The aim of this workshop is to try out some basic programming. Many of the little tricks you will encounter can come in handy in different circumstances, as we have seen for the use of the for() loop.

Debugging

Writing code is just half, debugging the other half (if you are lucky). Find some lovely code here. Something wrong with it though.

1.Download the code and run it. First, reproduce the problem and read the error message. Quite obvious what is goes wrong, but where?

2.use print() to see if you can find in which function the error is.

3. traceback() provides a fast way to find in what function the error is situated. Similarly, browser() can be very useful to got through a function step by step. For the purpose of exercise add browser() to the first line of the directApplication() function at the top (above t <- testbase…)

If only all the bugs were so easy. Here are some wise words by Kernighan and Plauger: “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

The apply family

The different apply functions can come in very handy. They both take standard functions, like mean, but also custom ones.

1.Make a 10 by 20 matrix using matrix(). If you are unsure how to do this, use R (?matrix). Take the mean over the rows using apply(theMatrix, MARGIN, function). Do the same for each column. (MARGIN takes a 1 and 2 for columns or rows: or was it the other way around….. Reverse engineer it).

2. You can use your own custom function as well. Instead of mean use function(theMatrix) your code. Try this, e.g. take the square root of the median of each column.

3. Lists are very useful as you can put different character types in it. However, they can be notoriously difficult to work with. Let’s make a relatively simple list of different alleles for three genotypes.
Make three vectors of different lengths (e.g. one <- c(sample(1:20, number of samples))).
Turn them into a list. listName <- list(one, …..). Print the list to the screen. Unusual structure.

4.If you want each list element to have a name, add 'name = ' before each element (list(name1 = one, name2 = …..)). Do this and look at the how it looks like. More familiar. Use str(listName) to check the structure. Access the first element by:
listName$name1 or listName[1]

the first element of the first list element can be accessed by
listName$name1[1] or listName[[1]][2]

5.lapply() is very useful to apply a function to each list element. Try to take the mean.
Note that is always returns a list.
Try sapply on the same data set. Notice the difference? Quite convenient!

5. Ecological data often has multiple different for example, we have a two regions in which we sampled and different subsamples within these regions. How to get the mean of our measurements? Try tapply().
Make a data set.

```
stemHeight <- runif(40)
site <- factor(sample(c("a", "b", "c"), 40, TRUE))
region <- factor(sample(c("M", "F"), 40, TRUE))
```

Use tapply to calculate the mean stemHeight for each region, for each and for each site within region. If you are not sure what to do, use ?tapply.

Try for the MARGIN argument: only the name of the factor, c(factor) and list(factor). Do the same for both factors. c() is often used but obscures the factor levels.

The apply family is very useful and the above are only very basic applications. They can often be used instead of for loops, making the code more easy to read and less prone to bugs.

Species competition model
Species interact with one another and this affects the population dynamics over time. Species may compete over food resources, fight over territories or warn one-another when a predator is observed.
The Lotka-Volterra model for competition is one of the classics in ecology.

The population size in the next generation of species 1 ($n_1(t+1)$) is captured by

$$n_1(t+1) = n_1(t) + r_1 n_1(t)(1 - \frac{n_1(t) + \alpha_{12} n_2(t)}{K_1})$$

And species 2 by

$$n_2(t+1) = n_2(t) + r_2 n_2(t)(1 - \frac{n_2(t) + \alpha_{21} n_1(t)}{K_2})$$

The subscripts denote the two species.
The parameters:
r_i is the growth rate of the ith species.
α_{ij} is the competition coefficient which represents the competition exerted by the jth species on the ith species.
K_i is the carrying capacity of the ith species.

1. Before you start to implement the two discrete-time recursion equations think about how you want to structure the program. What are your variables of interest and what are the parameters? Which of the latter might you want to change? This is quite obvious in this case but you will need to add more parameters to run the simulation (e.g. initial population sizes).

2. Implement the formulas in R. Although the Greek letters and subscripts look very cool, let’s simplify this. Avoid the subscripts (e.g. n1) and write Greek letters in letters (e.g. alpha12). Decide which parameter you make ‘global’ (can be used by any function to write later on). Assigning a new value to a global parameter and refer to the variable name saves a lot of time if you want to change the variable later on as you only need to do this at one place.

3. Use a for() loop to execute the recursions equations multiple times (define at the start of the program how often!). Take care, as the population size at time t occurs in both formulas (so don’t update on of the population sizes before having calculating the other). Store the population size for each time steps, including the initial values. Plot the change of the

population size over time for both species. Simulate for longer periods if the population sizes still change.

4. You can run your entire r script in one go from your console by using source(“name_script.r”). Try it using the script you just wrote.

5. Play around with different starting values and carrying capacities. Does this change the equilibrium values attained? Keep the alphas the same.

6. The choice of competition coefficients represents different biological scenarios. What does both alphas negative and both positive represent*? One positive and one negative**? Predict how the different alphas would change the population dynamics of both species. Check your intuition by running simulations.

* mutualism and competition
** parasitic

Genetic drift
Genetic drift is a process, which changes allele frequencies in a population due to random sampling. Imagine a situation where a small part of a population of organisms arrives on an island. For sake of simplicity we assume these are clonally reproducing, haploid organisms. Initially two different alleles are present in the population: A and B. The aim is to write a simulation to see the initial allele frequencies change over time and how factors like population size and mutation affect this rate.

A couple of programming hints:
- Keep a version of your program which works in case changes fail.
- When testing a change, keep it as simple as possible (e.g. test a for loop for one repetition first).
- R is not the speediest code for some application. You can check the speed of a function or section of your program by embedding it in system.time(your function/code).
- Sometimes, if you can’t find a bug it is a good idea to clean the console, rm(list = ls()), and start again.

Setting up the simulation
Drift depends on population size. Let’s check this out.

1. Make an outline of the different components of the simulation. First the question has to be well defined and from this derive the variables of interest. What aspects of the model dynamics are we interested in (here the change of allele frequencies over time)? What parameters should we include in the model (e.g. population size)? In other words, what do we think affect the model outcome? If you want to change the parameters frequently, this is the right time to think how to implement them. A flow diagram can help you with this process.

2. Initiate a population of individuals of size N. A certain proportion (p) has A alleles and 1- p has B alleles (the resampling tips page might come in handy).

3.To get the next generation we should let them reproduce. How can we implement this? Assume that the population size remains constant.

4. Run each step (in this simple model this is only reproduce) for a number of generations. For each time step calculate and store allele frequencies. If you run many generations it is better not to stare the data every generation. You can use a counter and if() statement to save every x generations.

```
# parameters
# general
numberGenerations <- 10000          # number of generations the simulation
                                       # will run for

# data storage
counter <- 0            # keeps track when you want to store the data
saveData <- 100         # data will be written every 100th generation
dataStorage <- rep(NA, numberGenerations / writeData)
                              # vector to store output


# the main part of the program
for(i in 1: numberGenerations) {

    #insert code to update allele frequencies to get the next  #generation

    # part to check if we need to store the data
    counter <- counter + 1
    if(counter == writeData) {
            dataStorage[i / saveData] <- allele frequency
            counter <- 0        # reset the counter
    }
    # end of saving data
}
```

5. Make a plot of the change of allele frequencies over time. How long does it take before only one allele is fixated? Do you think this time varies between runs? Check this (for relatively small population size).

6. The first element in the dataStorage vector is the allele frequency after the first saveData generations. How would you change this, so that the first element is the initial starting value?

The number of generations is the same for each simulation. In this way it is hard to find the 'time till fixation' of an allele as you need to re-run a simulation for a longer period if it did not fixate. But this run will differ from the previous run! We will focus here on the rate of change of the allele frequencies over time.

7. Optional: Instead of for() use while(conditional statement) to let the population reproduce. This function will keep on going till you tell the condition is not met.
To test if it work let the condition, which needs to be met (e.g. a threshold difference in allele frequency between A and B) to be quite common so you don't wait for hours to stop the simulation. It might be useful to print the condition to the console to see if it works fine (maybe every 100 generations or so, similar to the saving the output file).
If you want to be able to repeat a simulation, use set.seed() at the start. Change this between replicated simulations though; otherwise they will be all the same (unless other parameters are changed)

Population size
So far we have focussed on one specific parameter setting. Let's change this. Investigate the effect of a range of population sizes.

1. Nest your for loop in another for loop so that our 'reproduce for x generations' (loop 2) is run for a different population size each time (loop 1).

```
#loop 1
for(){
        code to change population size
        popSize <- ….

        #loop 2
        for() {
                code for reproduction
        }
}
```

Note that the variable you assign in the first for loop to the variable popSize, can be read in the next for loop. *

2. Store each simulation in its unique row in a data frame. Plot the allele frequency against generation time for different population sizes.

3. Pick a population size for which the speed of the simulation is still fast. Now, change the starting frequencies of the alleles and keep the population size constant. Does this matter much?

4. Always check if your simulations work correctly. One way to do this is to investigate a situation of which you know the outcome. Equal allele frequencies for A and B is such situation. What are you expectations?**

Mutation

Mutation will change the allele frequencies and will thus affect the change of allele frequency change as investigated above. We assume the mutation is always reverses the allele (A -> B and B -> A). Let's implement mutation.

1.In favour of which allele will mutation work? Why?

2.Think of a way to implement mutation. A mutation rate is needed which subsequently should be translated to how many mutations we find in our population. Implement this in your program, save the simulations and plot the change of allele frequencies for different mutation rates. *** If you use the same code as above you can make a set if replicas (same N and starting frequencies) for with and without mutation.

3. Print the dataframe you use to your working directory (that is the standard directory to save to).****

Optional

Population size changes over time

The island our haploid creatures landed on is lush and the population can grow.

1.Implement linear population growth in your simulation.***** One of the easiest ways is to use the generation counter (i) from the for loop. Does population growth change the effects of drift?

2. Although things looked like heaven at first, it turns out there is a weather cycle spanning multiple generations which severely affects the population size on the island. It looks very much like a sine curve. Implement this and assess the effects.

* You can make a vector filled with population size at the start of the program (make it a global variable) and use the for loop as an index. Each new iteration of the for() loop will use a new population size.

** A and B should increase in equal proportion in the absence of selection.
*** Mutation rates are often assumption to be very small (1 e-5). Choose higher values to test the effect of mutation.
**** write.table(name, filename = “”, sep = “,”).
***** In case you did not notice, you can change the number of items resampled in the sample() function.