Repeat from step 1

Now think about the problem of writing a generic code that might work regardless of in which dimension, the random walker walks. So the walker could walk in one dimensional line (1D), in a two-dimensional plane (2D), in a three-dimensional space (3D) and so on. So Ideally we would write our code in such a way that it has this structure but is more or less agnostic

(unaware) about *where*, the random walker walks, whether it is 1D, 2D, or even in a network, that does not matter. This is called **generic programming**. That is we will write code that will work in some generic context, and we avoid copying + pasting code, when the nature of the problem is same.

2. *Let's code it for one*

So a walker needs to initialize himself/herself and move on. So we can think about two functions, **initialize()** and **move()**, then applying these two functions he/she will get the new position and then walk again, and in this process we can save the trajectory! Here us how the function will look like!

```
# CodeBlock1.

"Simulate a walk for `T` steps."
function walk(T)
    pos = initialize()

    trajectory = [pos]    # make a Vector that contains just the current value of `pos`

    for t in 1:T
        new_pos = move(pos)
        push!(trajectory, new_pos)    # append to the Vector

        pos = new_pos    # update for next iteration
    end

    return trajectory
end
```

Note, we gave a "docstring" before the function for a good documentation!, this means now with Julia's help we can also see the details of the function.

```
# CodeBlock2.
?walk
```

Can we run the function and get the trajectory for **T** times? See the function and think maybe? Let's try,

```
# CodeBlock3.
walk(100)
```

You will get the following error

```
julia> walk(100)
# ERROR: UndefVarError: initialize not defined
# Stacktrace:
# [1] walk(::Int64) at ./REPL[13]:3
# [2] top-level scope at REPL[15]:1
```

What happened? Look at the error it says "initialize not defined". Of course both **initialize** and **move** are the functions we just made up, so in Julia there is no such function yet. So this means there's no problem to define a function which has unknown function inside, so in principle as long as the syntax is ok, we can define a function more or less in free way, it does not constrain to the scenario that we have to define everything before. However, when we will call it, this is going to be executed, but that is not possible because in this case the functions inside don't exist yet.

Now we will define the two functions. Let's think about a random walker moving in 1D. It starts at the origin (0) and jumps left or right with equal probability at each step:

```
# CodeBlock4.

# 1D walker

initialize() = 0 ## note if I define init = 0, this will be a variable,

random_jump() = rand( (-1, +1) ) # generate +1 or -1 randomly

move(x) = x + random_jump()'
```

here is the output,

```
julia> initialize() = 0
# initialize (generic function with 1 method)

julia> random_jump() = rand( (-1, +1) )
# random_jump (generic function with 1 method)

julia> move(x) = x + random_jump()
# move (generic function with 1 method)
```

👋 Questions:

- Why did we use tuple?
- It seems like the function **random_jump()** is redundant for the function **move()** in principle we could write **move(x) = x + rand((-1, +1))**. So what is the

benefit? Or more importantly is there any problem defining many small such functions, which will be combined at the end?

✂ Answers:

🔗 - Tuples are amazingly memory efficient. They do not allocate anything in the memory. In fact in this case they do not even produce an array, rather they generate some kind of expression (something like a string, we will see this later), which is called **Generators** and then evaluate and give us the result. Generators can produce values on demand, instead of allocating an array and storing them in advance. Let's have a look at some examples,

```
# CodeBlock5.

using BenchmarkTools

function tuple_check()
    for i = 1:100
        a = sum( (rand(Float64) for i = 1:10^2) )
    end
end

@btime tuple_check()

## Ok my PC crashed!

function array_check()
    for i = 1:100
        a = sum( [rand(Float64) for i = 1:10^2] )
    end
end

@btime array_check()

@btime begin
    (1, 2, 3)
end

# if you want to have the length of 100 tuple, but not the generator, you can always
# use the Tuple function!

Tuple( (rand(Float64) for i = 1:10^2) )
```

Here is the comparison using the **BenchmarkTools** package

```
julia> @btime tuple_check()
# 40.055 μs (0 allocations: 0 bytes)

julia> @btime array_check()
# 44.457 μs (100 allocations: 87.50 KiB)
```

So in principle we could use an array, but we intentionally did not to have a better performance.

```
# CodeBlock6.
rand([1, -1])
rand([-1, +1, 5])
```

- Now let's answer the other question, why did we use a seemingly redundant function. The reason is, for the small functions like this there is absolutely no harm because Julia compiler will do what is called **inlining**, roughly this means that, it will combine some functions in one-line and execute it something similar to a one line function. So you might think that writing too many functions might have some overhead issues, but in these cases it does not face this issue. So with small functions you should write as many functions as you need to make your clear, and this should not be any problem. We can also see this inlining in some cases, and the macro for this is **@code_typed** (which show just after the optimization is done in the compilation pipeline)

```
# CodeBlock7.
# recall the old functions
# random_jump() = rand((-1, +1)) # generate +1 or -1 randomly
# move(x) = x + random_jump()'

@code_typed move(2)
```

So inlining is a nice feature, this is a quote from Professor Sanders

It is idiomatic (i.e. regarded as good programming style) in Julia to define tiny functions like this that do one little thing. Note that there is no cost to doing so: Julia will inline such functions, removing the function-call overhead that you might have in other languages.

Having defined these functions, now we can call the **walk** function and get a trajectory

```
# CodeBlock8.
walk(100)
```

You will get following output

```
julia> walk(100)
# 101-element Array{Int64,1}:
#    0
#   -1
#   -2
#   -3
#   -4
#   -5
#   -4
#   -3
#   -4
#   ;
#  -11
#  -10
#   -9
#   -8
#   -9
#  -10
#   -9
#  -10
```

It will be a vector of dimension 101, and the very first element is the initialized position, then all the rests are the positions of the walker that moved on. So essentially the whole vector is a trajectory. We can also transpose the vector using `'`,

```
# CodeBlock9.
walk(100)' # "" means "transpose"; this is just to save space
```

Now let's plot the trajectory

```
# CodeBlock10.
using Plots ## you need to install Plots and then use it
using Random ## this package is pre-installed but you need to bring it
Random.seed!(6561); ## this will set the seed, and we can reproduce the same walk
plot(walk(100)) ## plot one random walk
plot!(walk(100)) ## plot another independent random walk, notice the bang will plot on top
```

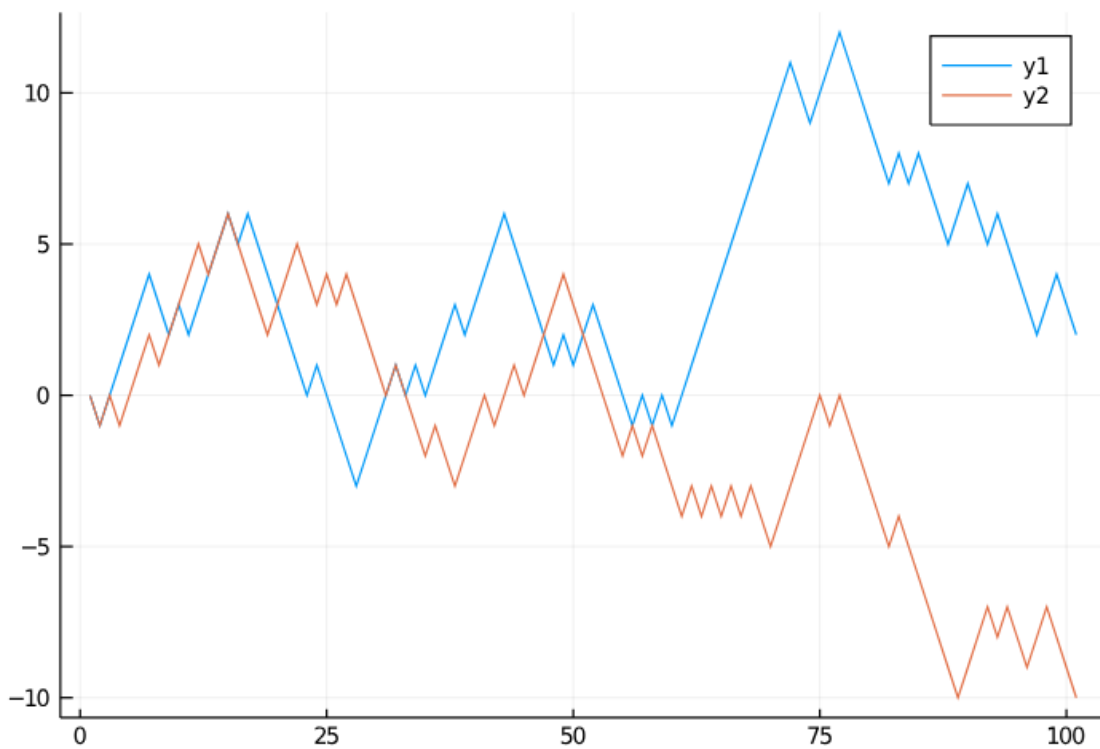


Figure: Random walks

We can also plot multiple random walks using a loop

```
# CodeBlock11.
p = plot(legend=false) # this will create an empty plot and turn off the legend/key

for i in 1:10
    plot!(walk(100)) # plot! adds to the existing plot (modify existing plot)end
end

## if you just finish here you will see nothing? Why it is printing nothing?
## This is because in julia forloop returns nothing!

# so we need to run this
p

# or we can also run this

plot!() # this will add nothing on the current plot
# p
```

3. Generalizing the function for 2D walker.

But now how could we generalize the function for a 2D walker? Recall our functions once again (just to recall)

```
# CodeBlock1. (from before).

"Simulate a walk for `T` steps."
function walk(T)
    pos = initialize()

    trajectory = [pos]    # make a Vector that contains just the current value of `pos`

    for t in 1:T
        new_pos = move(pos)
        push!(trajectory, new_pos)    # append to the Vector

        pos = new_pos    # update for next iteration
    end

    return trajectory
end

# CodeBlock4. (from before).
# 1D walker
initialize() = 0
random_jump() = rand( (-1, +1) )
move(x) = x + random_jump()
```

One thing we could do is copy the **walk** function and create a new function call it **walk2D** where inside we call the newly defined **initialize_2D** and **move_2D** functions for a 2D walker. For example, we can define

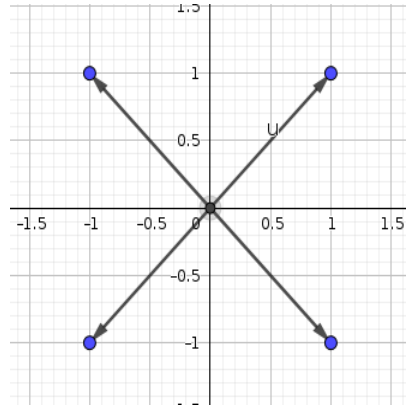
```
# CodeBlock12.
initialize_2D() = [0, 0]

move_2D(x) = [ x[1] + random_jump(), x[2] + random_jump() ]    # x is a vector
```

The last function will initialize a 2D point and then the move function will move from a point in a two-dimensional plane, e.g., (x_1, x_2) randomly. However, you can see that it cannot go to any arbitrary point in a 2D plane, if you start from $(0, 0)$, then you will see that the first position could be only one of the following four positions,

- $(+1, +1)$
- $(+1, -1)$
- $(-1, -1)$
- $(-1, +1)$

So it could be one of the four corners of a square of size 2. And then from any position it will again move to either one of these 4 corners.



Here is how we can check whether the initialize and move functions are working

```
# CodeBlock13.  
initialize_2D()  
move_2D([0, 0]);
```

Now we can copy+paste the last function and can define another function

```
# CodeBlock14.  
  
"Simulate a walk for `T` steps."  
function walk2D(T)  
    pos = initialize_2D()  
  
    trajectory = [pos]    # make a Vector that contains just the current value of `pos`  
  
    for t in 1:T  
        new_pos = move_2D(pos)  
        push!(trajectory, new_pos)    # append to the Vector  
  
        pos = new_pos    # update for next iteration  
    end  
  
    return trajectory  
end  
  
walk2D(100)
```

If you run this you will get following output

```
julia> walk2D(100)  
# 101-element Array{Array{Int64,1},1}:  
# [0, 0]
```

```
# [1, 1]
# [2, 0]
# [3, -1]
# [2, -2]
# [1, -3]
# [2, -4]
# [3, -5]
# [2, -4]
# [3, -5]
# :
# [18, 16]
# [17, 15]
# [18, 14]
# [17, 15]
# [16, 16]
# [17, 17]
# [16, 18]
# [17, 17]
# [16, 18]
```

But does this idea a good idea? In fact this is actually a **terrible** idea. If someone asks you now to create a 3D walker what would you do? Again copy+paste? Is it not possible to write a general function which will work in any dimension? Indeed we can do this. The key **idea** is we can *pass these 2D functions as arguments* to the **walk** function and then whatever dimension, the initialization and move takes place, our walk function will be able to handle that. Here is how we can do

```
# CodeBlock15.
function walk(initialize, move, T)
    pos = initialize()

    trajectory = [pos]

    for t in 1:T
        pos = move(pos)
        println(pos)           # *update* the value pointed to by `pos`
        push!(trajectory, pos)
    end

    return trajectory
end
```

If we run above command you will see the output **walk (generic function with 2 methods)**, why is that? Now actually we have methods for the same function. We can also check this using the **methods** function

```
# CodeBlock16.
methods(walk)
```

So what happened? We are passing the functions as arguments of another function. So how will I call this?

```
# CodeBlock 17,  
trajectory = walk(initialize_2D, move_2D, 10)
```

This is the output.

```
julia> trajectory = walk(initialize_2D, move_2D, 10)  
# 11-element Array{Array{Int64,1},1}:  
#  [0, 0]  
#  [1, -1]  
#  [0, 0]  
#  [-1, -1]  
#  [-2, -2]  
#  [-3, -1]  
#  [-4, -2]  
#  [-5, -3]  
#  [-4, -4]  
#  [-3, -3]  
#  [-2, -2]
```

so we have an array of arrays!

```
julia> trajectory[1]  
# 2-element Array{Int64,1}:  
#  0  
#  0
```

👋 Question: What if we redefine the old **initialize** and **move** function with the 2D version? Will we have two methods?

🚫 NO! because the argument is same, it will redefine the old function!

*For short vectors of fixed length like this, you should use **SVectors** from the [StaticArrays.jl](#) package to get much better performance. Try this out!*

Exercises

Try to solve the following exercises,

1. Two walkers in 1D

1. Suppose there are two walkers in 1D, one of which starts at position 0 and the other of which starts at position 2. How long does it take for them to meet each other, i.e. to land on the same location? Write a function **encounter** that runs *until* they meet each other and returns the number of steps.
2. Run this function many times and accumulate the resulting times in a vector.
3. Plot the histogram of the result. How does it behave?
4. How do these results change if the walkers live inside a box, e.g. from -20 to 20. At the ends of the box if they try to jump outside the box they "bounce back" and move one step back in the next step!

Counting the number of times the walker passes a point

Let's think about How many times does a walker w pass through the origin up to some final time? Now suppose we would like a random walker to have some "knowledge", for example, we want it to keep track of how many times it has passed through the origin. This is data that we somehow want to "associate to" the walker, which will need to be stored in a variable somewhere. The walker should be able to "remember" and modify this information.

In our above formulation it is not clear how to do this. We need to set up this variable to contain the initial position in the **initialize** function, and then update it in the **move** function, so this data needs to be shared between these two functions.

There are two common ways to deal with this:

- **Global variables**, which live outside any function and are updated inside each function. However, this means that the functions can no longer be understood just by looking at their definition, and it allows outside influences to modify the data.

Global variables make bad code that is hard to read and understand. Avoid them!

Actually there is a better solution, where we can make different walkers and refer to them individually.

References

1. David Sander's JuliaCon 2020 lecture title [Learn Julia via Epidemic Modeling](#)

There are some more details on counting, which I skipped, you can look at this if you are interested.