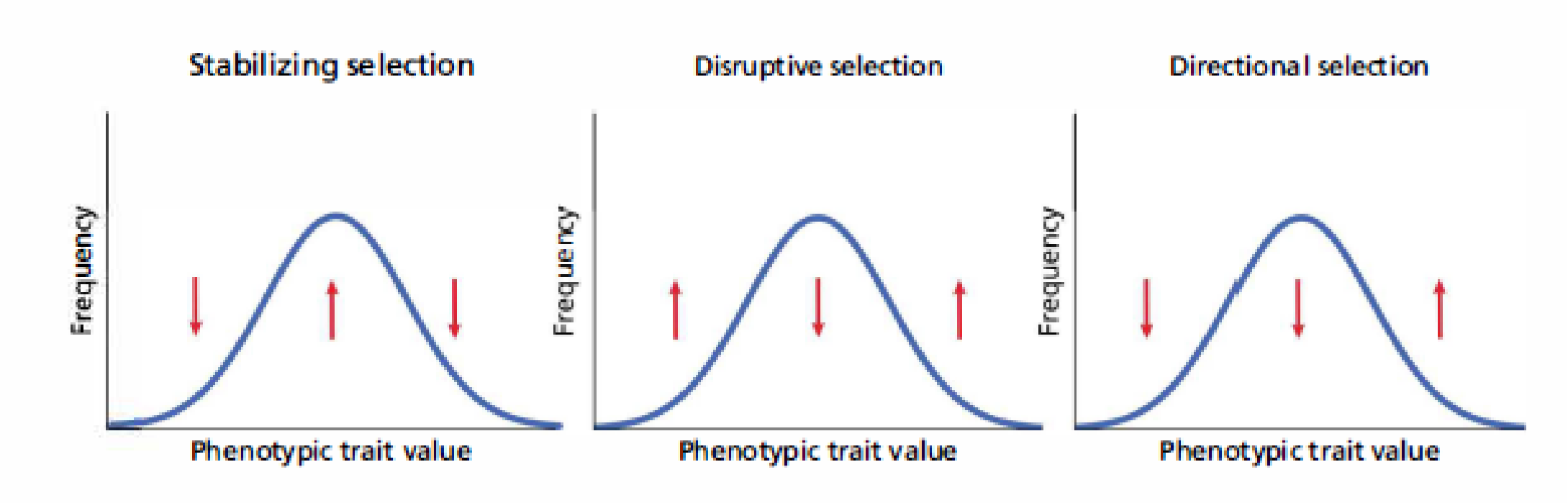
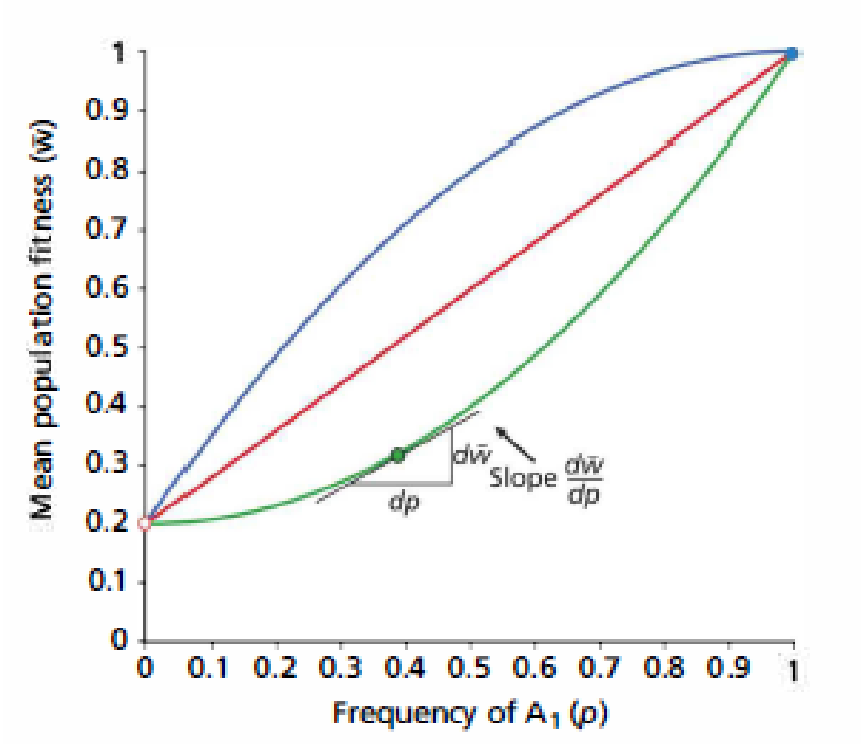
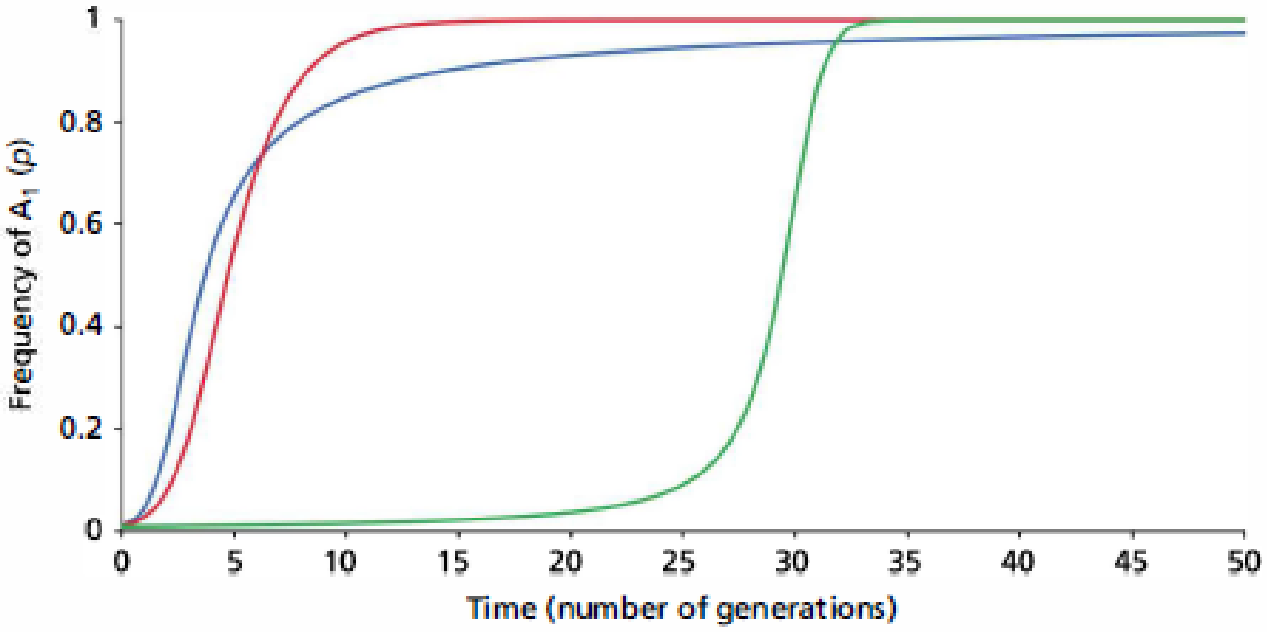
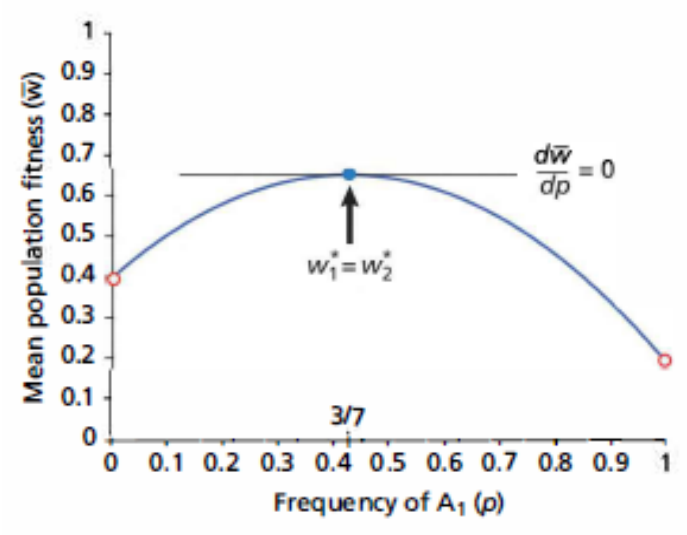
4. Doğal seçilim teorisi

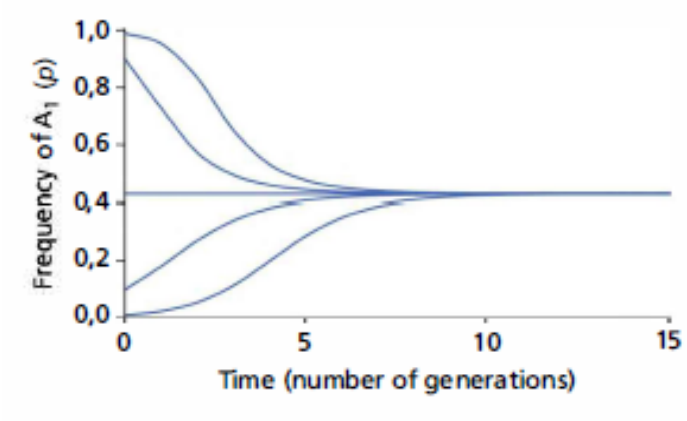
Teorik



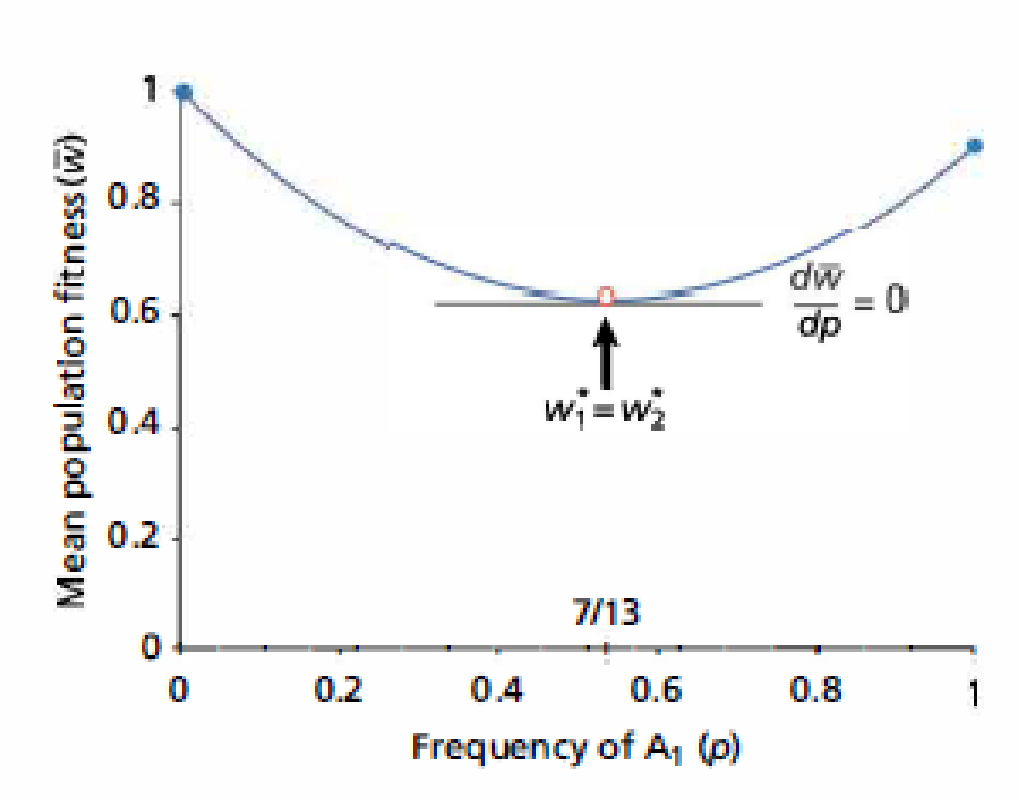


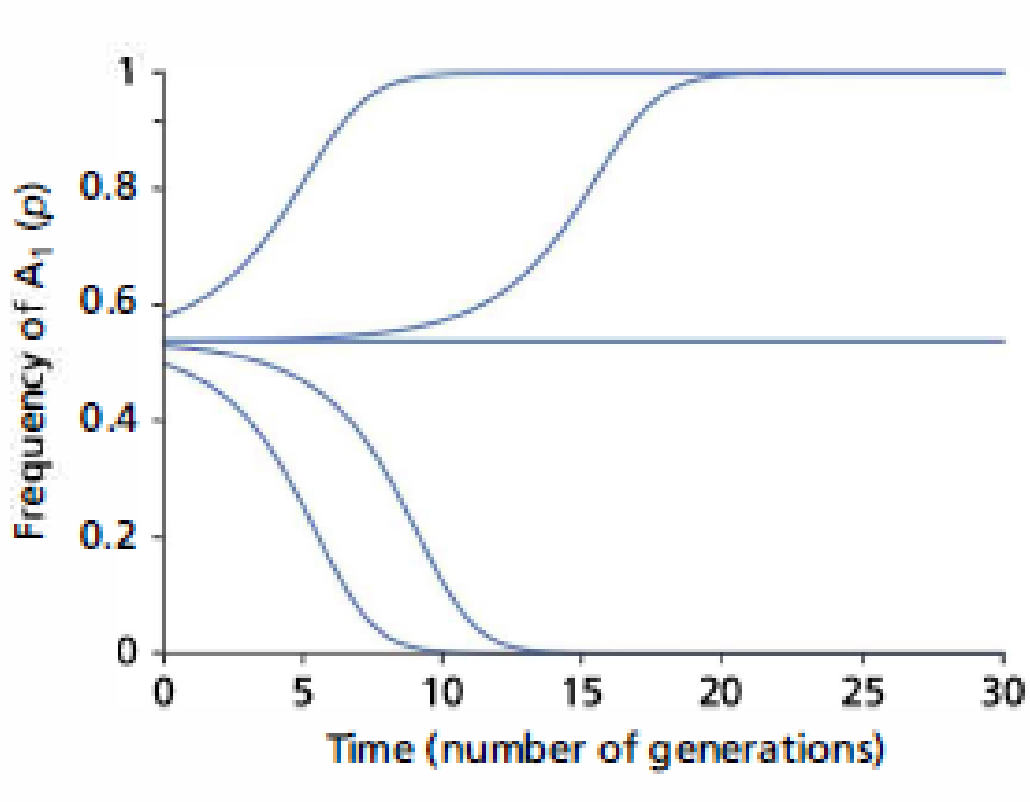


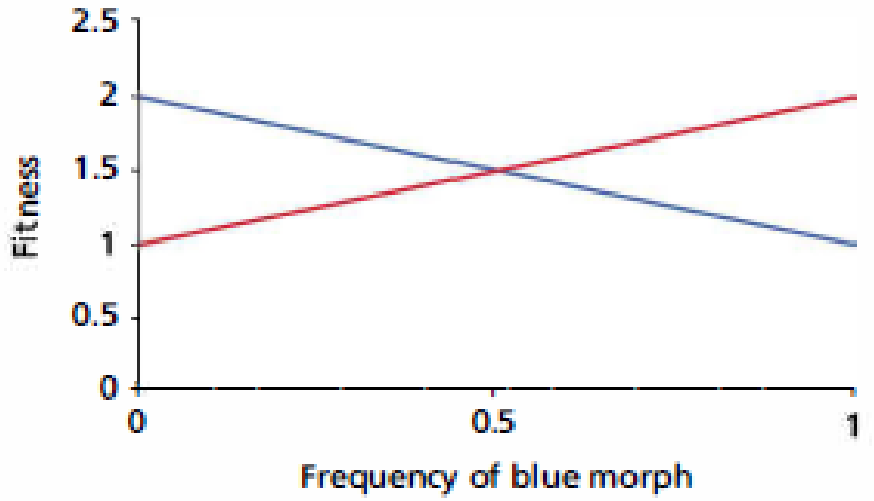


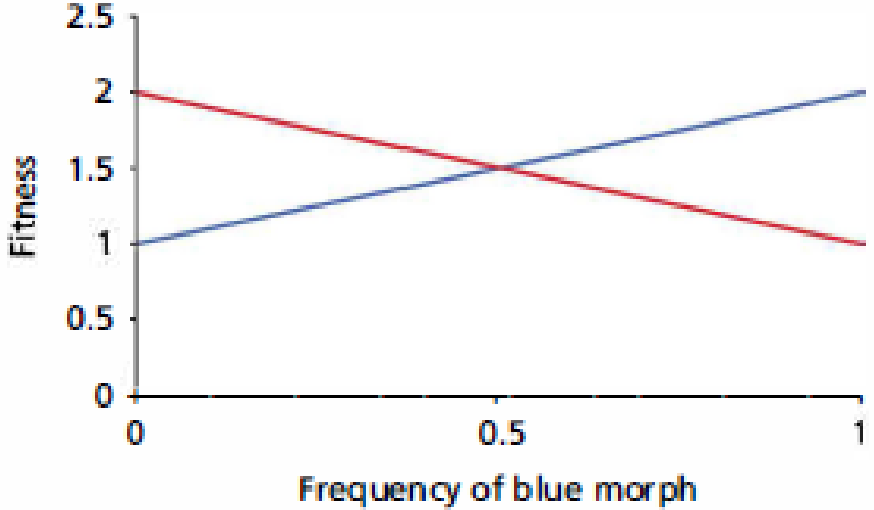


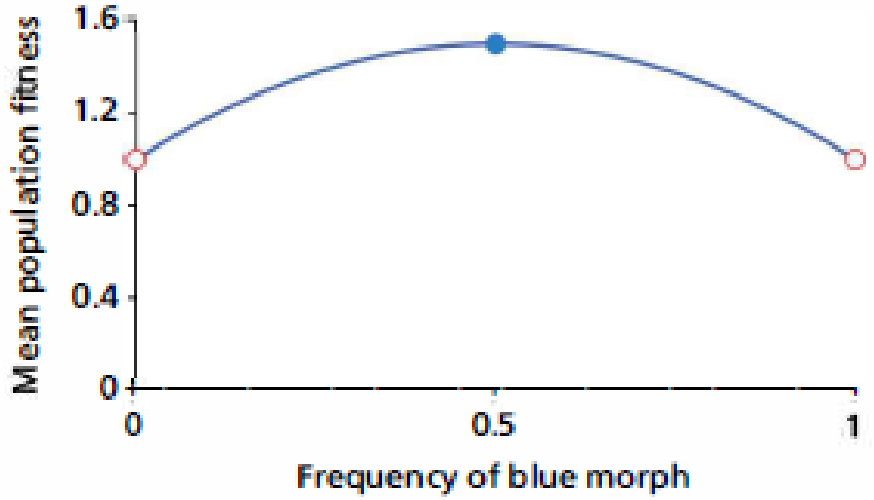
Figures and Tables were taken from Sætre, G. P., & Ravinet, M. (2019). *Evolutionary genetics: Concepts, analysis, and practice*. Oxford University Press, USA. For educational purpose only

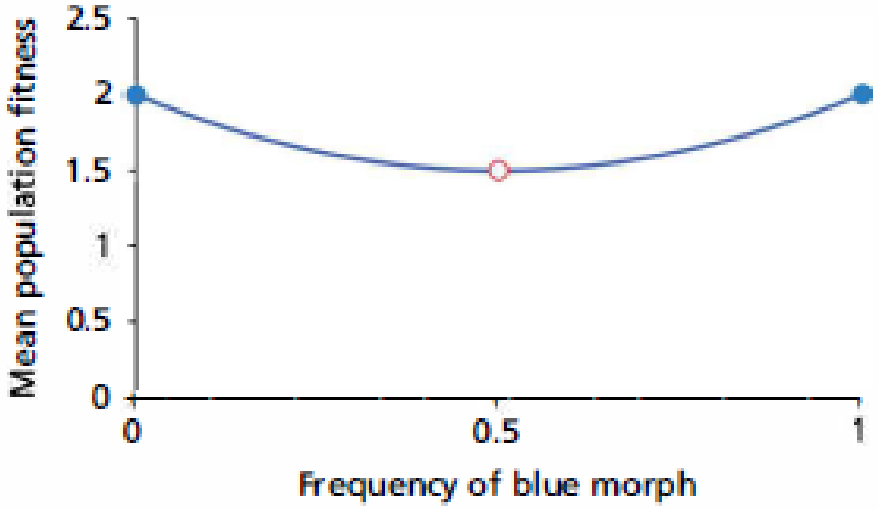


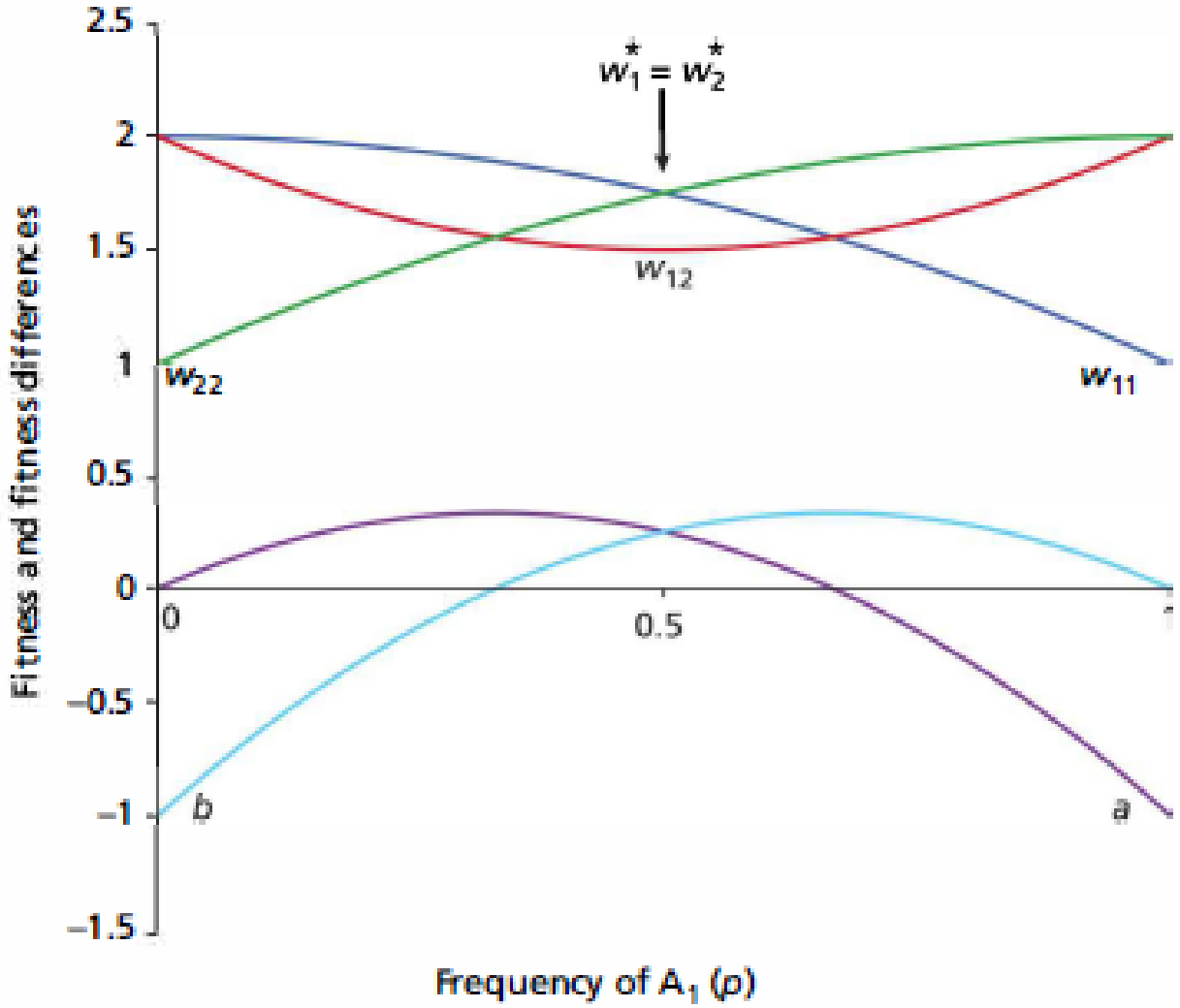


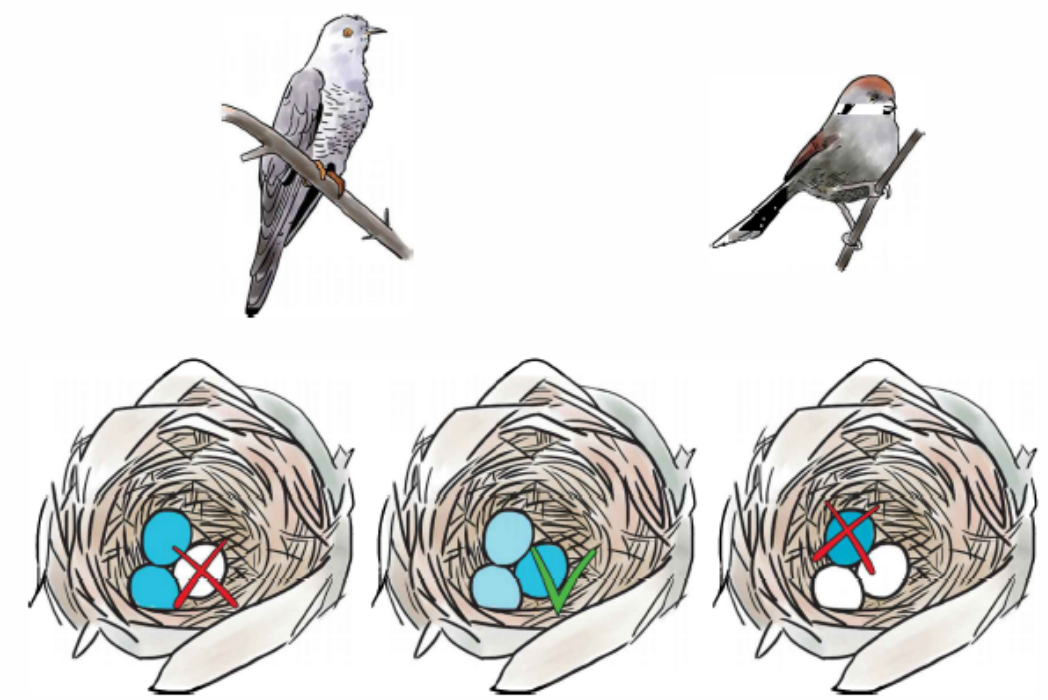


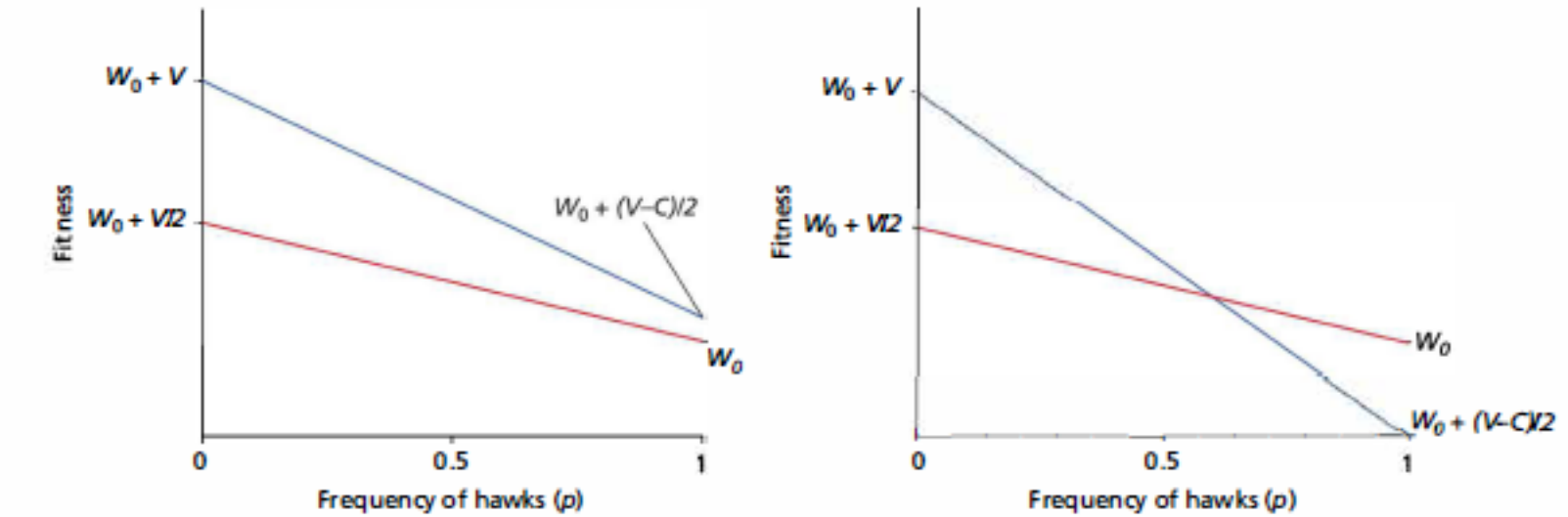


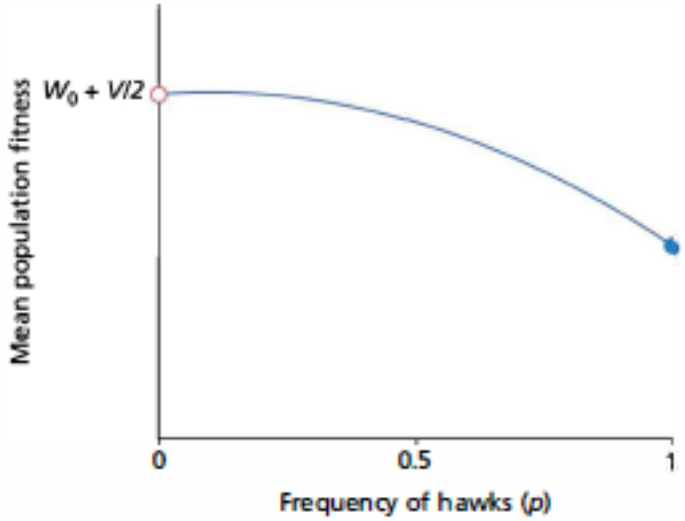


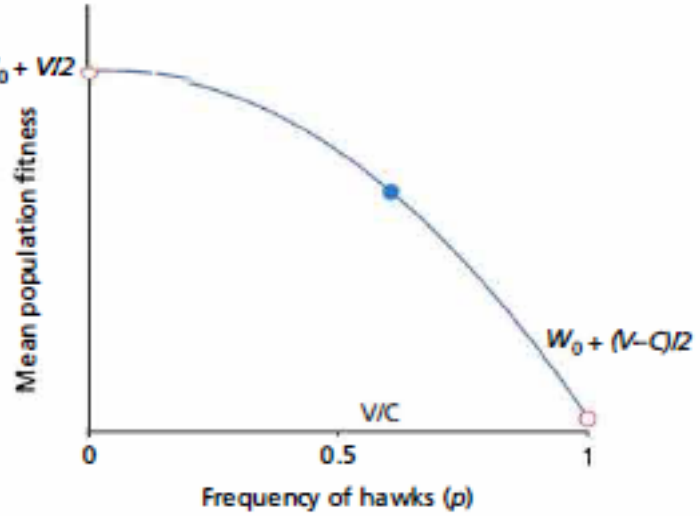






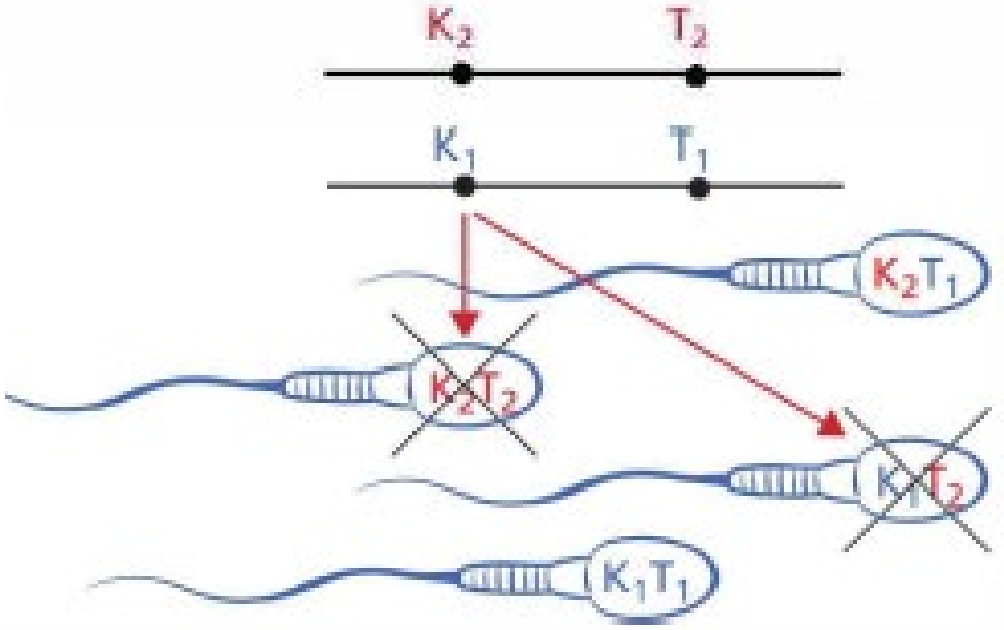






Figures and Tables were taken from Sætre, G. P., & Ravinet, M. (2019). *Evolutionary genetics: Concepts, analysis, and practice*. Oxford University Press, USA. For educational purpose only

(F × M)	Fitness	Frequency	Offspring genotype		
			rr	*Rr*	*RR*
rr × *rr*	1	g^2_{rr}	1		
rr × *Rr*	1 − 0.5*s*	$g_{rr}g_{Rr}$	0.5	0.5	
rr × *RR*	1 − *s*	$g_{rr}g_{RR}$		1	
Rr × *rr*	1	$g_{Rr}g_{rr}$	0.5	0.5	
Rr × *Rr*	1	g^2_{Rr}	0.25	0.5	0.25
Rr × *RR*	1	$g_{Rr}g_{RR}$		0.5	0.5
RR × *rr*	1	$g_{RR}g_{rr}$		1	
RR × *Rr*	1	$g_{RR}g_{Rr}$		0.5	0.5
RR × *RR*	1	g^2_{RR}			1



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

4. Doğal seçilim teorisi

Uygulama

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 exmaple, `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 code below is exactly the same as the code block above, but this time with numerical indexes - once again you see there are mulptile ways to do things in R. The names still hang about because we used them right from the start, so we also remove them here by assigning a `NULL`:

```
##  calculate the marginal fitness
w1 <- (p*rel_fit[1]) + (q*rel_fit[2])
w2 <- (p*rel_fit[2]) + (q*rel_fit[3])
# strip names
names(w1) <- names(w2) <- NULL
names(p) <- 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.

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:

$$P_{t+1} = \frac{pw_1^*}{\overline{w}}$$

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`.

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.

```
# a simple function to demonstrate the one locus selection model
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

  # 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(p = 0.5, rel_fit = c(1, 1, 0.75))
selection_model(p = 0.5, rel_fit = c(1, 1, 0.5))
selection_model(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 can do this neatly using the vectorisation skills we learned in the last session (https://evolutionarygenetics.github.io/Chapter3.html). We will initialise two values - the intial frequency `p` and the number of generations we want to simulate selection for, `n_gen`.

```
# first initialise the values
p <- p_init <- 0.5
n_gen <- 100

# use sapply to repeatedly run the selection model
p_vec <- sapply(1:n_gen, function(y){
  p_t <- selection_model(p = p, rel_fit = c(1, 1, 0.75))
  p <<- p_t
})

# combine the initial p and the values of p across each generation
p_vec <- c(p_init, p_vec)
```

So all we did here was wrap our `selection_model` function in an `sapply` command. We also took a moment to combine our initial value of p `p_init` to the the `p_vec` output. Note again that we used the `<<-` notation - this simply tells R to repeatedly update the value of `p` within our function. Why do we need to do this? Well you can run the following `sapply` wrapped version of `selection_model` to see.

```
# first initialise the values
p <- 0.5
n_gen <- 100

# use sapply to repeatedly run the selection model
sapply(1:n_gen, function(y){
  selection_model(p = p, rel_fit = c(1, 1, 0.75))
})
```

There is no assignment in the above block of code (on purpose so you don't write over your `p_vec`) but the output makes it clear - all we did here was rerun the selection model 100 times. We need to **update** `p` or else we are not properly simulating selection.

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 vectorised simulation from the previous section and make it into it's own function `selection_sim`.

```
## make a simulator function
selection_sim <- function(p, rel_fit, n_gen){
  # initialise variables
  p_init <- p
  my_rel_fit <- rel_fit

  # use sapply to repeatedly run the selection model
  p_vec <- sapply(1:n_gen, function(y){
    p_t <- selection_model(p = p, rel_fit = my_rel_fit)
    p <<- p_t
  })

  # combine the initial p and the values of p across each generation
  p_vec <- c(p_init, p_vec)

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

This is the same code, just wrapped in a function. The only thing you need to note here is that in order to distinguish between the `rel_fit` arguments made to the `selection_sim` and `selection_model` functions, we rename it as `my_rel_fit` within the function. This is a useful example of coding because here we have built a more complex function using another simple one we wrote early - this is the principle of modular programming (https://en.wikipedia.org/wiki/Modular_programming). It makes it much easier to diagnose issues as you can fix the individual functions separately.

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

```
# Test the selection simulator
selection_sim(p = 0.5, rel_fit = c(1, 1, 0.75), n_gen = 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. We can also do this very easily with vectorisation:

```
# set the vector for the relative fitness of A2A2
A2A2_rf <- seq(from = 0.2, to = 0.8, by = 0.2)

# run simulations for each
sel_sims <- sapply(A2A2_rf, function(z){
  selection_sim(p = 0.5, rel_fit = c(1, 1, z), n_gen = 200)
})

# assign names to the matrix
colnames(sel_sims) <- paste0("w22=", A2A2_rf)
```

Note that the last thing we did here was use `paste0` to create some column names for our matrix of selection simulations. We used `paste0` to combine `w22=` with the value of relative fitness for the A_2A_2 genotype. This will make sense in a moment when it is used in our plotting. Next up, we should get everything together in a `tibble` and use `gather` to prepare for plotting.

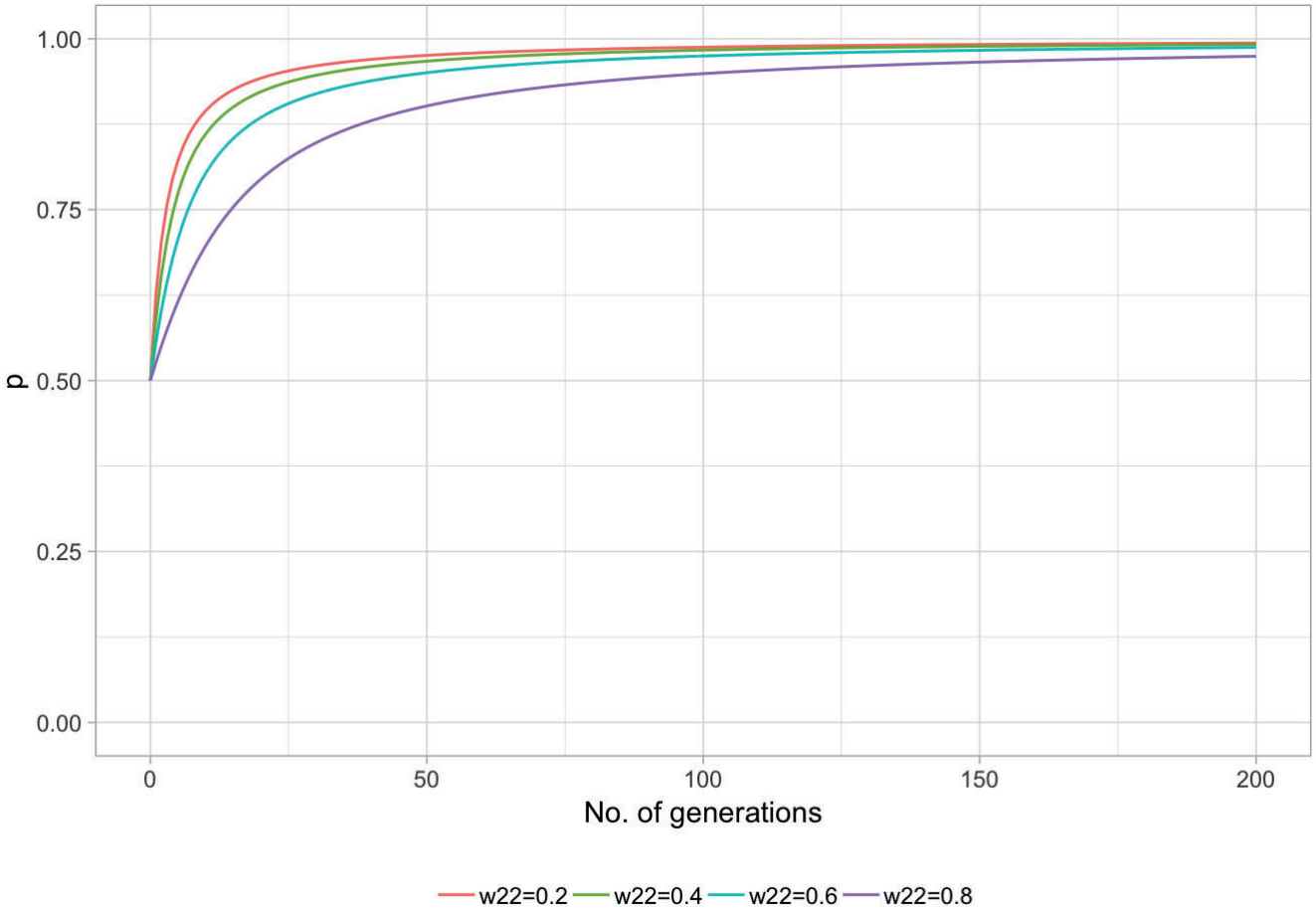
```
# create a generations vector
g <- seq(0, 200, 1)

# combine together to make a tibble
sel_sims <- as.tibble(cbind(g, sel_sims))
```

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

```
# use gather to rearrange for plotting
sel_sims_g <- gather(sel_sims, key = "rel_fit", value = "p", -g)
```

Just to make clear what we did here, we used `gather` to rearrange our `tibble` so that each value of `g` has a row for the different values of `rel_fit`. This makes it very straightforward to plot the different allele frequency changes over time using `ggplot2`. Like so:

```
# initialise plot
a <- ggplot(sel_sims_g, aes(g, p, colour = rel_fit)) + geom_line()
a <- a + xlab("No. of generations") + ylab("p") + ylim(0, 1)
a + theme_light() + theme(legend.position = "bottom", legend.title = element_blank())
```



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.

Getting more from our selection model function with lists

Let's take another look at our `selection_model` function.

```
# keeping the initial frequency constant
selection_model(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. However, we can't just write them all out as a vector; so to achieve this, we need to learn about a new R data structure - a **list**.

Lists are actually fairly straightforward. You can think of them as vectors which are able to store other vectors or objects. They are important because they can store data of different types. So for example, we will create a ist below that stores a **character** vector, a **numerical** vector and a **logical** vector.

```
# create vectors
a <- 1:10
b <- c("Ripley", "Lambert", "Ash", "Kane")
c <- c(TRUE, TRUE, FALSE, FALSE, FALSE)
# create list
mylist <- list(a = a, b = b, c = c)
```

One thing to note from the above code is that the vectors we stored as `a`, `b` and `c` are also all of different lengths. Anyway, now we can look at our list. First of all, since we explicitly named each of our vectors, i.e. `a = a` when we used the `list` function, we can access objects in the list by name.

```
# access list objects
mylist$a
mylist$b
mylist$c
```

But we can also use numbered indices to do the same thing.

```
# access list objects
mylist[1]
mylist[2]
mylist[3]
```

Strictly, because it's a list we should really use double square brackets. Compare the following code with the above.

```
# access list objects
mylist[[1]]
mylist[[2]]
mylist[[3]]
```

Now what if we wanted to get the first element of the `b` in our list? We could do this several ways:

```
# both of these are equivalent
mylist[[2]][1]
mylist$b[1]
```

The point here is that lists useful for storing information. So, let's integrate one into our `selection_model` function so we can get some of those calculations from out of it.

```
# a simple function to demonstrate the one locus selection model
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 list for output
  output <- list(p = p, q = q, geno_freq = gf, w_bar = w_bar,
                 w1 = w1, w2 = w2, p_t = p_t)

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

All we did here was combine some of the inner workings of our function into vectors (i.e. the `marginal_fit` object) and put everything together in a list we called `output`. So let's try our `selection_model` function once more.

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

When we run the model this time, we get a list with lots of different outputs! All of which will be very useful in the coming sections…

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.

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.

With our newly updated `selection_model`, we can do this again but we need to first update the simulation function to output the data in a more standard way. Copy and paste the code below to update your `selection_sim` function.

```
selection_sim <- function(p, rel_fit, n_gen){
  # initialise variables
  p_init <- p
  my_rel_fit <- rel_fit

  # use sapply to repeatedly run the selection model
  sim <- sapply(1:n_gen, function(y){
    out <- selection_model(p = p, rel_fit = my_rel_fit)
    p <<- out$p_t
    # return list
    return(out)
  }, simplify = FALSE)

  # use map data.frame to extract data we want
  sim_data <- sim %>%
    map_dfr(magrittr::extract, c('p', 'q', 'w_bar',
                                 'w1', 'w2', 'p_t'))

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

Just to give some insight about what is different here. Our `selection_model` function now outputs a `list` - so `sapply` no longer produces a vector of `p_t` but rather a list of lists. This is quite a complicated structure but using the `map_dfr` function from the `purr` package (part of the tidyverse), we can easily extract vectors of the variables we are interested in. It is beyond the scope of this tutorial to explore this in too much detail but we will return to it in a future session with some more advanced R tasks.

Whatever the case, if we rerun our `selection_sim` function, we get a nice `tibble` of the output. Like so:

```
# keeping the initial frequency constant
selection_sim(p = 0.01, rel_fit = c(0.7, 1, 0.8), 100)
```

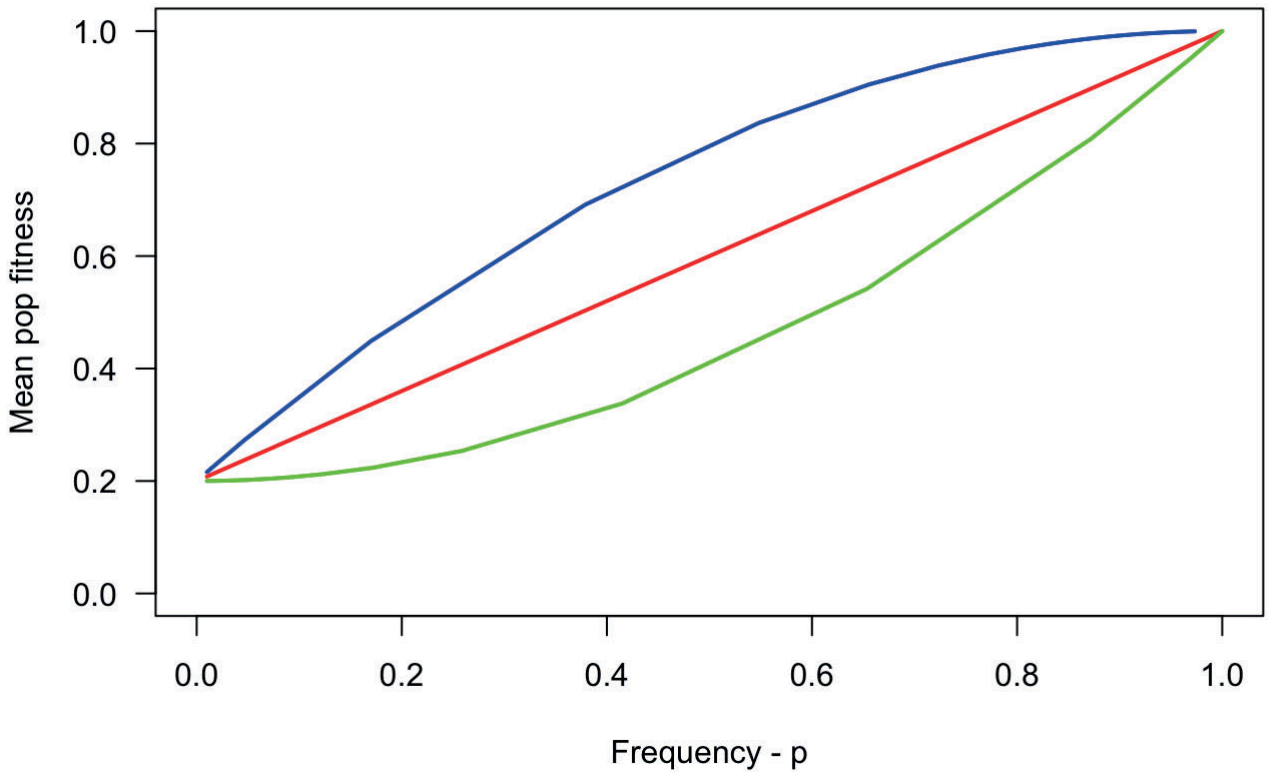
To simplify our analyses, we will use three different scenarios. In each of them, we will start with a p of 0.01, and we will simulate 50 generations. In all cases A_1A_1 will have the highest relative fitness of 1 and A_2A_2 the lowest of 0.2. The only thing we will vary is the relative fitness of the 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
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. Also it's good to learn new approaches for plotting!

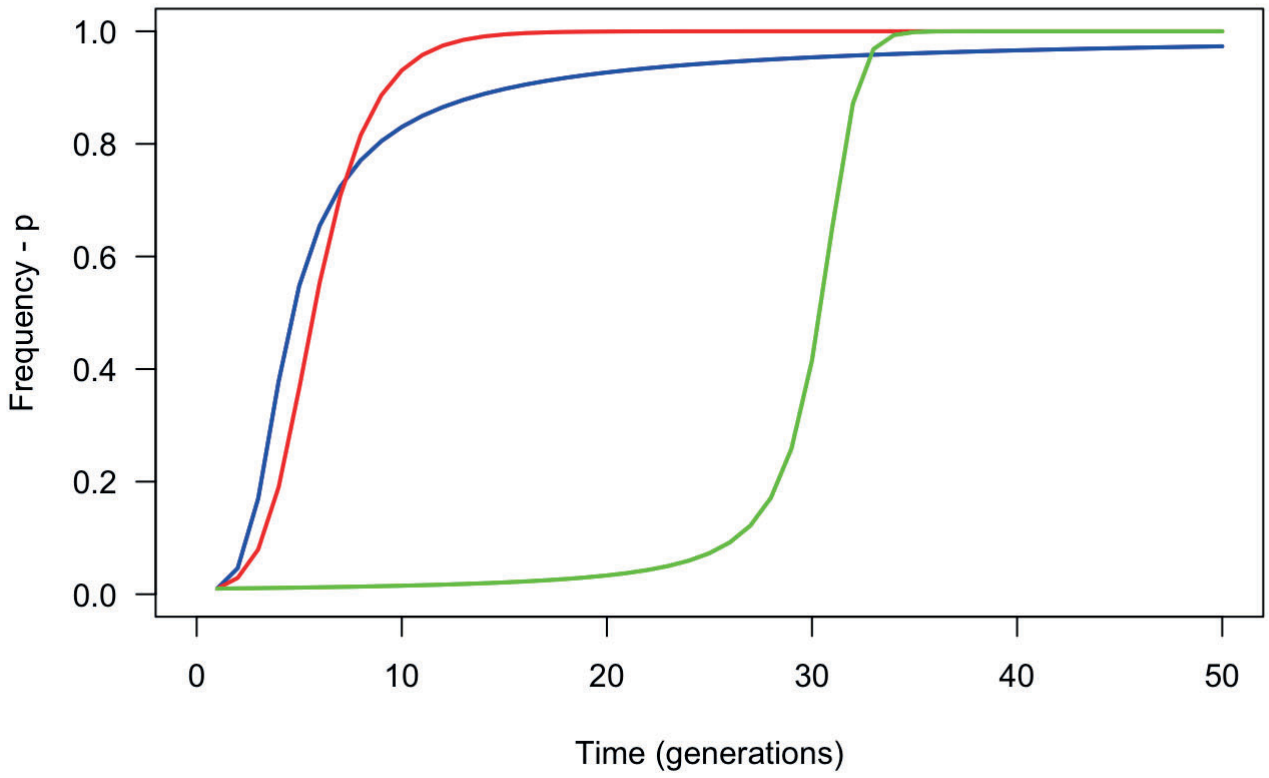
```
# intialise plot
plot(NULL, xlim = c(0, 1), ylim = c(0, 1),
     xlab = "Frequency - p", ylab = "Mean pop fitness", las = 1)
# add curves for each case
lines(dom$p, dom$w_bar, lwd = 2, col = "blue")
lines(add$p, add$w_bar, lwd = 2, col = "red")
lines(rec$p, rec$w_bar, lwd = 2, col = "green")
```



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.

```
g <- seq(1, n_gen, 1)

# 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(g, dom$p, lwd = 2, col = "blue")
lines(g, add$p, lwd = 2, col = "red")
lines(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.

Over and underdominance

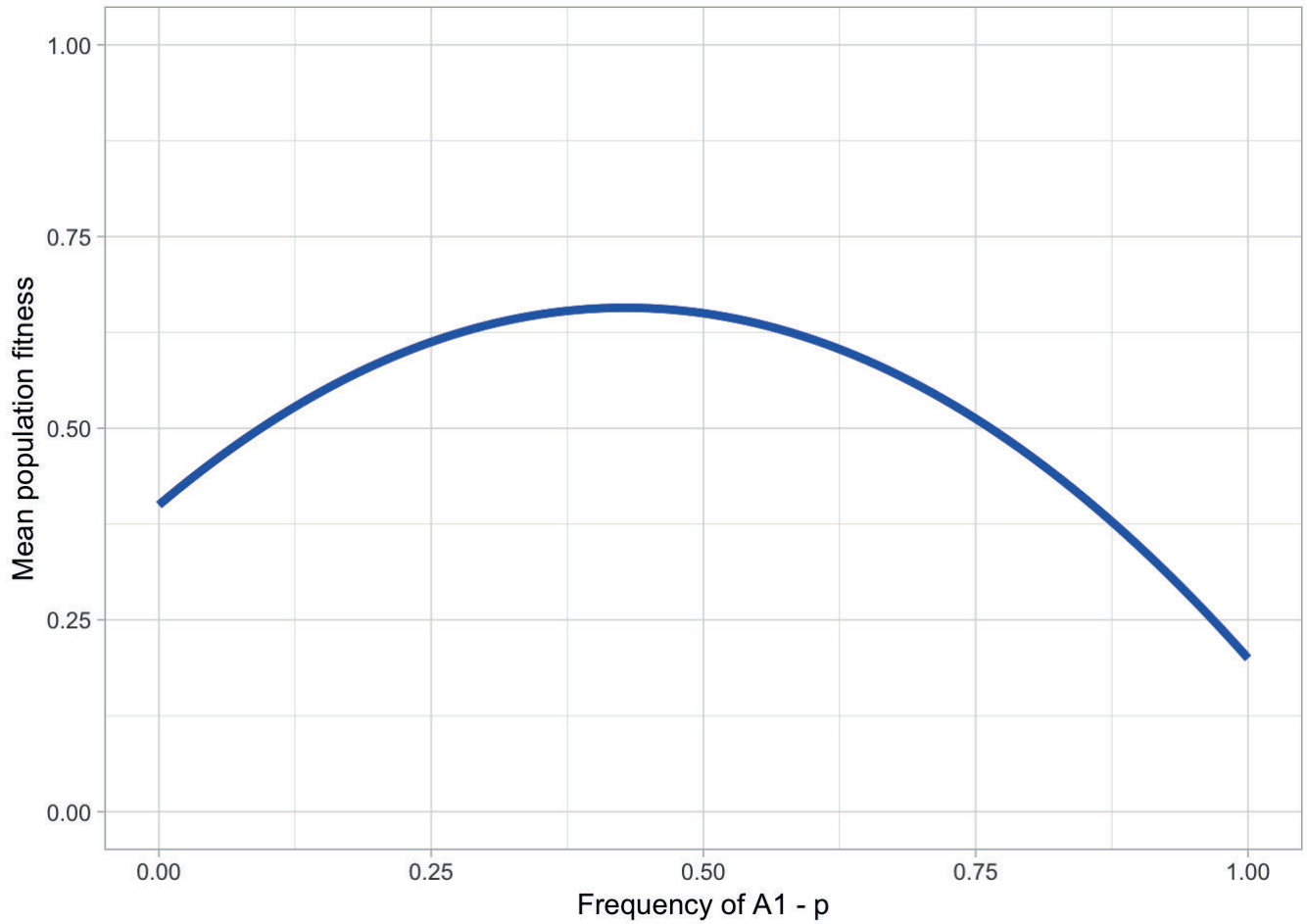
Earlier on, we learned a bit about heterozygote advantage. This also referred to as **overdominance** - i.e. when the fitness of the heterozygote is higher than either homozygyote. However, we also touched upon the fact that this can only be stable in under certain conditions - i.e. $w_1{}^* = w_2{}^*$. 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.

```
# set the range of p
p_range <- seq(0, 1, 0.01)
# use sapply to fit the model for all values of p and create a list
estimates <- map(p_range, function(z) selection_model(p = z, rel_fit = c(0.2, 1, 0.4)))
# extract what we're interested in from the list
overdom <- map_dfr(estimates, magrittr::extract, c('p', 'w_bar'))
```

So with this short piece of R code, we first set the range of all values of p with `seq`, then we used the `map` function to run our `selection_model` function on each value. Finally we used `map_dfr` to extract the vectors of `p` and `w_bar`. Now we can visualise these 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.

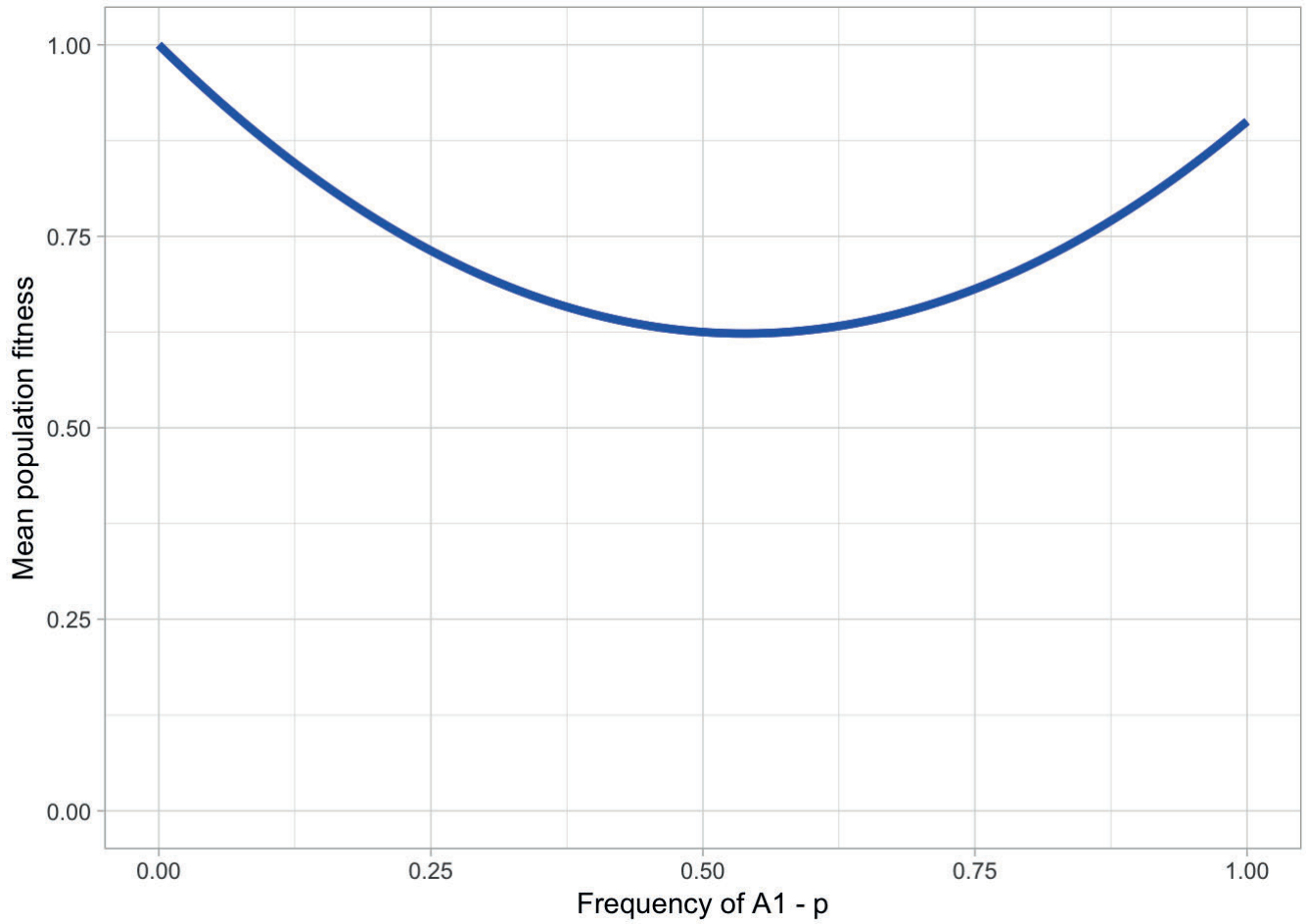
Underdominance is the opposite of overdominance - i.e. it is heterozygote disadvantage. In short, relative fitness of the heterozygote is lower than either homozygote. Once again, we can visualise the stable equilibrium for a model of overdominance using more or less exactly the same code as before. 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.

```
# set the range of p
p_range <- seq(0, 1, 0.01)
# use sapply to fit the model for all values of p and create a list
estimates <- map(p_range, function(z) selection_model(p = z, rel_fit = c(0.9, 0.3, 1)))
# extract what we're interested in from the list
underdom <- map_dfr(estimates, magrittr::extract, c('p', 'w_bar'))
```

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.

Study questions

For study questions on this tutorial, download the `chapter4_R_questions.R` from Canvas or find it here (https://evolutionarygenetics.github.io/chapter4_R_questions.R).