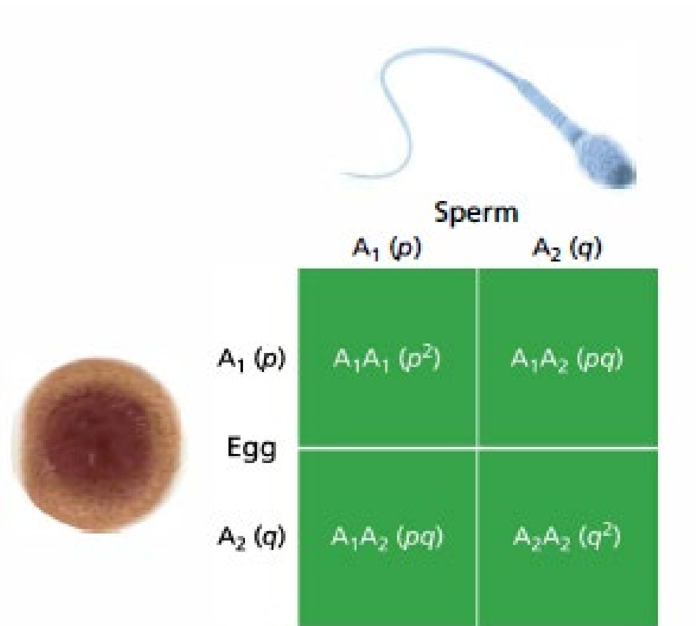
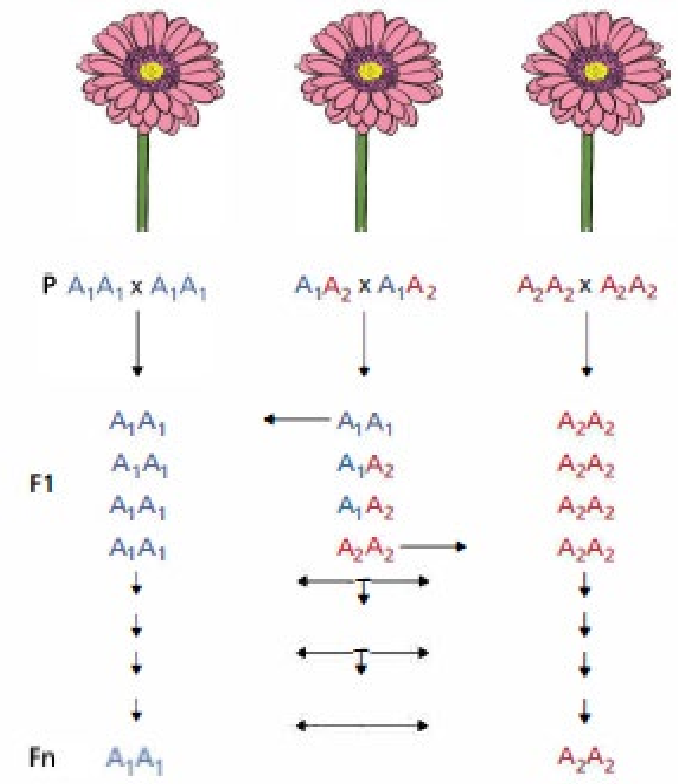
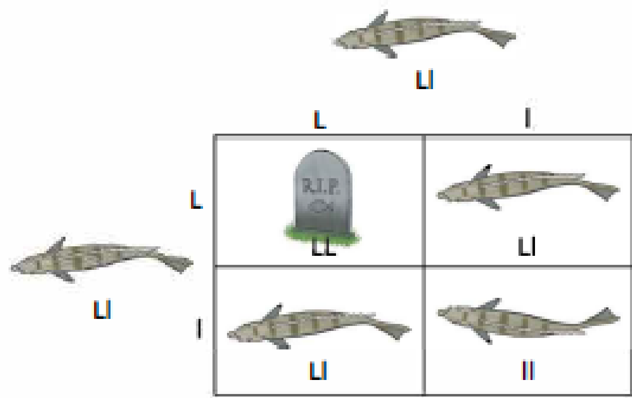
3. Allel ve genotip frekansındaki değişimler

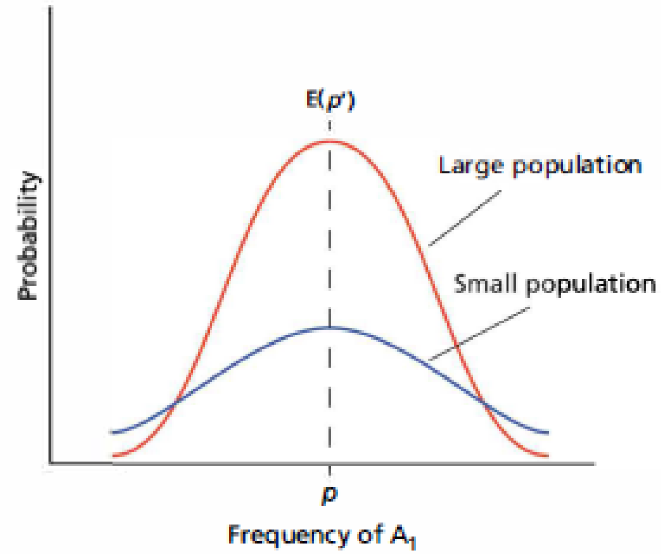
Teorik

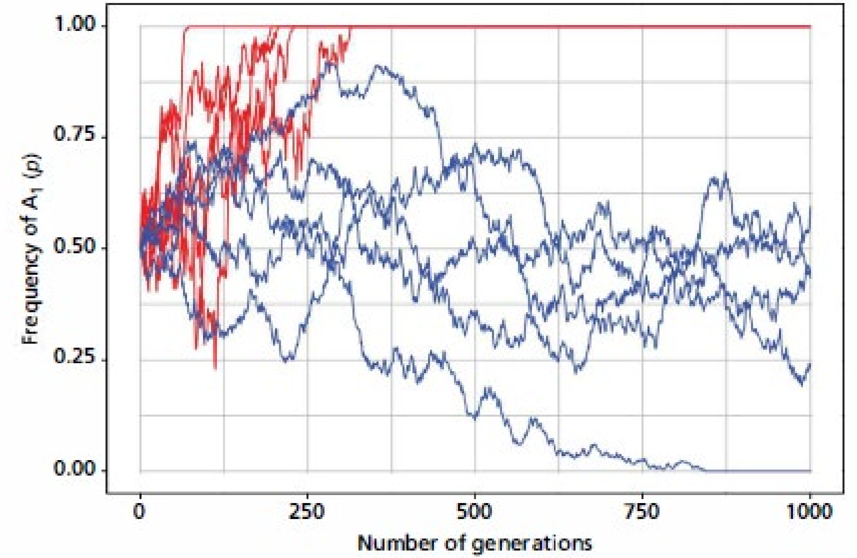




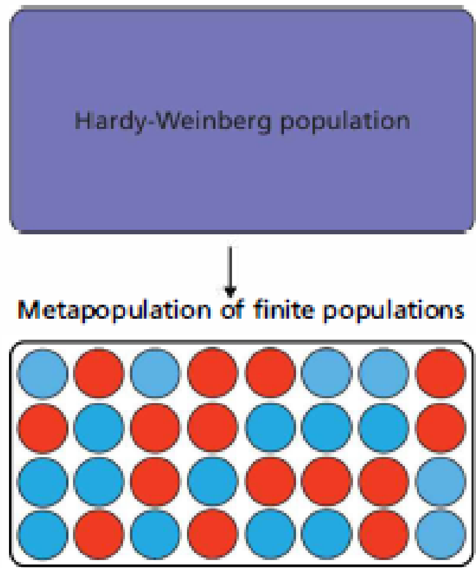
Figures and Tables were taken from Sætre, G. P., & Ravinet, M. (2019). *Evolutionary genetics: Concepts, analysis, and practice*. Oxford University Press, USA. For educational purpose only

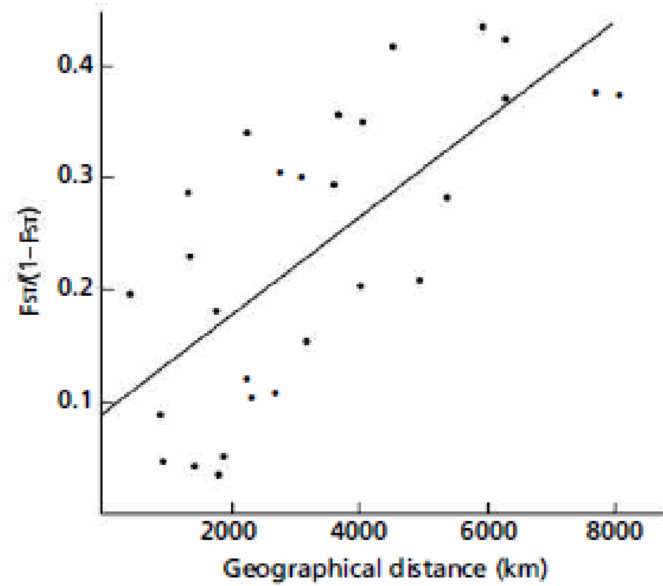


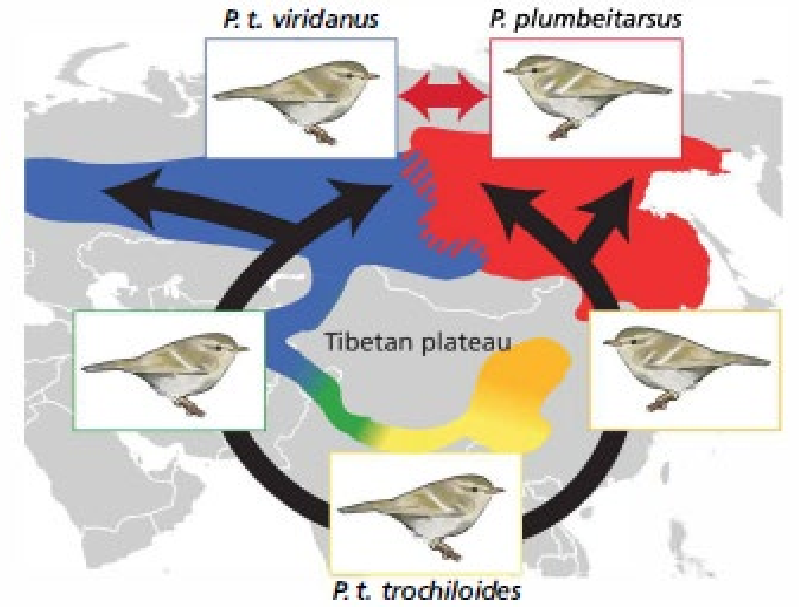




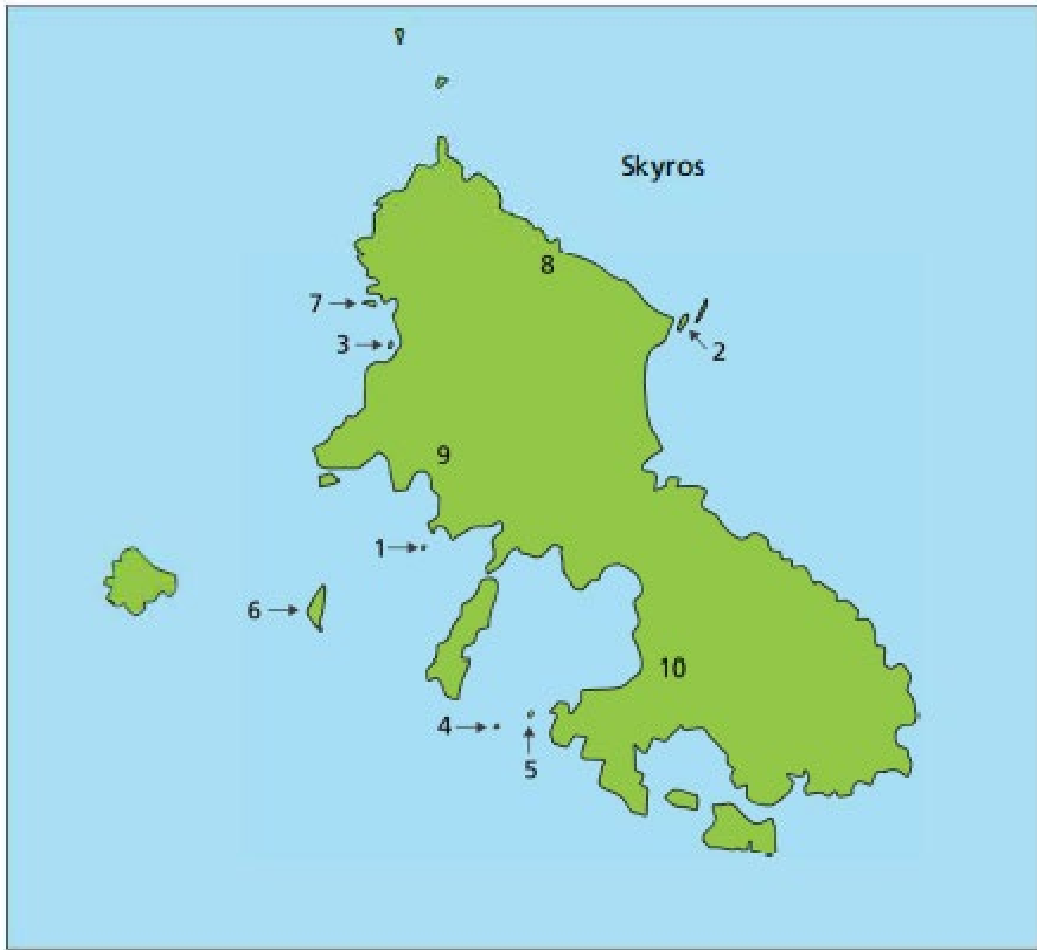
Figures and Tables were taken from Sætre, G. P., & Ravinet, M. (2019). *Evolutionary genetics: Concepts, analysis, and practice*. Oxford University Press, USA. For educational purpose only

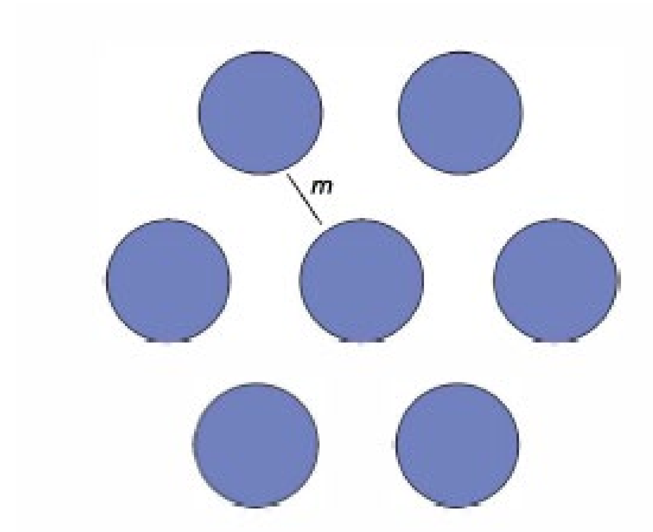


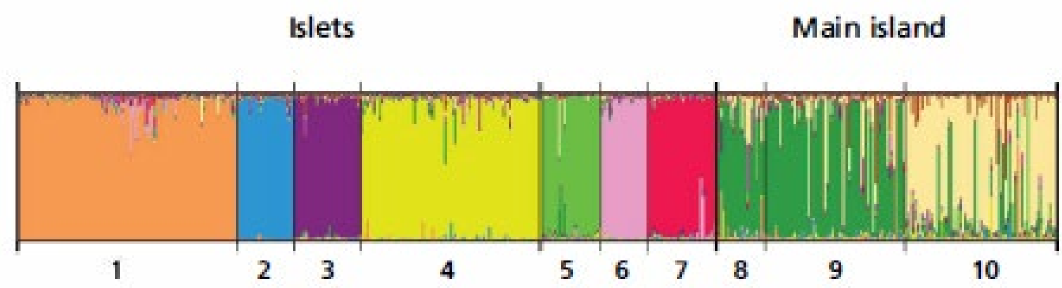




Figures and Tables were taken from Sætre, G. P., & Ravinet, M. (2019). *Evolutionary genetics: Concepts, analysis, and practice*. Oxford University Press, USA. For educational purpose only







Figures and Tables were taken from Sætre, G. P., & Ravinet, M. (2019). *Evolutionary genetics: Concepts, analysis, and practice*. Oxford University Press, USA. For educational purpose only

3. Allel ve genotip frekansındaki değişimler

Uygulama

The Hardy-Weinberg Model

The HW model is a way of demonstrating the relationship between allele and genotype frequencies. It is mathematically basic but very important for understanding how different processes can alter these frequencies. As you might recall from the main text, the model has five major assumptions:

1. All individuals in the population mate randomly.
2. The population is infinitely large.
3. No selection occurs.
4. No mutation occurs.
5. No gene flow occurs.

Imagining our idealized population, we will focus on a single locus, A which has two alleles, A_1 and A_2. We can use p to denote the frequency of the A_1 allele and q for A_2. This means that if we know p, we can derive q as $q = 1 - p$, since $p + q = 1$. Remember that the HW-model is a means of representing the relationship between allele and genotype frequencies across generations. With our assumptions outlined above in place, the only thing that determines the frequency of genotypes is their probabilities, derived from the allele frequencies.

It is quite simple to determine these probabilities. With 2 alleles A_1 and A_2, we have three possible genotypes and three ways to determine their probabilities as:

- $A_1A_1 = p^2$
- $A_1A_2 = 2pq$
- $A_2A_2 = q^2$

Another way to look at these probabilities is as the expected frequencies of the three genotypes at Hardy-Weinberg equilibrium given the allele frequencies. Let's use some R code to actually demonstrate how we might calculate these expected frequencies. For our test example, we will use a simple case where the frequency of A_1, p is 0.8 and the frequency of A_2, q is 0.2.

```
# first we set the frequencies
p <- 0.8
q <- 1 - p
# check p and q are equal to 1
(q + p) == 1
# calculate the expected genotype frequencies (_e denotes expected)
A1A1_e <- p^2
A1A2_e <- 2 * (p * q)
A2A2_e <- q^2
# show the allele frequencies in the console
c(A1A1_e, A1A2_e, A2A2_e)
# since these are genotype frequencies, they should also sum to 1 - you can check this like s
o
sum(c(A1A1_e, A1A2_e, A2A2_e))
```

So you can see it is quite easy to calculate the frequencies you need for the HW model in R. If you want to, you can play around with the initial frequencies defined at the start of the R code to see how it alters the genotype frequencies. This is one nice thing about using R for evolutionary genetics, it is pretty simple to demonstrate things mathematically to benefit your own understanding. However, it might be even more useful to demonstrate the relationships the HW model lays out graphically. That way you can see how the expected genotype frequencies are altered over a whole range of allele frequencies. Luckily for us, R is great for this sort of visualisation *and* this is an excellent opportunity to demonstrate some programming in R.

Plotting the expected genotype frequencies

The first thing we need to do is generate a range of allele frequencies to visualize the expected genotype frequencies across. We'll do this in the simplest way first, before going into how you might take a programming approach to the same problem. Either way, generating a range is simple - we just need to do it for a single allele, A_1 and then we can very easily derive the frequency of A_2. We will do this like so:

```
# generate a range for p
p <- seq(0, 1, 0.01)
# and also for q
q <- 1 - p
```

All we did here was use the `seq` function, which we first saw back in Chapter 1 (https://evolutionarygenetics.github.io/Introduction.html). Remember, we only have to do it once because q is the inverse of p. Next we need to generate the expected frequencies. Because R is really great at handling vectors, the code for this is identical for if we were using a single value.

```
# generate the expected genotype frequencies
A1A1_e <- p^2
A1A2_e <- 2 * (p * q)
A2A2_e <- q^2
```

So now we basically have the expected frequencies at HWE of our different genotypes across a range of allele frequencies with just a little vector calculation. However, recalling from before that we often need to get our data into a `data.frame` in order to use `ggplot2`, we need to do a bit of work on this first. We'll now get our data into a `tibble` ready for working on…

```
# arrange allele frequencies into a tibble/data.frame
geno_freq <- as.tibble(cbind(p, q, A1A1_e, A1A2_e, A2A2_e))
```

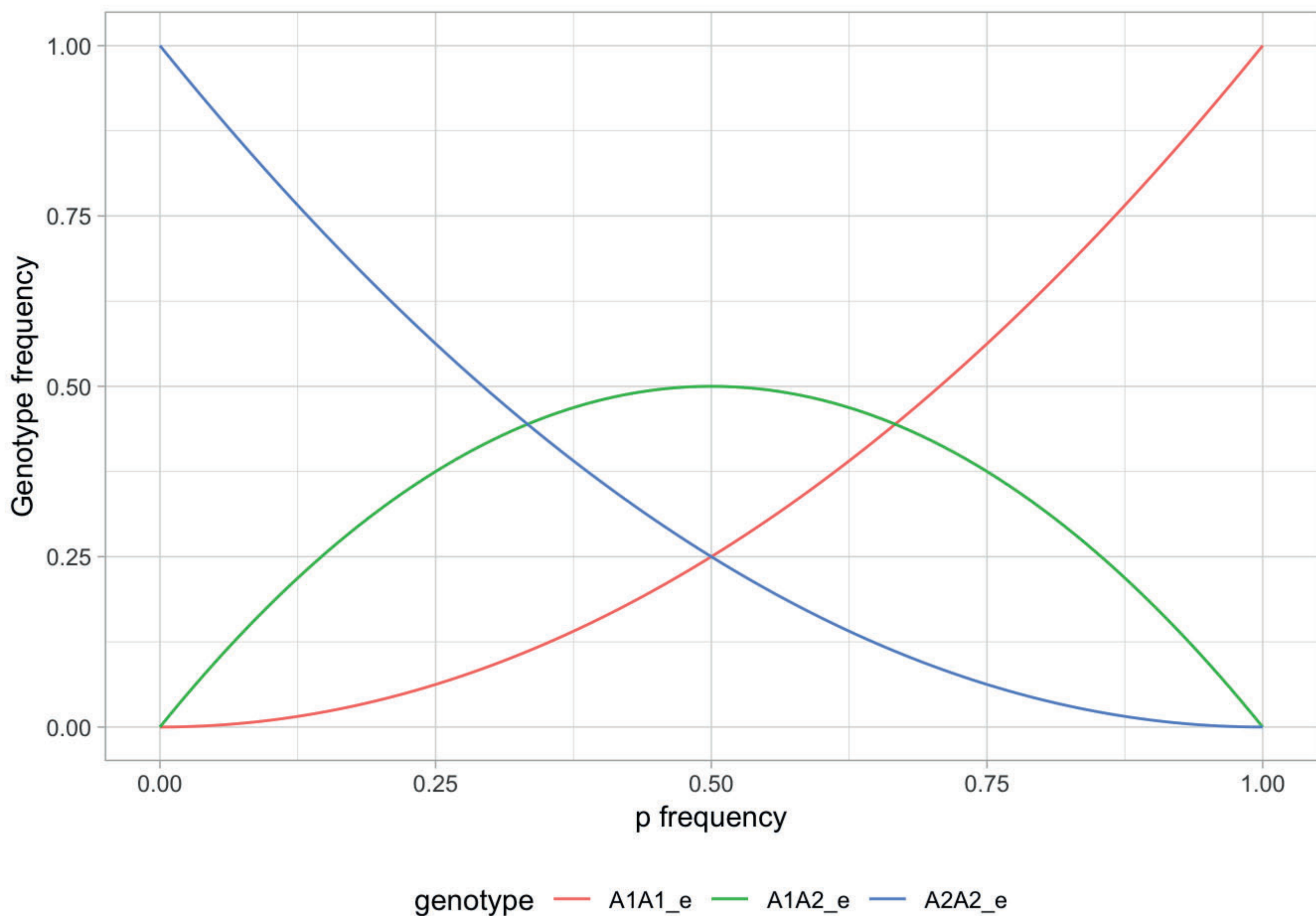
```
## Warning: `as.tibble()` is deprecated, use `as_tibble()` (but mind the new semantics).
## This warning is displayed once per session.
```

You also might remember from the previous session that we need to sometimes do some data manipulation to get the data into a form that we can easily plot. Luckily, this is quite straightforward in this case as we can use the `gather` function to do it quickly and easily.

```
# Use gather to reshape the data.frame for straightforward plotting
geno_freq <- gather(geno_freq, key = "genotype", value = "freq", -p, -q)
```

Now it is really straightforward to make a plot that shows the expected HW—frequencies across our allele frequencies.

```
# plot the expected genotype frequencies
a <- ggplot(geno_freq, aes(p, freq, colour = genotype)) + geom_line()
a <-a + ylab("Genotype frequency") + xlab("p frequency")
a + theme_light() + theme(legend.position = "bottom")
```



Some programming tricks for the HW model

The example above, where we derived the expected frequencies of the HW-model took advantage of R's ability to easily handle vectors. In principle, this is the simplest way to achieve what we wanted and it was quick and clear to code. However, there are actually several different ways we could have approached this issue using some programming tricks. We will introduce these briefly here as a demonstration but these are topics we will return to throughout these tutorials. First up, we will delve into using a `for` loop.

For whom the bell tolls

What if we want to calculate the genotype frequencies on the fly? So rather than using 4-5 separate lines of code to generate the range of allele frequencies and expected frequencies, we could do it all in one short code block? One way to do this might be to use a `for` loop. To imagine what a `for` loop does, it is essentially a block of code that is repeated for different values. In plain language, we are saying for each value of x, do y. This might actually be a little confusing the first time you encounter it but a simple example might help so before we run an example with the allele frequencies, we will show the simplest `for` loop we can.

```
for(x in 1:10){
  print(x)
}
```

What happened here? Firstly you will see `for` is highlighted a bit differently to a normal function - this is because it is a **control flow** statement - i.e. reserved for programming in the R environment. The standard structure of a `for` statment in R is `for(variable in vector)` - in the example above, we wrote `for(x in 1:10)` . All this means is that we are cycling through `1:10` and for each value of this (i.e. 1, 2, 3 and so on), we will call it `x` and then do something to it. The curly brackets that follow our `for` statement are what we will then 'do' to our value of `x` . In this case, we simply `print` x to the screen.

We can do much more than this though - what if we want to add 10 to each value of x?

```
for(x in 1:10){
  print(x + 10)
}
```

Note that in this simple example, we need to `print` x each time. We might not want to print each value of `x`, instead we might just assign it to a vector. So, let's return to our genotype frequency example. We will write out the full `for` loop and then break it down in to more detail.

```
# set up p
p_vec <- seq(0, 1, 0.01)
# initiate null vectors for the expected frequencies
A1A1_e <-  NULL
A1A2_e <- NULL
A2A2_e <- NULL

# run the for loop
for(p in p_vec){
  # define q
  q <- 1 - p
  # calculate genotype frequencies
  A1A1_e <- append(A1A1_e, p^2)
  A1A2_e <- append(A1A2_e, 2 * (p * q))
  A2A2_e <- append(A2A2_e, q^2)
}

# combine them into a matrix (as an example here)
cbind(A1A1_e, A1A2_e, A2A2_e)
```

This block of code is a bit trickier than the simple examples of `for` loops we showed previously but the general principle is the same. The main difference here is that we first had to declare `NULL` variables so that we could save the output of our loop. For each value of p, our loop first calculates q, then it calculates all the genotype frequencies. You will see that it also uses the `append` function to save each calculation to the null variables we initiated before running the loop. Note that we also demonstrated here that you can change what you will call the variable in a `for` loop. For example, here we changed it to `p`, whereas our original example used `x`.

By now you might be reasonably looking at this code and thinking - well surely this is much more complicated than the 4-5 lines we wrote previously? And you'd be absolutely right! Although we used this as a demonstration of how to use a `for` loop, it is also a neat demonstration of the fact that you don't have to overcomplicate things - sometimes the simplest code is best.

Vectorisation

Actually there is another lesson we can learn here - `for` loops in R are often not optimal. They can be slow when applied to very large amounts of data for a start and also they are often unnecessarily complicated - as we saw above. In these cases, if we do want to use a programming approach it is much better to use **vectorisation**. This is generally a much faster and more lightweight approach than control flow statements. To demonstrate the power of vectorisation, we will use the `sapply` function. However, it can seem a bit counterintuitive the first time you encounter it. As with the `for` loop, we will give simple examples first.

```
sapply(1:10, function(x) x)
```

Firstly, this is identical to the first for loop we showed you. Let's break it down. `sapply` stands for 'simple apply' and it does exactly that. You give it a vector, `1:10` in this case and then it does something to each value of that vector. The `function(x)` statement basically just says we are going to call each element of our vector `x`.

After this, we can actually write what we want to do `x`; since here all we wrote is `x`, we are just printing it to the screen. What if we want to add 10 to each value of our vector, like with our second `for` loop example?

```
sapply(1:10, function(x) x + 10)
```

Pretty simple but not hugely different right? Well it is when we ramp up the complexity that you will start to see the difference. Let’s try rewriting the complex `for` loop as a vectorised operation here. We’ll introduce it and then break it down.

```
# set up p
p_vec <- seq(0, 1, 0.01)

geno_freq <- sapply(p_vec, function(p){
  # define q
  q <- 1 - p
  # calculate probabilities
  A1A1_e <- p^2
  A1A2_e <- 2 * (p * q)
  A2A2_e <- q^2
  # write out
  c(A1A1_e, A1A2_e, A2A2_e)
})

# transpose the matrix to get it the right way up!
geno_freq <- t(geno_freq)
```

OK, so that is still more code than with the simple vector calculation approach. However, this hopefully demonstrates how a vectorised approach provides a simpler solution to a `for` loop. If the length of `p_vec` was longer too, the vectorised approach would be much faster. The main thing here is that the code for the vectorised approach is identical to that when we did the vector calculations, except it is just wrapped in an `sapply` function. You’ll also notice the `{}` brackets are in this use of `sapply`. They are not needed when we do a simple, one line `sapply` call, but here with a more complicated function, we do need them to make things easier to read. One other point to note is that again we changed `x` to `p` in the `function(p)` statement - demonstrating once more that you can call your variables inside your vectorised functions whatever you’d like. Finally the `t` function is just a simple call to transpose the matrix that `sapply` outputs. You can try running the code without this and look at what `geno_freq` looks like - you’ll see it is orientated oddly if we want to use it in plotting.

A not so final word on vectorisation and for loops

In a moment, we’ll jump back into the evolutionary genetics again. However we just want to reiterate that the aim here is to introduce you to these concepts. It takes a lot of practice to become familiar with loops and vectorisation. They can both be very confusing when you start out with them. Personally, we try to avoid `for` statements where possible - vectorisation is usually always superior (although not always)! We will certainly return to these techniques - the aim here was to give you a taster.

Testing for deviations from the Hardy-Weinberg Expectation

We’ve already learned that the HW model is essentially an idealised scenario where no demographic or evolutionary processes are shaping the relationship between allele and genotype frequencies. We can therefore use it as a null model against which to test our data. Basically, we can compare our observed genotype frequencies from those expected under HWE. To get the expected frequencies, we just generate them from the allele frequencies, as we did in the previous section of the tutorial.

We can work through an example of this together. Once again we'll assume a locus A with two alleles, A_1 and A_2. This means three genotypes, A_1A_1, A_1A_2 and A_2A_2. We sample 150 individuals from a population. The next block of R code shows the numbers of each genotype we collected and also combines them into an `observed` vector - for later use.

```
# numbers of genotypes
A1A1 <- 80
A1A2 <- 15
A2A2 <- 55
# make into a vector
observed <- c(A1A1, A1A2, A2A2)
```

In order to go further here, we need to work out the allele frequencies from our observed data. This is quite straightforward - homozygotes have two copies of an allele whereas heterozygotes have only one. So we derive the number of each allele and divide it by the total:

- $p = \frac{2(A_1A_1) + A_1A_2}{n}$
- $q = \frac{2(A_2A_2) + A_1A_2}{n}$

Where the genotype notation here represents the number of genotype class and n is the total number of alleles. We can calculate the frequency the alleles in R like so:

```
# calculate total number of alleles
n <- 2*sum(observed)
# calculate frequency of A1 or p
p <- (2*(A1A1) + A1A2)/n
# calculate frequency of A2 or q
q <- (2*(A2A2) + A1A2)/n
# print frequencies to screen
c(p, q)
```

Note that with the code above, we summed the number of observed genotypes and then multiplied by 2 as there are 2 alleles for each individual (i.e. there are 150 individuals and 300 alleles). We now have the allele frequencies, so we can use the code we worked out in the previous section to calculate the expected Hardy-Weinberg frequencies for this population.

```
# generate the expected genotype frequencies
A1A1_e <- p^2
A1A2_e <- 2 * (p * q)
A2A2_e <- q^2
# combine into a vector
expected_freq <- c(A1A1_e, A1A2_e, A2A2_e)
```

So now that we have the **expected** genotype frequencies, the last thing we need to do before testing whether there is a deviation from HWE in our data is calculate the **expected number** of each genotype. This is easy - we just multiply the frequencies by the number of individuals we sampled - 150.

```
# calculate observed genotype frequencies
expected <- expected_freq * 150
```

The next thing we'd like to do is see whether our observed genotypes deviate from the expected. We will test this formally, but first let's just look at the differences between them. The simplest way to do this is combine them into a matrix:

```
# combine observed and expected frequencies into a matrix
mydata <- cbind(observed, expected)
# add rownames
rownames(mydata) <- c("A1A1", "A1A2", "A2A2")
```

Well, just from eyeballing this matrix, it does look like there is probably a difference. Notably there seems to be a lower frequency of heterozygotes than we expect. Still, just looking at this is not enough, we need to test it formally. As you will probably remember from the main text, we can use a **Chi-squared goodness of fit test** for this purpose. The Chi-squared test is quite straightforward:

$$X^2 = \sum_{i=1}^{k} \frac{(E_i - O_i)^2}{E_i}$$

In other words, this is the sum of the squared differences between the expected and the observed, divided by the expected. It sounds complicated but with R's vector handling capabilities, we can do this extremely easily:

```
chi <- sum(((expected - observed)^2)/expected)
```

Our chi-squared value is very high. But we still don't know if it is significant. To do this we need to check our value against the known distribution of chi-squared. This is very easy to do in R using the `pchisq` function. This is part of a suite of functions for probability distributions which we will explore more thoroughly in a future tutorial. But for now, let's work out whether our value is significant.

```
1 - pchisq(chi, df = 2)
```

`pchisq` takes two arguments, the chi-squared value and the degrees of freedom. For our chi-squared test here, we use 2 degrees of freedom. You can see that our p-value is 0 - this means we have a highly significant deviation from HWE - in other words, some assumption of the Hardy-Weinberg model is violated. What process could cause this significant heterozygote deficit? Maybe non-random mating?

An easier way?

Since the chi-squared test is such a fundamental statistical tool, it probably won't surprise you to find that there is actually already a `chisq.test` function for you to use. We didn't explain it earlier because… well calculating things by hand is sometimes the best way to learn! The function is handy though since we can calculate the chi-squared value and also test the significance in one go. We can use it like so:

```
# perform a chi-squared test
mychi <- chisq.test(observed, p = expected_freq)
```

Here we provide the `chisq.test` function with our observed genotype counts. However, we also need to specify our expected frequencies, so there is still some calculation required. Nonetheless, the function returns the same chi-squared value we have previously calculated and shows us that we have a highly significant deviation from the expectation.

Note as well that the `mychi` object has a lot of information stored within it. We can take a closer look using the following:

```
str(mychi)
```

Using the dollar sign notation we learned about in Chapter 1 (https://evolutionarygenetics.github.io/Introduction.html), we can extract any of these at will. For example `mychi$expected` will produce the expected values whereas `mychi$statistic` will produce the chi-squared

value.

Simulating genetic drift

One of the main assumptions of the HW model is that populations are infinite. This is obviously unrealistic! All populations are finite - but what does this mean for how evolution unfolds? An important concept in evolutionary genetics is how population size can influence the relative power of a process such as **genetic drift**. Drift essentially describes the random component of survival in a population - some individuals will survive and reproduce more than others by chance. An important factor contributing to such genetic drift is the random sampling of alleles that takes place at fertilization. For instance, just like some pairs produce an excess of sons and daughters, a heterozygous mating pair (A_1A_2) can produce offspring with an excess of the genotypes A_1A_1 (or A_2A_2) thereby causing a random change in allele frequency in the population. When a population is large, the effects of drift will be small relative to other processes such as selection. However, when a population is small, drift as a result of random chance has a much bigger effect.

One of the best ways to get your head around a concept like genetic drift and how it interacts with population size is to actually simulate, which is exactly what we are going to do here. We'll first explain how we can simulate it over a single generation and then we are going to build our own **custom R function** to do all the leg work for us. So we'll not only be exercising your understanding in evolutionary genetics, but also your R programming chops too.

A simple case of genetic drift over a single generation using the binomial distribution

To demonstrate a simple case of drift, we'll use the example from the book. We have a population of N = 8 individuals and we will focus on a single locus A with two alleles, A_1 and A_2. We will assume both alleles are equally common in the population - i.e. $p = q$ = 0.5. Our little population is in HWE such that there are two of each A_1A_1 and A_2A_2 homozygotes and four A_1A_2 heterozygotes.

To simulate drift, we need to allow individuals to mate at random. In the book you will recall we used a coin-flipping experiment to sample our alleles to form genotypes, since both alleles have a frequency of 0.5. This is essentially a form of **binomial sampling**. We can easily recreate this in R using the `rbinom` function to sample the binomial distribution. We will recreate the coin-tossing experiment exactly here, using the following code:

```
# recreate coin flipping experiment using binomial sampling
rbinom(n = 16, size = 1, prob = 0.5)
```

What did we do here? Like with the `pchisq` function we used before, we made use of R's suite of functions for statistical distributions (a topic we will cover in more detail later). The `rbinom` function just allows us to sample the binomial distribution. With the `n = 16` flag, we sampled it 16 times. We set the `size` to 1 - meaning we sample either 0 or 1 (i.e. heads or tails) and finally we set the probability of sampling to the same for both outcomes with `p = 0.5`.

However, in the textbook we actually used pairs of coin flips to simulate genotypes. We could do this in R, but it is a bit clunky (it is much easier to just flip a coin!) so we could actually use binomial sampling in a different way to achieve this. We could set the size of the population as `N` and then just sample one value from the distribution. This value could then represent the number of A_1 alleles in the next generation - so we could easily work out $p\acute{}$ and $q\acute{}$. We'll also take this opportunity to set the value of p here.

```
# set population size
N <- 8
# set allele frequency
p <- 0.5
# recreate coin flipping experiment using binomial sampling
nA1 <- rbinom(1, size = 2*N, prob = p)
```

Remember since we are sampling alleles for diploid individuals, we need to multiply `N` by 2. The value of `nA1` will differ for everyone since this is a **random** sample of the binomial distribution. We can now easily work out the value of $p\acute{}$ - i.e. the new frequency of the A_1 in the next generation. Note that R doesn't allow the prime notation so for the actual R code, we will use `pa`.

```
# calculate p´
pa <- nA1/(2*N)
```

OK so now we have a value for `pa` - again it will differ for everyone because we performed random sampling. You will see though that `pa` is likely different to `p` solely because of random sampling - i.e. we have demonstrated drift over a single generation. To properly understand drift over multiple generations, we will need to use some quite advanced R code - so for the next section, we will take a break from population genetics and instead focus on **how to write a custom R function**.

Writing your own R functions

Writing functions in R is an extremely useful thing to learn. It might seem a little daunting but it is actually quite straightforward once you have a handle on the basic format and structure necessary. Functions can be useful for cleaning up your code - instead of writing the same code block again and again, you can use your own function! You also might want to do it to then wrap it in a more complicated operation - like a `for` loop or a vectorised approach. We will actually build towards something like the latter, but for now we will just use a simple example. Let's write a function that adds 10 to any numerical value we give it.

```
# write a custome function to add 10 to every numerical variable it is given
add_ten <- function(x){
  y <- x + 10
  return(y)
}
# test it out
add_ten(100)
add_ten(1000000)
add_ten(22)
```

Let's breakdown what we did here in a bit more detail. Firstly, we need to give our function a name - in this case we called it `add_ten`. Crucially, this is also an object we assign. Once assigned, the function is callable in the R environment, just like any other function we load from a package or in the standard R distribution. We then use the statement `function(x)` to show it is a function. The `x` here is the name of the variable we pass to our function. The name here is completely arbitrary; we could just as easily use a human name or anything we choose. However, what we pass to the function will be referred to this throughout.

Inside the curly brackets - `{}` is the body of our function. Here we define a new object `y` that is simply the value of `x` plus `10`. Finally, we use the `return()` function to tell R to produce the value of `y` when we use our `add_ten` function. We can really write a function for almost anything in R and it wouldn't take us long to fill this tutorial with example after example. For now though, we will show you just one more example - a function that takes two arguments, instead of just one like our `add_ten` function did. This time, we will write a function to multiply two numerical values.

```
# write a custome function to multiply two numerical variables
multiply <- function(x, y){
  z <- x * y
  return(z)
}
# test it out
multiply(10, 10)
multiply(3, 5)
multiply(22, 44)
```

So this is exactly the same as before, but now we are passing two arguments - `x` and `y`. In the body of the function, we multiply them both and then return the answer. Hopefully this demonstration of how to write functions is a sufficient example. As with our previous programming topics, it is something we will return to throughout the course, building on it further and further. Again, we certainly don't expect you to go from these examples to writing complex functions!

Using a custom function to simulate drift over multiple generations

Recalling our code for simulating genetic drift, let's do the same thing again but this time wrap it in a simple function. To start with we do this only for a single generation.

```
# a custom function for simulating drift in a single generation
drift_sim <- function(N, p){
    pA <- rbinom(1, 2*N, p)
    p <- pA/(2*N)
  return(p)
}
# run drift sim on the example data above
drift_sim(N = 16, p = 0.5)
```

Now we have the `drift_sim` function, we can easily play around with the allele starting frequency(i.e. `p`) and also the population size. But we still haven't managed to extend our simulations across multiple generations. Now we have the basic function, we can alter it to do just that:

```
# a custom function for simulating drift across multiple generations
drift_sim <- function(N, p, ngen){
  # initialise p
  p_init <- p
  # sample across all the generations
  pvec <- sapply(1:ngen, function(x){
    pA <- rbinom(1, 2*N, p)
    p <<- pA/(2*N)
  })
  # create a vector of p over time
  p <- c(p_init, pvec)
  # write out
  return(p)
}
# run drift sim on the example data above
drift_sim(N = 16, p = 0.5, ngen = 100)
```

OK so this will require a little explaining! Firstly you will notice we have added the `ngen` argument - which is the number of generations. Inside the function, we need to define the initial value of p - i.e. before the simulations begin - `pinit`. We then use an `sapply` call to use run our code over each generation. Each time we do this, we update the value of p - this explains the double arrow `<<-` we used. We have to do this updating because otherwise the value of `p` will also be reset to the initial value, which is not suitable when simulating drift over time. The final part of the function outputs our vector `p` and includes the original, initiating value first.

Now that we have a function to simulate data, across generations, we can easily do this for different population sizes. Let's simulate 1000 generations of drift for three populations of 10, 100 and 1000 individuals. We'll keep the initial allele frequency, `p` at 0.5 in all cases. We will then combine them all into a `tibble` alongside the number of generations

```
# simulate drift for three different population sizes
n10 <- drift_sim(N = 10, p = 0.5, ngen = 1000)
n100 <- drift_sim(N = 100, p = 0.5, ngen = 1000)
n1000 <- drift_sim(N = 1000, p = 0.5, ngen = 1000)
# get number of generations
g <- seq(0, 1000, 1)
# combine into a tibble
mydrift <- as.tibble(cbind(g, n10, n100, n1000))
```

Note that we could also have run `drift_sim` in a single line of code using `sapply` like so:

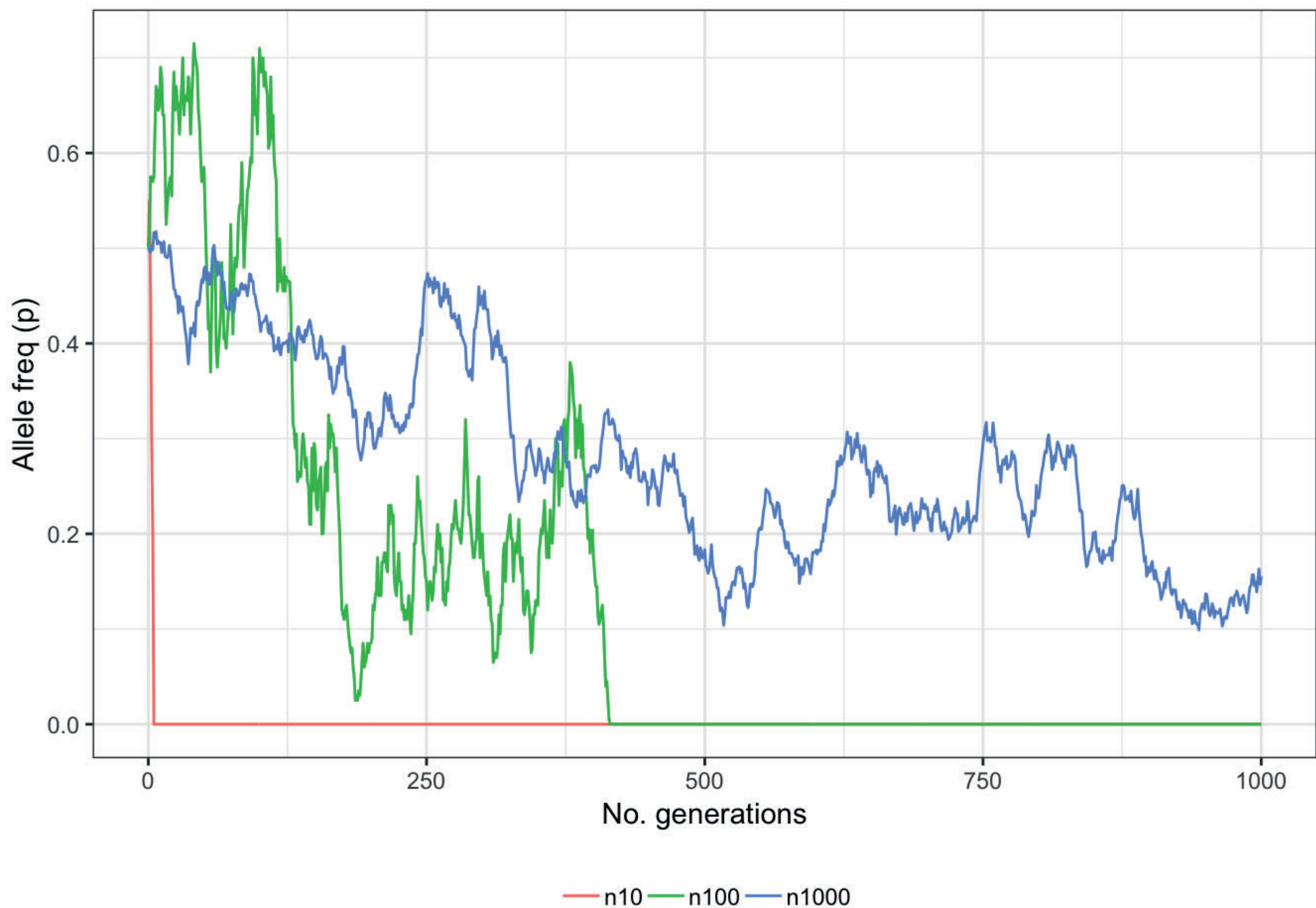
```
# simulate drift for three different population sizes
a <- sapply(c(10, 100, 1000), drift_sim, p = 0.5, ngen = 1000)
# rename matrix
colnames(a) <- c("n10", "n100", "n1000")
# get number of generations
g <- seq(0, 1000, 1)
# combine into a tibble
mydrift <- as.tibble(cbind(g, a))
```

Both ways work, but this is a nice demonstration of how you can use something like `sapply` to make your code more succinct. Anyway, what we ultimately want to do is *visualise* drift over time. So we need to get our data in a position to plot it using `ggplot2` - once again, we turn to `gather`.

```
# use gather to get data ready for plotting
mydrift <- gather(mydrift, key = "pop_size", value = "p", -g)
```

All we did here was collapse the population sizes into a single factor for each generation. `gather` commands can take a while to get the hang of, the best thing is to continually practice them like this! Now we will plot our data using the following code:

```
# plot data
p <- ggplot(mydrift, aes(g, p, colour = pop_size))
p <- p + geom_line()
p <- p + xlab("No. generations") + ylab("Allele freq (p)")
p + theme_bw() + theme(legend.position = "bottom", legend.title = element_blank())
```



And there we have it! A plot of our simulated data showing allele frequency changes over time as a result of genetic drift. Although the plot you get will differ from the one you see here (because of the random sampling), what you should be able to see is that with a low population size alleles are either lost or go to fixation very rapidly. This is because the effect of drift is much stronger here, whereas in larger populations it may take a long time for drift to fix or remove an allele - indeed in some cases it will not happen at all.

Study questions

For study questions on this tutorial, download the `chapter3_R_questions.R` from Canvas or find it here (https://evolutionarygenetics.github.io/chapter3_R_questions.R).

Going further

There are a lot of great resources available on programming in R and evolutionary genetics. There are also some good further tutorials on understanding HWE and drift in R too. Below is a collection of resources you might find of use.

- The software carpentry github has an extensive set of tutorials for learning R programming (https://swcarpentry.github.io/r-novice-inflammation/)
- A great datacamp tutorial on functions and functional programming (https://www.datacamp.com/community/tutorials/functions-in-r-a-tutorial)
- Hadley Wickham's Advanced R tutorials (http://adv-r.had.co.nz/Functions.html)
- A comprehensive guide to vectorisation in R (http://www.noamross.net/blog/2014/4/16/vectorization-in-r--why.html)
- Datacamp's loop tutorial (https://www.datacamp.com/community/tutorials/tutorial-on-loops-in-r)
- An excellent and very detailed primer on population genetics in R (http://grunwaldlab.github.io/Population_Genetics_in_R/index.html)