The Theory of Natural Selection

Contents

Introduction **2**

What to expect . . . 2

Getting started . . . 2

1 R programming: Making custom functions **2**

1.1 Motivation . . . 2

1.2 Function basics . . . 3

1.3 A simple example . . . 3

1.4 Some important function properties . . . 4

1.5 A slightly more useful example . . . 5

1.6 Example: calculating genotype frequencies . . . 6

1.7 Creating a function of the drift simulation . . . 7

2 Evolutionary biology **8**

2.1 Understanding fitness . . . 8

2.2 One-locus model of viability selection . . . 10

2.3 Directional selection . . . 16

2.4 Over and underdominance . . . 18

2.5 Study questions . . . 20

2.6 Going further . . . 20

Important concept:
Green boxes summarises important concepts from the text.

Additional info
Blue boxes contain small tips and additional information for those that are interested. What's in these is hopefully useful, but not mandatory. Feel free to skip these if you want.

Advanced code
Yellow boxes contain code that you need to run in order to complete the tutorial, but that you don't necessarily need to understand.

Text in bold contains small exercises to do on your own throughout the tutorial. These are for your own understanding only, so you don't need to hand them in.

Introduction

Understanding natural selection is fundamental to properly understanding evolution. The basics of natural selection are relatively straightforward but surprisingly easily misunderstood. Our aim with this practical session is to use the R environment to actually demonstrate some models of selection and to also simulate evolution by natural selection. Thus we hope you'll start to see how R can be useful for helping you grasp difficult theoretical concepts as well as handling data.

What to expect

In this section we will:

- Learn about creating your own functions in R
- recreate fitness functions in R to develop our understanding of natural selection
- use the one-locus viability model of selection to simulate evolution
- model directional selection, overdominance and underdominance
- continue to practice your R skills

Getting started

The first thing we need to do is set up the R environment. We won't be using anything other than base R and the `tidyverse` package today. So you'll need to load the latter.

```
library(tidyverse)
```

1 R programming: Making custom functions

1.1 Motivation

In last week's tutorial and assignment, you used a for-loop to simulate genetic drift. If you did the assignment, you are probably tired of seeing code-snippets looking like this or similar:

```
N <- 10
ngen <- 1000
p_init <- 0.5

n10 <- rep(NA, ngen)
n10[1] <- p_init

for (i in 2:ngen){
  nA1 <- rbinom(1, 2*N, n10[i-1])
  n10[i] <- nA1 / (2*N)
}
```

Although the for-loop saved you thousands of lines of code compared to writing out the simulation manually, you still had to copy and paste the for loop itself many times. You had to write it 3 times for the tutorial and another 3 times for the assignment, just changing a single number each time! You may have thought that there has to be a better way of doing this, and there is! You can make this simpler by making your own custom functions, which you will learn about in the R-part of this tutorial.

1.2 Function basics

You have already used a lot of functions in R, with names like `mean()`, `sum()` and `ggplot()`. Common for these is that they take some input—arguments—does something with that input and returns the result. For making your own function, you need three things:

- A name for your function. This could be anything, but it's best to give them a sensible name.
- The arguments to your function. These are the values that you want to be able to change each time you run the function.
- What the function does with the arguments and what it returns. This is called the "body" of the function.

A basic custom function typically looks like this:

```
function_name <- function(argument){
  #the body of the function is inside curly brackets
  #here you do something with argument1 and argument2, for example:
  result <- argument^2

  # functions typically end with returning some value
  return(result)
}
```

The above function will take an input value, square it, and return the result. You can now use the function by writing `function_name(somevalue)`, like you would with the functions you've encountered so far. The arguments go into the parentheses of `function()`, and the body is inside curly brackets. The value to return is wrapped inside `return()` Note that the name of the function and arguments can be whatever you'd like. You can also have as many arguments as you want, separated by comma.

Important concept:
Custom functions makes a piece of code reusable, either for doing something many times, or for using for other purposes later. A function takes one or more *arguments*, does something with the arguments in the *body* and *returns* the result.

1.3 A simple example

Let's make a simple function that allows you to see clearly what the function is doing. The following function takes an argument, and prints it in a sentence:

```
print_arg <- function(argument){

  # make a string explaining what the argument is
  sentence <- paste("the argument is:", argument)

  # return the string
  return(sentence)
}
```

Note that when you run this code, nothing happens. But now you can call your new function with any argument:

```
print_arg(5)
#> [1] "the argument is: 5"

# you can store the returned value to an object and print it
horse_sentence <- print_arg("horse")
horse_sentence
#> [1] "the argument is: horse"

# you can put any expression as the argument,
# for example a logical statement
print_arg(pi > exp(1))
#> [1] "the argument is: TRUE"
```

You can also reference the argument by name when calling the function, like you learned about in week 1.

```
print_arg(argument = "zebra")
#> [1] "the argument is: zebra"
```

While this particular function is quite useless, notice how much typing you have saved already. We will move on to some more useful functions shortly, after an exercise.

Exercise: create a new function called `print_args()` that takes two arguments as input, and returns the text "the first argument is [argument1] and the second is [argument2]".

```
print_args <- function(argument1, argument2){
  # make a string
  sentence <- paste("the first argument is:", argument1, "and the second is", argument2)
  # return the string
  return(sentence)
}

# test the code

print_args(5, "horse")
#> [1] "the first argument is: 5 and the second is horse"
```

1.4 Some important function properties

One important thing about functions is that any objects you create **only exist inside the function**. This means that even though the above function creates an object named `sentence`, and we've run the function many times, there is no object called sentence in your R session:

```
sentence
#> Error in eval(expr, envir, enclos): object 'sentence' not found
```

This means that you don't need to worry about naming conflicts when running a function, which is convenient. However, it also means that you can't access any objects that are made inside your function, only whatever you put inside `return()`

Another important thing is that **you can only return 1 thing from a function**. This means that if you want to return multiple objects, you need to make these into a vector. In the following three examples, only the last function will work as intended:

```r
return_2 <- function(x, y){
  return(x, y) # will produce an error, you can only return a single object!
}
return_2(2, 4)
#> Error in return(x, y): multi-argument returns are not permitted

return_twice <- function(x, y){
  return(x) # x is returned, and the function stops
  return(y) # the function stops before this line, so y is lost forever
}
return_twice(2, 4)
#> [1] 2

return_vector <- function(x, y){
  return_vec <- c(x, y)
  return(return_vec) # this is OK, only one object is returned
}
return_vector(2, 4)
#> [1] 2 4
```

Important concept:

- Any variables you create inside a function only exists within that function.
- If you want to return more than 1 value, you need to bind the values together in e.g. a vector.

1.5 A slightly more useful example

A more useful function is often one that does some calculations and returns the result. Here's an example of a function that takes two arguments and divides the first by the other.

```r
divide <- function(numerator, denominator){
  result <- numerator/denominator
  return(result)
}

divide(5, 3)
#> [1] 1.666667
divide(10, 0)
#> [1] Inf
# oops
```

Exercise: create a function that takes two arguments and returns the sum of those arguments

```r
add <- function(x, y){
  result <- x + y
  return(result)
}
```

1.6 Example: calculating genotype frequencies

In last week's tutorial, you used the following snippet of code to calculate genotype frequencies from allele frequencies:

```
# first we set the frequencies
p <- 0.8
q <- 1 - p

# calculate the expected genotype frequencies (_e denotes expected)
A1A1_e <- p^2
A1A2_e <- 2 * (p * q)
A2A2_e <- q^2
# show the genotype frequencies in the console
c(A1A1_e, A1A2_e, A2A2_e)
#> [1] 0.64 0.32 0.04
```

This is something I imagine you will do often in this course and later in life if you pursue evolutionary biology, so it would make sense to make it into a function. I will let you try to do it on your own before showing you how to do this. Use the hints if you get stuck!

Exercise: create a function that takes a single argument, `p`, and returns the expected genotype frequencies according to the Hardy-Weinberg expectation.

Show hint

Start by wrapping the whole code snippet above inside the curly brackets of a function.

Show another hint

Remember that you can only return 1 object, so you need to make the genotype frequencies into a vector. Also, remember that you should be able to change p, so you should not have the line `p <- 0.8` in the body of the function.

```
calc_geno <- function(p){

  # calculate q from p
  q <- 1 - p

  # calculate the expected genotype frequencies (_e denotes expected)
  A1A1_e <- p^2
  A1A2_e <- 2 * (p * q)
  A2A2_e <- q^2

  # return the genotype frequencies
  geno_freq <- c(A1A1_e, A1A2_e, A2A2_e)
  return(geno_freq)
}
```

When testing your function, it should show the following output:

```
calc_geno(0.2)
#> [1] 0.04 0.32 0.64
calc_geno(0.5)
#> [1] 0.25 0.50 0.25
calc_geno(0.7)
#> [1] 0.49 0.42 0.09
```

Congratulations, you have now made your first truly useful function! Any time you need to calculate genotype frequencies from allele frequencies from now on, it will be a breeze!

Important concept:
Making a function isn't all that hard when you get used to it. A general recipe for converting code into a function is:

1. Paste your code inside the function body.
2. Make any objects you want to be able to change into an argument of the function (`p` in this example). Remember to remove those objects from the body of the function to avoid overwriting your arguments.
3. Wrap your result in `return()`.

1.7 Creating a function of the drift simulation

Now we're ready to make a function to simulate drift. Just to remind you, this is the code we used last week for the simulation:

```
# set population size
N <- 8

# set number of generations
ngen <- 100

# set initial frequency of A1
p_init <- 0.5

# create vector for storing results
p <- rep(NA, ngen)
# set first element to initial p
p[1] <- p_init


for (i in 2:ngen){
  # sample number of A1 alleles based on p in previous generation
  nA1 <- rbinom(1, 2*N, p[i-1])

  # set frequency of A1 as p in the current generation
  p[i] <- nA1 / (2*N)
}

p
```

Notice that there are three parameters we will probably want to be able to change for each simulation: `N`, `ngen` and `p_init`. It makes sense, then, to make these into parameters.

Exercise: Make the above drift simulation into a function. It should take three arguments: `N`, `ngen` and `p_init`, and return a vector of p's for each generation.

Show hint

Use the three steps from the previous section: paste the code into the body, turn `N`, `ngen` and `p_init` into arguments and return your result.

```r
drift_sim <- function(N, ngen, p_init){

  # create vector for storing results
  p <- rep(NA, ngen)
  # set first element to initial p
  p[1] <- p_init


  for (i in 2:ngen){
    # sample number of A1 alleles based on p in previous generation
    nA1 <- rbinom(1, 2*N, p[i-1])

    # set frequency of A1 as p in the current generation
    p[i] <- nA1 / (2*N)
  }

  return(p)
}
```

That concludes the R-part of this tutorial! The rest of the tutorial will be about fitness and selection, but will make use of functions for exploring this. Remember to check back here if you become unsure about how functions work!

Exercise: when working through the rest of the tutorial, think once in a while: "Will I need to do this many times? Could I make this code into a function? How would I do that?"

2 Evolutionary biology

2.1 Understanding fitness

What do we mean by **fitness**? In its simplest form, fitness is the defined as whether or not an organism is able to reproduce. Fitness is often mistaken as an individual attribute, but it is actually better explained as a difference in reproductive success among characters, traits or genotpes. If genotype A_1A_1 produces more offspring than A_2A_2 because of a trait the locus produces, we can say A_1A_1 has a higher fitness.

2.1.1 Absolute, relative and marginal fitness

So we might talk about **absolute fitness** - e.g. the expected reproductive success of A_1A_1. In population genetic models though, we more often than not refer to **relative fitness** - i.e. how fit genotypes are relative to one another. To denote relative fitness, we will follow the notation in the main text - i.e. w_{ij} for A_iA_j. Assuming locus A with two alleles, on average genotypes A_1A_1 and A_1A_2 produce 16 offspring each, whereas A_2A_2 produces 11 offspring on average. We calculate relative fitness as follows:

```r
# define the number of offspring per genotype
a <- c(A1A1 = 16, A1A2 = 16, A2A2 = 11)
# find the maximum fitness
max_fit <- max(a)
# determine the relative fitness
rel_fit <- a/max_fit
```

Note here that when defining our vector `a`, we also actually named each element of it. Names work a little differently in vectors to how they do in a `data.frame`. For example, `a$A1A1` will not work. However `a["A1A1"]` does.

Returning to the population genetics, we formulated things a little differently to how you first encountered this example in the main text, in order to demonstrate some R code. Nonetheless, the results and the general process are the same. Essentially, we define fitness relative to the maximum fitness. Since both A_1A_1 and A_1A_2 produce the highest number of offspring, their fitness is 1 whereas A_2A_2 has a lower relative fitness.

We might also want to calculate the **mean population fitness**, denoted as $\overline{w}$. This is essentially the sum of the relative fitness of each genotype multiplied by the genotype frequency. With R, calculating this is simple - we simply multiply a vector of genotype frequencies with the relative fitness and sum the result. We do this below:

```
# define the genotype frequencies - note different way to define!
geno_freq <- c(A1A1 = 0.65, A1A2 = 0.15, A2A2 = 0.2)
# calculate mean population fitness
w_bar <- sum(rel_fit * geno_freq)
```

So far, we have dealt with fitness for genotypes. This makes sense because selection acts on genotypes (and the phenotypes they convey). But what if we want to define the fitness of a specific allele? In this case, things become a bit more complicated because allelic fitness depends on the genotype the allele finds itself in. In our previous example, A_2 has a high fitness when it is in the heterozygous A_1A_2 genotype, but not when it is homozygous.

To account for this, we estimate **marginal fitness** for a given allele i as w_i^*. So for two alleles, the marginal fitness is:

- $w_1^* = pw_{11} + qw_{12}$
- $w_1^* = pw_{12} + qw_{22}$

Where p and q are the frequencies for A_1 and A_2 respectively. In other words, marginal fitness is a component of the fitness of the genotypes an allele occurs in *AND* the frequency of those genotypes. Let's calculate the fitness of our alleles using R.

```
## first calculate the allele frequencies
# define the total number of alleles
n <- 2*sum(a)
# calculate p
p <- ((a["A1A1"] * 2) + a["A1A2"])/n
# calculate q
q <- 1 - p

## now calculate the marginal fitness
w1 <- (p*rel_fit["A1A1"]) + (q*rel_fit["A1A2"])
w2 <- (p*rel_fit["A1A2"]) + (q*rel_fit["A2A2"])
```

Note again that we are explicitly naming elements of the vector to make the mathematics here clearer to you. However this can cause some annoying names to follow around your data. Take a look at `w1` and `w2` - you should see they have genotype names. The names still hang about because we used them right from the start, so we also remove them here by assigning a `NULL`:

```
# strip names
names(w1) <- NULL
names(w2) <- NULL
names(p) <- NULL
names(q) <- NULL
```

Now that we have the basics in place in terms of calculating fitness, we can go on to create a one-locus model of viability selection.

2.2 One-locus model of viability selection

When we talk about viability, we simply mean that individuals with different genotypes will vary in their probability of survivng until they are able to reproduce. Variation in viability will therefore effect the ability of individuals to reproduce and thus pass on their genes to the next generation. In other words, allele frequencies will change as a result of selection.

To model this, we will use a locus A with alleles A_1 and A_2 - each with a frequency p and q. From the last section, we have the relative fitness of the genotypes with w_{11}, w_{12} and w_{22}. For simplicity we will assume that fitness remains constant and that the frequency of the zygotes at each generation are in line with the Hardy-Weinberg expectation. Like in the book, we summarise the basics in the table below:

Genotype	Zygote frequency	Fitness
A_1A_1	p^2	w_{11}
A_1A_2	$2pq$	w_{12}
A_2A_2	q^2	w_{22}

In our model, we will calculate p_{t+1} the frequency of allele A_1 after a single generation of selection. This depends on three things: p - the allele frequency before selection, ${w_1}^*$, the marginal fitness of the A_1 allele and the difference between this and $\overline{w}$, the mean population fitness. So our model is basicaly:

$$P_{t+1} = \frac{p w_1^*}{\overline{w}}$$

We will use our previous results to calculate how p changes after a round of selection:

```
p_t <- (p*w1)/w_bar
```

If you compare `p` and `p_t`, you will see how the frequency in `p` changed as a result of selection. In fact, this is Δp. In R, we can easily calculate it as:

```
delta_p <- p_t - p
```

If everything worked correctly, your `delta_p` should be `0.0372093`.

2.2.1 Simulating selection under the one-locus model

So far, we have recreated the model for a single generation to try and understand how it works. But the beauty of R is that we can easily change the parameters to see how this will vary the change in frequency of our allele. The easiest way to do this is if we write a function that neatly summarises the code we have already explored - then all we need to do to see how the parameters have an effect is simply alter the arguments we give our function.

We will now write a function that takes the initial frequency of p and a vector consisting of the relative fitness of each genotype. This function will then calculate the allele frequencies, the mean population fitness and the marginal fitness of the alleles. Note how this is essentially all the code we have made so far pasted into the body of a function, and `p` and `rel_fit` is turned into arguments.

```
# a simple function to demonstrate the one locus selection model
selection_model_simple <- function(p, rel_fit){
  # define q
  q <- 1 - p
  # calculate genotype frequencies (under HWE)
  gf <- c(p^2, 2*(p*q), q^2)

  # calculate mean pop fitness
  w_bar <- sum(rel_fit*gf)

  # calculate marginal allele frequencies
  w1 <- (p*rel_fit[1]) + (q*rel_fit[2])
  w2 <- (p*rel_fit[2]) + (q*rel_fit[3])

  # calculate freq of p in the next generation
  p_t <- (p*w1)/w_bar

  # return the results
  return(p_t)
}
```

With this simple function, we can play around with the initial frequency of the A_1 allele and the relative fitness of the 3 genotypes. Try a few different values for yourself to see.

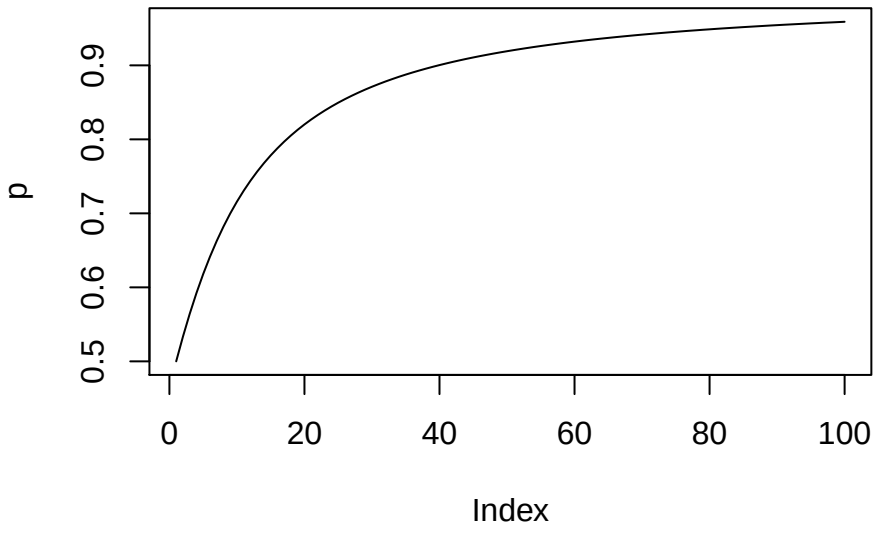
```
# keeping the initial frequency constant
selection_model_simple(p = 0.5, rel_fit = c(1, 1, 0.75))
selection_model_simple(p = 0.5, rel_fit = c(1, 1, 0.5))
selection_model_simple(p = 0.5, rel_fit = c(1, 1, 0.3))
```

To get a good idea of how the model works, what we really want to do is examine its change in frequency over time. We will initialise two values - the intial frequency `p` and the number of generations we want to simulate selection for, `n_gen`. Like the simulation we made last week, this can be done with a for-loop.

```
# first initialise the values
p_init <- 0.5
ngen <- 100
rel_fit <- c(1, 1, 0.75)

p <- rep(NA, ngen)
p[1] <- p_init

for(i in 2:ngen){
  p[i] <- selection_model_simple(p = p[i-1], rel_fit = rel_fit)
}

p
```

The change in allele frequency is now stored in your object `p`. This can easily be plotted straight away with base R's `plot()`:

```r
plot(p, type = "l")
```



2.2.2 Visualising selection with different parameters.

We already know that R's visualisation capabilities are extremely important for developing an understanding of data and concepts. So let's combine our programming and visualisation skills to demonstrate how varying the parameters really effect the outcome of our model.

First things first, we will take our simulation from the previous section and make it into it's own function `selection_sim`. Like before, paste the entire code into the body, and convert the relevant objects to arguments.

```r
## make a simulator function
selection_sim_simple <- function(p_init, rel_fit, ngen){

  p <- rep(NA, ngen)
  p[1] <- p_init

  for(i in 2:ngen){
    p[i] <- selection_model_simple(p = p[i-1], rel_fit = rel_fit)
  }

  return(p)
}
```

Note here how we have a custom function wrapped inside another custom function. This is a really useful concept, and allows for easier use and maintenance of your code. More on this in the assignment!

Anyway, now we can easily simulate selection over multiple generations. For example:

```
# Test the selection simulator
selection_sim_simple(p_init = 0.5, rel_fit = c(1, 1, 0.75), ngen = 1000)
```

So, now we will perform 4 simulations for 200 generations, keeping our initial frequency of `p` at 0.5. However, we will alter the relative fitness of the A_2A_2 genotype from 0.2 to 0.8.

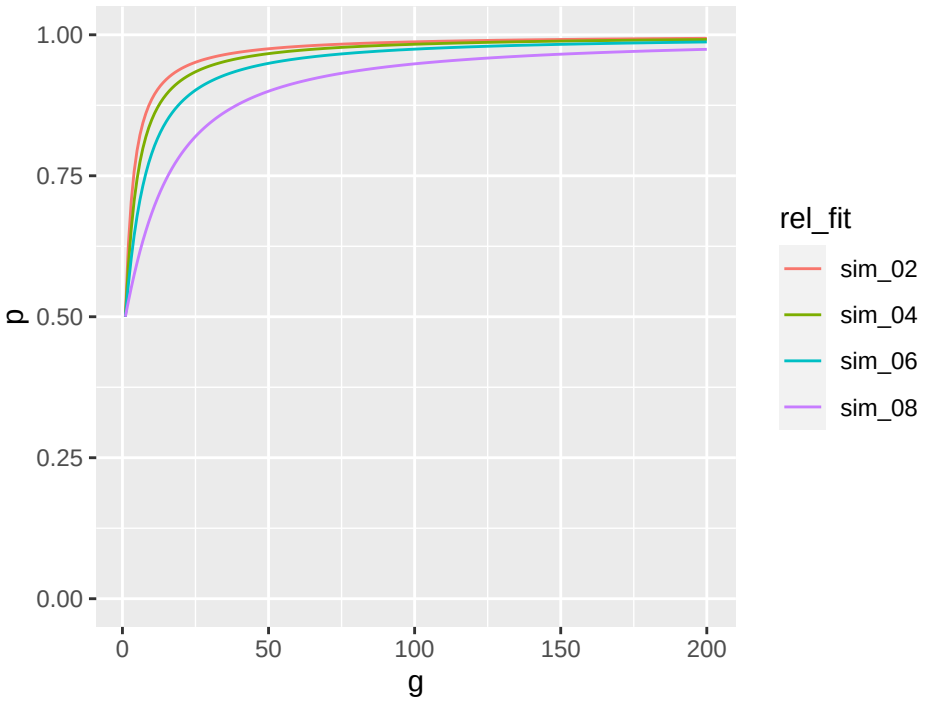
```
sim_02 <- selection_sim_simple(p_init = 0.5, rel_fit = c(1, 1, 0.2), ngen = 200)
sim_04 <- selection_sim_simple(p_init = 0.5, rel_fit = c(1, 1, 0.4), ngen = 200)
sim_06 <- selection_sim_simple(p_init = 0.5, rel_fit = c(1, 1, 0.6), ngen = 200)
sim_08 <- selection_sim_simple(p_init = 0.5, rel_fit = c(1, 1, 0.8), ngen = 200)
```

Now we have made 4 simulations. To plot these with `ggplot()`, we need to make them into a data frame and convert them to long form with `pivot_longer()`. See the tutorial from week 2 if you're unsure about why we do this.

```
sel_sims <- data.frame(
  g = 1:200, # number of generations
  sim_02,
  sim_04,
  sim_06,
  sim_08
)

# use gather to rearrange for plotting
# note the use of -g to select all the other columns
sel_sims_l <- pivot_longer(sel_sims, -g, names_to = "rel_fit", values_to = "p")
```

Then we can plot the data.

```
ggplot(sel_sims_l, aes(x = g, y = p, col = rel_fit)) +
  geom_line() + ylim(c(0,1))
```



So you can see from this plot that as the difference between the marginal fitness of A_1 and the mean population fitness $\overline{w}$ decreases, the proportional increase in allele frequency per generation slows down. More plainly, we can see that when w_{22} is 0.8, the increase in p per generation is slower than when w_{22} is 0.2.

2.2.3 Getting more from our selection functions

Let's take another look at our `selection_model_simple()` function.

```
# keeping the initial frequency constant
selection_model_simple(p = 0.5, rel_fit = c(0.8, 1, 0.7))
```

You will recall that when we defined the code for this function, we actually calculated quite a lot of stuff inside it - the frequency of the A_2 allele, the genotype frequencies, mean population fitness and marginal frequencies. But the only thing we used `return` to write out was the frequency of p in the next generation.

What if we want to extend our function to give us everything we calculated? This will be very useful for the upcoming sections where we will need all these parameters. The code below creates two updated functions for simulating selection. The difference is that instead of outputting a single value, we get a lot of extra information about fitness parameters. The functions are a bit more complicated than what you've seen so far, however, so don't worry if you don't understand everything that's happening.

Advanced code:
You need to run this code to complete the tutorial, but you don't need to understand it.

```
selection_model <- function(p, rel_fit){
  # define q
  q <- 1 - p
  # calculate genotype frequencies (under HWE)
  gf <- c(p^2, 2*(p*q), q^2)
```

```
  # calculate mean pop fitness
  w_bar <- sum(rel_fit*gf)

  # calculate marginal allele frequencies
  w1 <- (p*rel_fit[1]) + (q*rel_fit[2])
  w2 <- (p*rel_fit[2]) + (q*rel_fit[3])

  # calculate freq of p in the next generation
  p_t <- (p*w1)/w_bar

  # make vector for output
  output <- c(p = p, q = q, w_bar = w_bar,
              w1 = w1, w2 = w2, p_t = p_t)

  # return the results
  return(output)
}
```

```
selection_sim <- function(p_init, rel_fit, ngen){

  # Set first generation
  mod_pars <- t(selection_model(p = p_init, rel_fit = rel_fit))

  # simulate for n generations
  for(i in 2:ngen){
    mod_pars <- rbind(mod_pars, selection_model(p = mod_pars[i-1, "p_t"], rel_fit = rel_fit))
  }

  # make generations object
  g <- 1:ngen

  # return the result as a data frame
  return(as.data.frame(cbind(g, mod_pars)))
}
```

Our new `selection_model()` outputs a vector of useful values. The element `p_t` denotes p in the next generation, which was the output of the old `selection_model_simple()`.

```
selection_model(p = 0.5, rel_fit = c(0.8, 1, 0.7))
#>         p         q     w_bar        w1        w2       p_t
#> 0.5000000 0.5000000 0.8750000 0.9000000 0.8500000 0.5142857
```

The new `selection_sim()` returns a data frame, with 1 generation per row, and the fitness parameters in columns.

```
selection_sim(p_init = 0.5, rel_fit = c(0.8, 1, 0.7), ngen = 5)
#>   g         p         q     w_bar        w1        w2       p_t
#> 1 1 0.5000000 0.5000000 0.8750000 0.9000000 0.8500000 0.5142857
#> 2 2 0.5142857 0.4857143 0.8763265 0.8971429 0.8542857 0.5265021
#> 3 3 0.5265021 0.4734979 0.8772990 0.8946996 0.8579506 0.5369449
#> 4 4 0.5369449 0.4630551 0.8780120 0.8926110 0.8610835 0.5458728
#> 5 5 0.5458728 0.4541272 0.8785351 0.8908254 0.8637618 0.5535093
```

All these modeling results will be very useful in the next sections.

2.2.4 Can a rare mutant establish in a population?

To understand a selection model like the one we have just developed, it can be useful to see whether a rare mutant is able to establish in a population where the alternative allele is nearly fixed. This is the basis of **invasion fitness analysis**.

We can do this using a simple case where heterozygotes have a greater advantage than homozygotes. Using our newly modified `selection_model` function, we can test this by setting our relative fitness to show a higher relative fitness in heterozygotes and setting `p` to a high frequency, close to 1.

```
# keeping the initial frequency constant
selection_model(p = 0.99, rel_fit = c(0.7, 1, 0.8))
```

We see here that `w_bar` is around 0.7 - which we would expect given the frequency of A_1 (i.e `p`) and the relative fitness. Invasion fitness of A_2 is equivalent to the marginal fitness for the allele - so 0.99 here - close to 1, the relative fitness for the A_1A_2 heterozygote.

When invasion fitness is greater than resident mean population fitness, the model is not at a stable equilibrium as an allele can easily invade and increase in frequency. What would happen if the frequency of A_2 was almost fixed?

```
# keeping the initial frequency constant
selection_model(p = 0.01, rel_fit = c(0.7, 1, 0.8))
```

In this case, `w_bar` (the resident fitness) is higher than when the A_1 allele is fixed, however the marginal fitness of the A_1 allele (the invasion fitness) is higher than this - so in this scenarion A_1 could easily invade and increase in frequency too.

In order for the equilibrium to be stable with heterozygote advantage, the marginal fitness of the two alleles should equal one another. We will return to this in a short while.

2.3 Directional selection

Earlier, we simulated selection in order to understand our model. This was an example of **directional selection**. Now we are going to explore that in more detail - in particular, we want to see how genotypic fitness can alter the outcome of selection. We can visualise this using an **adaptive landscape**. Here we will use a simplified, 2D landscape - i.e. a plot of mean population fitness $\overline{w}$ against the allele frequency p.

To simplify our analyses, we will use three different scenarios. In each of them, we will start with a p of 0.01, and we will simulate 50 generations. In all cases A_1A_1 will have the highest relative fitness of 1 and A_2A_2 the lowest of 0.2. The only thing we will vary is the relative fitness of the heterozygote A_1A_2. Therefore we are simulating three types of genotypic fitness:

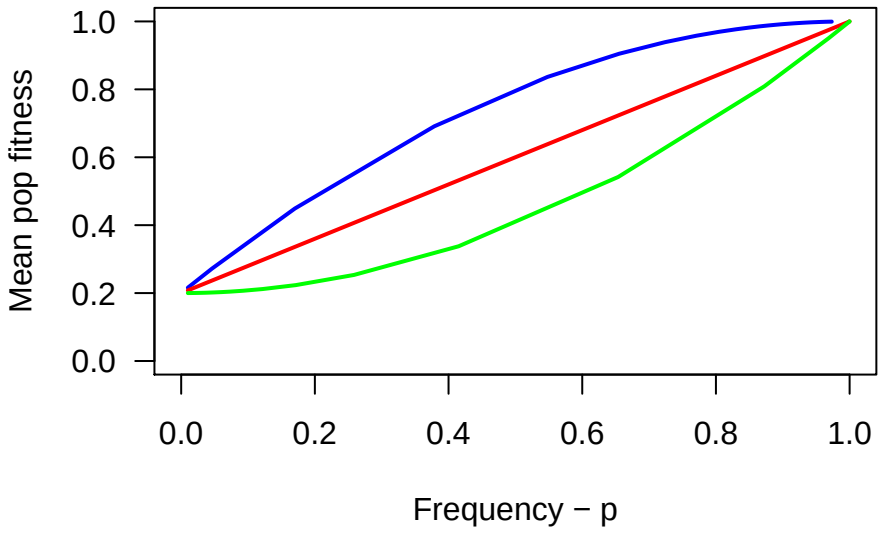
- dominance ($w_{12} = w_{11}$)
- additive inheritance ($w_{12} = \frac{w_{11} + w_{22}}{2}$)
- recessive ($w_{12} = w_{22}$)

```
# set generations
n_gen <- 50
# run simulations
```

```r
dom <- selection_sim(p = 0.01, rel_fit = c(1, 1, 0.2), n_gen)
add <- selection_sim(p = 0.01, rel_fit = c(1, 0.6, 0.2), n_gen)
rec <- selection_sim(p = 0.01, rel_fit = c(1, 0.2, 0.2), n_gen)
```

We'll take a brief break from the tidyverse approach now and plot this in base R. This is just because we want you to focus on what we are modelling, rather than reshaping the data too much. Don't worry too much about how this plot is created, the important thing is analysing the result!

```r
# intialise plot
plot(NULL, xlim = c(0, 1), ylim = c(0, 1),
     xlab = "Frequency - p", ylab = "Mean pop fitness", las = 1)
# add curves for each case
lines(dom$p, dom$w_bar, lwd = 2, col = "blue")
lines(add$p, add$w_bar, lwd = 2, col = "red")
lines(rec$p, rec$w_bar, lwd = 2, col = "green")
```

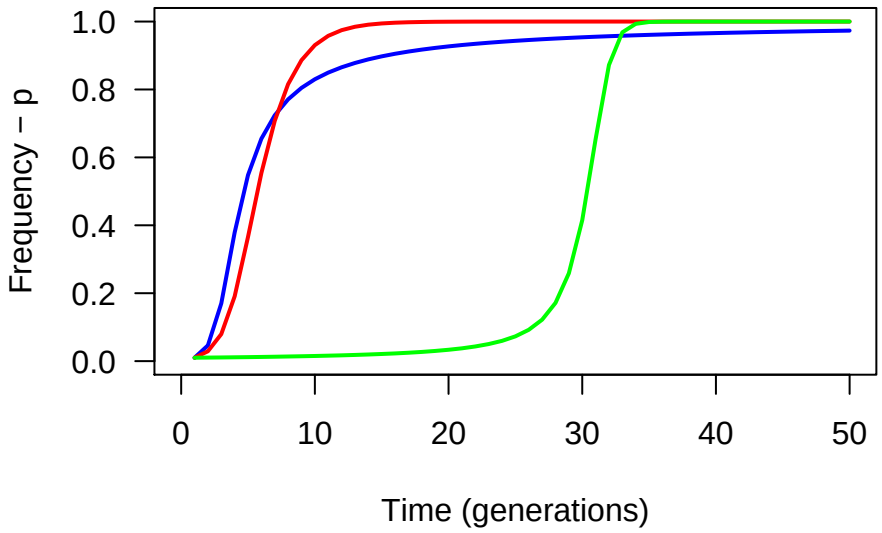


What does this show us? Well firstly, you might remember that this figure is more or less identical to **Figure 4.2** in the main textbook. The **blue line** is our case of dominance - i.e. both A_1A_1 and A_1A_2 identical, higher relative fitness compared to A_2A_2. In this case, the frequency of the A_1 allele quickly increases but slows down as the allele reaches fixation - hence the plateau at higher values of p. The **green** curve shows the recessive case. The increase in A_1 is lower when p is low because then the new mutant genotype is more likely to occur in heterozygotes. However as the frequency increases (i.e. p goes up), the mean population fitness increases rapidly two. In contrast, the additive case - in **red** - is just a linear increase with frequency.

We can try looking at the results of these simulations in a slgithly different way - to see how the frequency of A_1 alters over the 50 generations we simulated it for. Again we will use base R code to achieve this.

```r
# intialise plot
plot(NULL, xlim = c(0, n_gen), ylim = c(0, 1),
     xlab = "Time (generations)", ylab = "Frequency - p", las = 1)
```

```
# add curves for each case
lines(dom$g, dom$p, lwd = 2, col = "blue")
lines(dom$g, add$p, lwd = 2, col = "red")
lines(dom$g, rec$p, lwd = 2, col = "green")
```



All of these curves are S-shaped to some extent - so the transition from high to low frequency is rapit but the approach to fixation is much slower. The most marked difference is in the **green** line - the recessive case. Here, it takes time for A_1 alleles to occur in A_1A_1 homozygotes, so the new mutation remains at low frequency for quite a number of generations.

2.4 Over and underdominance

Earlier on, we learned a bit about heterozygote advantage. This also referred to as **overdominance** - i.e. when the fitness of the heterozygote is higher than either homozygyote. However, we also touched upon the fact that this can only be stable in under certain conditions - i.e. ${w_1}^* = {w_2}^*$. We also learned that when p or q were 0 (i.e. the population is fixed for either allele), these equilibria are unstable. So at what allele frequency is the population in a stable equilibria?

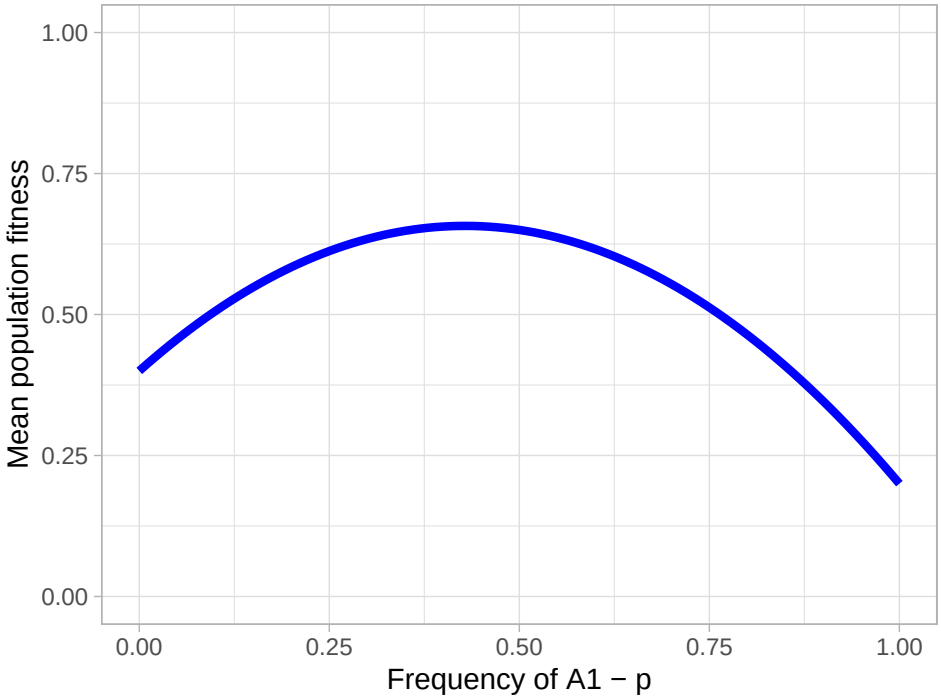
Previously we simulated data but this time, we are going to run our selection model for a range of values of p and see where mean population fitness, $\overline{w}$ is maximised. Then we'll visualise it make it clearer to ourselves. As with the book, we will set relative fitness as 0.2, 1 and 0.4 for the A_1A_1, A_1A_2 and A_2A_2 genotypes.

Advanced code:

```
# set the range of p
p_range <- seq(0, 1, 0.01)
# run selection_model for all values of p
overdom <- map_dfr(p_range, function(z) selection_model(p = z, rel_fit = c(0.2, 1, 0.4)))
```

The code above runs `selection_model()` for a range of values of p, and wraps the result in a data frame. Now this can be visualised with ggplot to see the stable equilibrium.

```
# initialise plot
a <- ggplot(overdom, aes(p, w_bar)) + geom_line(colour = "blue", size = 1.5)
a <- a + xlim(0, 1) + ylim(0, 1)
a <- a + xlab("Frequency of A1 - p") + ylab("Mean population fitness")
a + theme_light()
```



We can see mean population fitness is maximised at 0.42. This is the stable point on our 2D adaptive landscape.

Underdominance is the opposite of overdominance - i.e. it is heterozygote disadvantage. In short, relative fitness of the heterozygote is lower than either homozygote. Once again, we can visualise the stable equilibrium for a model of overdominance using more or less exactly the same code as before. All we really need to change is the relative fitness which we will set to 0.9, 0.3 and 1 for the A_1A_1, A_1A_2 and A_2A_2 genotypes.

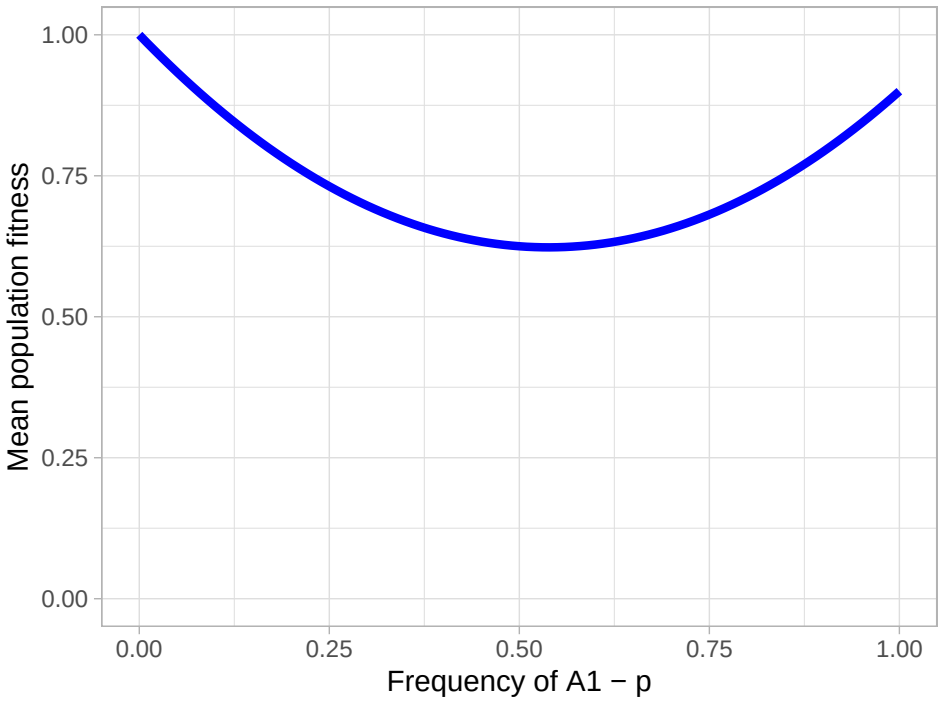
First, we run the model over different values of p.

Advanced code:

```
# set the range of p
p_range <- seq(0, 1, 0.01)
# run selection_model for all values of p
underdom <- map_dfr(p_range, function(z) selection_model(p = z, rel_fit = c(0.9, 0.3, 1)))
```

Then we plot it using `ggplot2`!

```
# initialise plot
a <- ggplot(underdom, aes(p, w_bar)) + geom_line(colour = "blue", size = 1.5)
a <- a + xlim(0, 1) + ylim(0, 1)
a <- a + xlab("Frequency of A1 - p") + ylab("Mean population fitness")
a + theme_light()
```



Here we see a scenario where underdominance is at a stable equilibria at $p = 0.53$.

2.5 Study questions

For study questions on this tutorial, download the `chapter4_R_questions.R` from Canvas or find it here.

2.6 Going further

- An introduction to lists in R
- More on lists from Hadley Wickham
- The definitive introduction to `purr` and the `map` function from Jenny BC
- Graham Coop's excellent notes on one-locus models of selection