Human Genome Variation Lab

December, 2022

Table of Contents

# 1 About this Course

## 1.1 How to Use the Online Textbook

# 2 Evolutionary simulations

In this lab, we will build a simulation from scratch in R to explore genetic drift using the Wright-Fisher model.

## 2.1 Learning objectives

After completing this chapter, you’ll be able to:

1. Describe the phenomenon of genetic drift.
2. Explain why random draws from a binomial distribution are a good way to mimic the effect of drift.
3. Interpret allele frequency patterns that occur as a result of drift.
4. Write a for loop in R.
5. Write a function to run code multiple times with different parameters.

## 2.2 Background

### 2.2.1 Genetic drift

In all populations, **genetic drift** acts to change allele frequencies over time. Drift refers to random changes in an allele’s frequency between generations. These random changes occur because individuals carrying different alleles will have different numbers of offspring solely due to chance.

Drift differs from **selection**, which is a deterministic (non-random) change in an allele’s frequency. If an allele is under selection, it’s more likely to increase or decrease in frequency depending on whether it is beneficial or deleterious. Genetic drift, on the other hand, cannot consistently cause an allele’s frequency to increase or decrease.

Although both genetic drift and selection occur in real populations, drift is thought to be the primary driver of allele frequency changes, far outweighing the impacts of selection. This is called the [neutral theory of molecular evolution](https://www.nature.com/scitable/topicpage/neutral-theory-the-null-hypothesis-of-molecular-839/).

### 2.2.2 The Wright-Fisher model

The **Wright-Fisher model** is one of the most commonly used models of genetic drift.

#### 2.2.2.1 Population size

A Wright-Fisher population is a set of individuals who mate randomly. The model assumes that this number remains constant between generations. For simplicity, in this module we’ll also assume that every individual is haploid (so that they can only carry one copy of an allele).

Populations in the real world don’t behave as randomly as an idealized Wright-Fisher population, so their **effective population size** Ne is usually much smaller than their actual population size. The effective population size of the human population is only [12,800–14,400](https://www.pnas.org/content/109/44/17758) individuals, even though its actual size is around 7.8 billion.

#### 2.2.2.2 Allele frequency, fixation, and extinction

The Wright-Fisher model describes the behavior of a single **allele**, which is a variable site in a population (a SNP, insertion/deletion, version of a gene, etc.). If A and a are two alleles, representing two different sequences that exist at the same genomic location, Wright-Fisher allows us to track what happens to one of those alleles under genetic drift.

The allele of interest begins the simulation at some initial **allele frequency (AF)**. This allele frequency is the proportion of individuals in the population who carry that allele, and is always between 0 and 1.

An allele becomes **fixed** in a population when it reaches an allele frequency of 1. At this point, it is no longer considered an allele because everyone in the population carries it. Similarly, an allele goes **extinct** when it reaches an allele frequency of 0 and nobody in the population carries it.

## 2.3 Modeling allele frequencies

In the Wright-Fisher model, we track a population over the course of many generations. When creating each new generation of individuals, we determine how many of them carry A, our allele of interest.

For every individual, we perform a coin flip to determine whether or not they have the A allele. But unlike a coin, the probabilities of having A vs. a aren’t equal.

Instead, the probability of receiving the A allele is equal to A’s **allele frequency** in the current generation. The more common A is in this generation, the more likely it is that someone in the next generation will also carry it.

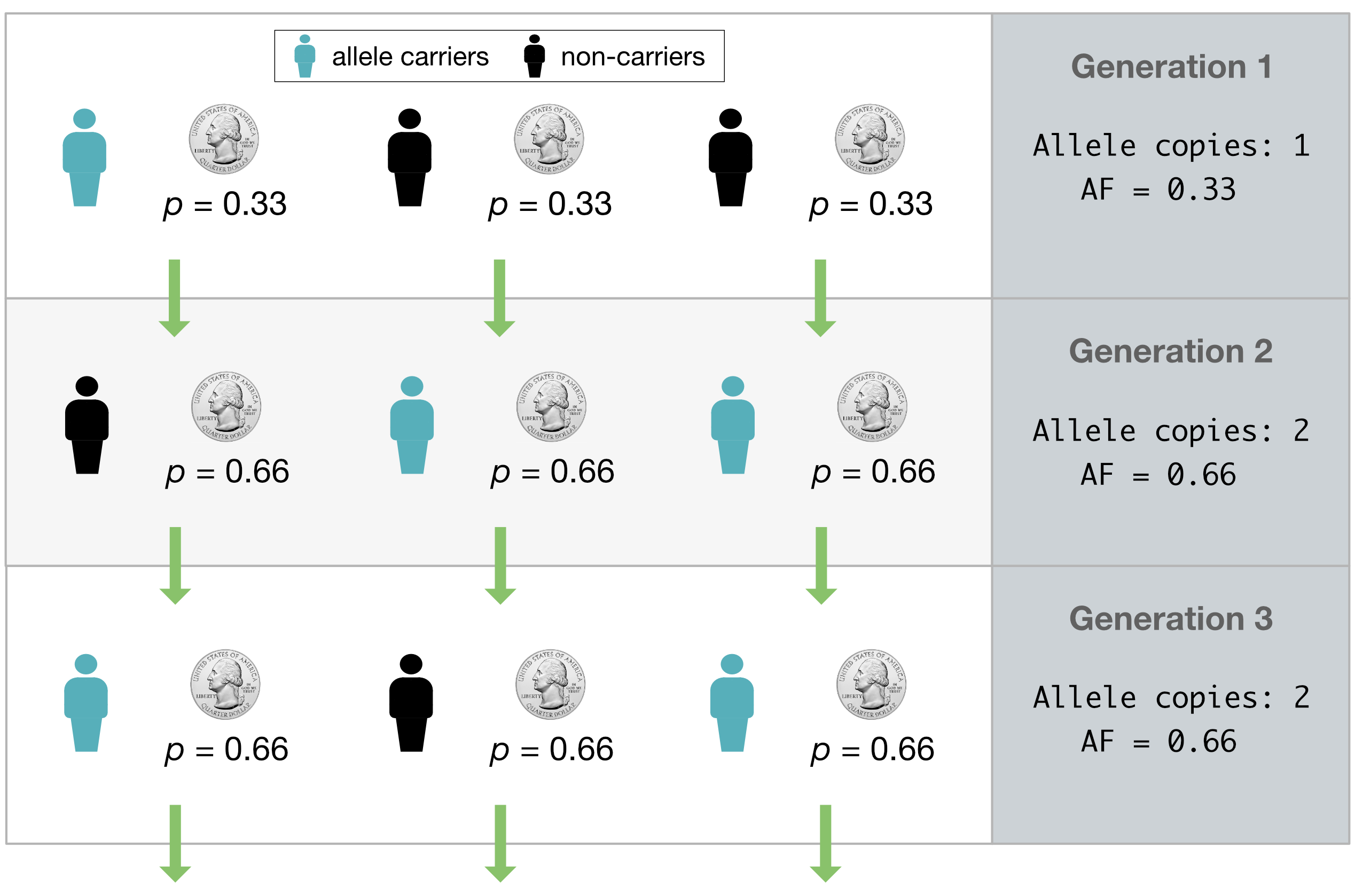
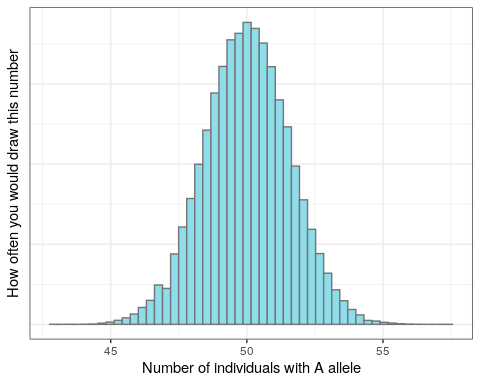


Figure : **Fig. 1. Modeling allelic inheritance with coin flips.** At the end of a generation, every individual flips a weighted coin to determine whether they will carry the blue allele in the next generation. The probability of carrying the allele is equal to the allele’s frequency in the current generation.

We do this coin flip N times, once for each individual in the population, to get the number of people in the next generation who carry the A allele (**Fig. 1**). Conveniently, instead of having to actually simulate all the coin flips, we can get this number by performing a single draw from a **binomial distribution** with size N and success probability p = AF. This distribution tells you how many successes you expect to see from a set of N independent coin flips.

### 2.3.1 The binomial distribution

If we try to draw 100,000 times from a binomial distribution with population size 100 and success probability (AF) 0.5, it’ll look something like this:



The majority of the distribution lies between 48 and 52. Just as we expect based on the allele frequency, the next generation will most likely have around 48-52 individuals with the A allele. But because this is a random draw, there’s a small chance that we might end up with many more or many fewer than that number.

## 2.4 Setup

### 2.4.1 R packages

Load the tidyverse library (which includes ggplot2) to use for the rest of the lab:

library(tidyverse)

## 2.5 A Wright-Fisher simulation

### 2.5.1 Drawing one generation from a binomial distribution

The basis of our simulation is R’s rbinom function, which allows us to sample from a binomial distribution. rbinom takes three arguments:

* n: how many times we’re drawing from the distribution
* size: the size of the population we’re sampling from (i.e. N)
* p: the success probability (i.e. allele frequency)

Every generation, we will draw **once** to produce the number of individuals carrying the A allele in the next generation.

Let’s once again look at a population of size 100, and an A allele currently at AF = 0.5. We use rbinom to get the number of individuals in the next generation who will have A:

rbinom(n = 1, size = 100, prob = 0.5)

## [1] 45

**Coding exercise**: Change the code above so that it returns the allele frequency rather than the number of individuals.

Solution

# divide by the population size to get AF  
rbinom(n = 1, size = 100, prob = 0.5) / 100

## [1] 0.51

**Question**: You’ll notice that every time we run the rbinom line, the number we get is different. Why is that?

Solution

rbinom generates a random number between 1 and 100. Because it’s random, the number it draws will be different every time we run it.

Currently, we’re drawing from a population of 100 individuals. Now let’s see what happens when we increase the population size. (Feel free to run this code block multiple times!)

rbinom(n = 1, size = 10000, prob = 0.5) / 10000

## [1] 0.4934

If you run the code block above multiple times, you’ll observe that the AF is much closer to 0.5 than it was with a population of size 100. This lends to our intuition that an allele’s frequency fluctuates much more when a population is small, and is more stable when the population size is large.

**Question**: As you increase population size, how does that affect an allele’s time to fixation?

Answer

As population size gets larger, the allele will take longer to fix.

### 2.5.2 Simulating multiple generations with a for loop

We can now draw *once* from a binomial distribution to get the number of individuals in one generation who carry the A allele. How do we adapt this to simulate multiple generations?

**Question**: Can we increase n (for example, with rbinom(n = 10, size = 100, prob = 0.5)) to draw multiple times?

Solution

No. Increasing n only gives you multiple replicate draws from the same distribution. It won’t update the AF between generations based on the new number of A alleles, because it uses prob = 0.5 every time.

Instead of drawing multiple times from the same distribution, we write a **for loop** to repeatedly generate and update the number of individuals with the A allele.

A **for loop** allows you to run some code X number of times. For example:

for (i in 1:3) {  
 print(i)  
}

## [1] 1  
## [1] 2  
## [1] 3

This for loop goes through all the values between 1 and 3, and prints each of them out.

We can write a similar for loop that includes rbinom:

for (i in 1:3) {  
 print(rbinom(n = 1, size = 100, prob = 0.5) / 100)  
}

## [1] 0.49  
## [1] 0.45  
## [1] 0.43

This is very close to what we want. However, we’re still running rbinom with the same AF in every iteration.

How do we change this to update the AF each generation? We can add a freq variable that keeps track of the current allele frequency:

# start an initial AF of 0.5  
freq <- 0.5  
  
for (i in 1:3) {  
 # run rbinom to generate the AF for the next generation  
 new\_freq <- rbinom(n = 1, size = 100, prob = freq) / 100  
 # print so that we can see what `freq` is each time  
 print(new\_freq)  
 # update `freq` in each iteration of the loop  
 freq <- new\_freq  
}

## [1] 0.53  
## [1] 0.51  
## [1] 0.52

Now the for loop is updating the allele’s frequency every generation and running rbinom with that new frequency.

One more modification: Providing the variable freq as an input to rbinom allows us to update freq with each for loop iteration. But it also allows us to pass rbinom a variable, rather than hard-coding in a population size.

**Coding exercise**: Add to the code above so that we also provide Ne (effective population size) as a variable (*without* updating it in the for loop).

Solution

# set effective population size outside of for loop  
Ne <- 100  
# start an initial AF of 0.5  
freq <- 0.5  
  
for (i in 1:3) {  
 # run rbinom to generate the AF for the next generation  
 new\_freq <- rbinom(n = 1, size = Ne, prob = freq) / Ne  
 # print so that we can see what `freq` is each time  
 print(new\_freq)  
 # update `freq` in each iteration of the loop  
 freq <- new\_freq  
}

## [1] 0.53  
## [1] 0.47  
## [1] 0.43

## 2.6 Plotting

### 2.6.1 Visualizing changes in AF over generations

Try increasing the number of generations we run the simulation for. What patterns of change do you observe in the allele frequencies?

Ne <- 100  
freq <- 0.5  
  
for (i in 1:20) {  
 new\_freq <- rbinom(n = 1, size = Ne, prob = freq) / Ne  
 print(new\_freq)  
 freq <- new\_freq  
}

## [1] 0.53  
## [1] 0.44  
## [1] 0.41  
## [1] 0.48  
## [1] 0.55  
## [1] 0.49  
## [1] 0.5  
## [1] 0.5  
## [1] 0.44  
## [1] 0.48  
## [1] 0.51  
## [1] 0.46  
## [1] 0.45  
## [1] 0.48  
## [1] 0.46  
## [1] 0.51  
## [1] 0.55  
## [1] 0.58  
## [1] 0.54  
## [1] 0.56

It would be useful to plot how the AF changes over time, so that we can look at it visually. We can do this by storing the AF at each generation in a **vector**, which you can think of as R’s version of a list.

Vectors are formed with the c() function, which stands for “combine”:

my\_vec <- c(0.5, 0.6)  
my\_vec

## [1] 0.5 0.6

You can append elements to a vector called my\_vec by running: my\_vec <- c(my\_vec, new\_element).

**Coding exercise**: Modify the code block with our for loop to create a vector for storing allele frequencies, and then append the updated AF to it every generation.

Hint

Create the vector **before** the for loop. Then append to the vector **within** the for loop.

Solution

Ne <- 100  
freq <- 0.5  
# create vector to store AFs in  
freq\_vector <- freq  
  
for (i in 1:20) {  
 new\_freq <- rbinom(n = 1, size = Ne, prob = freq) / Ne  
 # add new freq to the AF vector  
 freq\_vector <- c(freq\_vector, new\_freq)  
 freq <- new\_freq  
}  
  
freq\_vector

## [1] 0.50 0.52 0.64 0.63 0.65 0.63 0.61 0.68 0.68 0.65 0.63 0.63 0.64 0.62 0.57  
## [16] 0.60 0.61 0.63 0.64 0.65 0.68

We can plot this allele frequency vector with ggplot. First, because ggplot requires its input data to be formatted as a table, we have to convert the vector into some form of table. (We chose tibble form here because it’s easy to convert to.)

sim\_results <- tibble(af = freq\_vector)  
sim\_results

## # A tibble: 21 x 1  
## af  
## <dbl>  
## 1 0.5   
## 2 0.52  
## 3 0.64  
## 4 0.63  
## 5 0.65  
## 6 0.63  
## 7 0.61  
## 8 0.68  
## 9 0.68  
## 10 0.65  
## # … with 11 more rows

This table contains the information that we want on the plot’s y axis. We can now add in a column containing the plot’s x axis data. This should be the **generation** that each AF value corresponds to.

sim\_results <- tibble(af = freq\_vector,  
 gen = 1:21)  
sim\_results

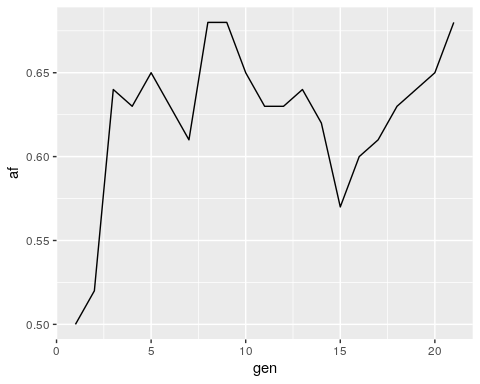
## # A tibble: 21 x 2  
## af gen  
## <dbl> <int>  
## 1 0.5 1  
## 2 0.52 2  
## 3 0.64 3  
## 4 0.63 4  
## 5 0.65 5  
## 6 0.63 6  
## 7 0.61 7  
## 8 0.68 8  
## 9 0.68 9  
## 10 0.65 10  
## # … with 11 more rows

Why does the gens column go from 1 to 21 (instead of 20)?

We add our starting allele frequency to freq\_vector, and then simulate 20 generations of drift. This means that we end up with 21 AFs in our vector.

Now we can finally plot the trajectory of AFs over time with ggplot.

ggplot(data = sim\_results,  
 aes(x = gen, y = af)) +  
 geom\_line()



## 2.7 Creating a simulation function

### 2.7.1 Simulating with different parameters with a function

It would be nice to be able to run the Wright-Fisher simulation with different parameters – like different starting allele frequencies, population sizes, etc. – without having to modify the for loop code every time. We can use a **function** to generalize the code above so we can easily re-run it.

Whenever you have a bunch of lines at the beginning of your code where you set variables, you should always be thinking that you can make it into a function.

The structure of an R function

You’ve already encountered many functions in R, even if you didn’t realize it at the time - rbinom, ggplot, and print are all examples of functions. An R function has [four parts](https://www.tutorialspoint.com/r/r_functions.htm):

<Name> <- function(<Argument(s)>) {  
 <Body>  
 <return()>  
}

* **Name** − The function is stored in your R environment as an object with this name, and you use the name to call it
* **Argument(s)** − Optional; input values that the function performs operations on
* **Body** − The code that describes what the function does
* **Return** − Optional; a return statement allows the function to return a value to the user. Without a return statement, you won’t be able to access the function’s output

Here’s an example function that takes in three parameters for running rbinom, and returns the output of rbinom.

binom\_sim <- function(myN, mySize, myProb) {  
 output <- rbinom(n = myN, size = mySize, prob = myProb)  
 return(output)  
}

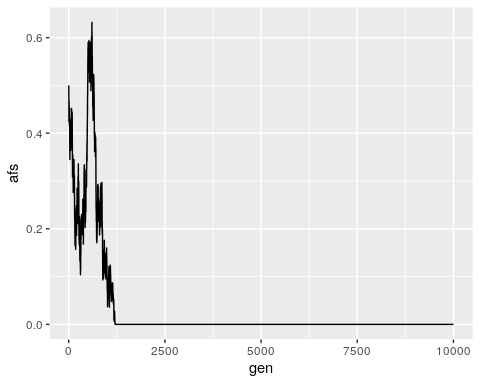
We want our function to take in parameters for the starting allele frequency, population size, and number of generations to simulate. It should return the sim\_results dataframe so that we can plot the allele frequency trajectory.

To write a function, we can place the code that we just wrote into the function body:

run\_sim <- function(Ne, freq, generations) {  
   
 # note how we don't define our initial parameters for Ne, freq, etc.  
 # because we're passing in those parameters as arguments  
   
 freq\_vector <- freq  
 for (i in 1:generations) {  
 new\_freq <- rbinom(n = 1, size = Ne, prob = freq) / Ne  
 freq\_vector <- c(freq\_vector, new\_freq)  
 freq <- new\_freq  
 }  
   
 # convert vector of AFs into a tibble for plotting  
 sim\_results <- tibble(afs = freq\_vector,  
 gen = 1:(generations+1))  
   
 # return the tibble of AFs, so that we can access the results  
 return(sim\_results)  
}

Now we can run the function with parameters of our choosing and plot the output:

# run function  
results <- run\_sim(Ne = 1000, freq = 0.5, generations = 10000)  
  
# plot output  
ggplot(data = results,  
 aes(x = gen, y = afs)) +  
 geom\_line()



**Exercise**: Run your run\_sum function a few times with different input population sizes and allele frequencies. How does changing thesse inputs affect the allele frequency trajectories that you see?

How do I know when to use a function?

Functions are useful whenever you have code that you want to run multiple times with slightly different parameters. If you find yourself copying over code several times and changing just a few things, you should consider writing a function instead.

## 2.8 Conclusion

In this lab, we’ve successfully built a Wright-Fisher simulation for one allele, allowing us to track how we expect its frequency to change over time under the principles of genetic drift.

This simple simulation forms the core of most models used in evolutionary genetics research. For example, the evolutionary simulation software [SLiM](https://messerlab.org/slim/) extends our model to perform Wright-Fisher simulations with multiple alleles simultaneously, allowing us to reconstruct complex phenomena like balancing selection (**Fig. 2**).

Fig. 2. Simulating balancing selection with SLiM. Each vertical line represents a SNP, with height corresponding to the SNP’s current frequency. As the simulation progresses, two distinct haplotypes of mutations begin to rise and fall in frequency together, but neither fully fixes because balancing selection favors intermediate frequencies.

Figure : **Fig. 2. Simulating balancing selection with SLiM.** Each vertical line represents a SNP, with height corresponding to the SNP’s current frequency. As the simulation progresses, two distinct haplotypes of mutations begin to rise and fall in frequency together, but neither fully fixes because balancing selection favors intermediate frequencies.

## 2.9 Homework

#### 2.9.0.1 Goals & Learning Objectives

The goal of this homework is to explore different extensions of the Wright-Fisher model.

**Learning Objectives**

* Required homework: Practice visualizing allele frequencies with ggplot
* Extra credit: Practice writing functions and interpreting allele frequency trajectories

### 2.9.1 Required

Currently, our allele frequency trajectory plot only shows the AF of one allele at a locus. (i.e., if individuals in a population have either an A or a C at some locus, we’re only plotting the trajectory of the A allele.)

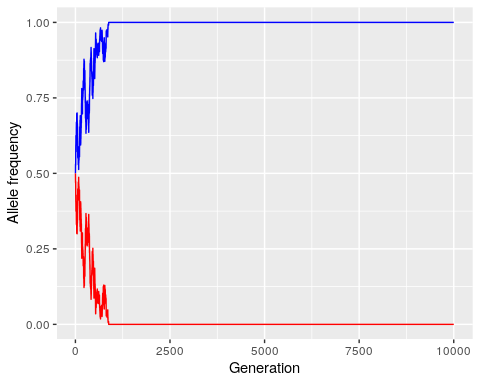
**Assignment:** Add a line to allele frequency trajectory plot that shows the frequency of the other allele at the locus. Give the two alleles different colors.

Hint

You don’t need to modify the simulation function for this.

Solution

# run the simulation again to get output data  
sim <- run\_sim(Ne = 1000, freq = 0.5, generations = 10000) %>%  
 # add in a column with the AF of the minor allele  
 # this is 1 - AF of the major allele  
 dplyr::mutate(afs\_minor = 1 - afs)  
  
ggplot() +  
 geom\_line(data = sim, aes(x = gen, y = afs), color = "blue") +  
 # add in another line with the frequency of the minor allele  
 geom\_line(data = sim, aes(x = gen, y = afs\_minor), color = "red") +  
 ylim(0, 1) +  
 ylab("Allele frequency") +  
 xlab("Generation")



### 2.9.2 Extra credit: Selection

One way to extend our simple Wright-Fisher model is to add in selection as a parameter. Selection affects our model by altering the probability of sampling our allele of interest each generation (e.g., positive selection increases the probability, and negative selection decreases it).

Previously, we assumed that this probability was equivalent to the allele’s frequency, or , where is the population size and is the number of individuals who carry the allele.

For the purposes of this homework, we assume that in a model with selection, this probability is instead:

where is the **selection coefficient**, and ranges from -1 to 1.

**Question**: What does this probability become in the absence of selection (i.e., when )?

The probability becomes , which is the same as the allele frequency.

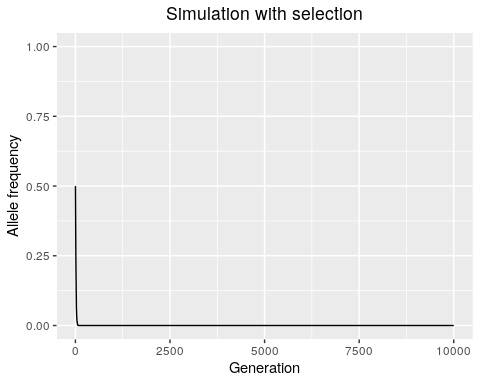
**Assignment:** Modify your run\_sim function so that it takes in a selection coefficient s as a parameter. Run the simulation a few times with and without (s = 0) selection, but keeping other parameters the same (Ne = 10000 – to mimic the Ne of humans, freq = 0.5, generations = 10000). What do you notice about the allele frequency trajectories?

Solution

run\_sim\_selection <- function(Ne, freq, generations, s) {  
   
 freq\_vector <- freq  
 for (i in 1:generations) {  
 # calculate p, the probability of sampling the allele, based on s  
 i <- freq \* Ne # number of individuals who currently carry the allele  
 p <- i\*(1+s) / (Ne - i + i\*(1+s))  
   
 # prob is now `p`, rather than `freq`  
 new\_freq <- rbinom(n = 1, size = Ne, prob = p) / Ne  
 freq\_vector <- c(freq\_vector, new\_freq)  
 freq <- new\_freq  
 }  
   
 # convert vector of AFs into a tibble for plotting  
 sim\_results <- tibble(afs = freq\_vector,  
 gen = 1:(generations+1))  
   
 # return the tibble of AFs, so that we can access the results  
 return(sim\_results)  
}

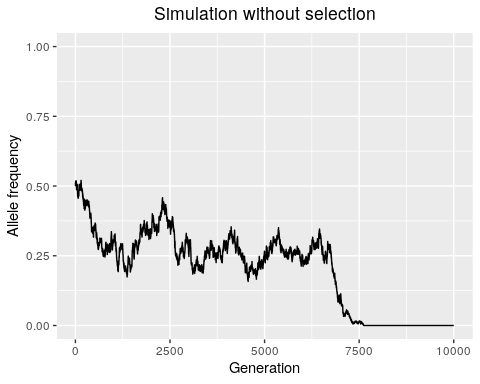
Run and plot the simulation with selection:

results <- run\_sim\_selection(Ne = 10000,  
 freq = 0.5,  
 generations = 10000,  
 s = -0.1)  
ggplot() +  
 geom\_line(data = results, aes(x = gen, y = afs)) +  
 ylim(0, 1) +  
 ylab("Allele frequency") +  
 xlab("Generation") +  
 ggtitle("Simulation with selection") +  
 theme(plot.title = element\_text(hjust = 0.5)) # to center the title



Run and plot the simulation without selection:

results <- run\_sim\_selection(Ne = 10000,  
 freq = 0.5,  
 generations = 10000,  
 s = 0)  
ggplot() +  
 geom\_line(data = results, aes(x = gen, y = afs)) +  
 ylim(0, 1) +  
 ylab("Allele frequency") +  
 xlab("Generation") +  
 ggtitle("Simulation without selection") +  
 theme(plot.title = element\_text(hjust = 0.5)) # to center the title



We observe that selection (at least, strong selection where s = 0.1) tends to decrease the time it takes for an allele to either fix or go extinct. This is because selection directionally biases the probability of sampling that allele.

Decreasing the absolute value of the selection coefficient will make the simulation behave more like drift - most selection coefficients are thought to be [very small](https://academic.oup.com/view-large/figure/326984394/345fig4.jpeg), and the largest known selection coefficients in humans are around [0.05](https://elifesciences.org/articles/63177).

# About the Authors

| Credits | Names |
| --- | --- |
| **Pedagogy** |  |
| Instructor | [Rajiv McCoy](https://mccoy-lab.org/) |
| Content Author | [Kate Weaver](https://kweav.github.io/) |
| Content Author | [Stephanie Yan](https://stephaniemyan.github.io/) |
| **Funding** |  |
| JHU Center for Educational Resources | Techology Fellowship Grant |

## ─ Session info ───────────────────────────────────────────────────────────────  
## setting value   
## version R version 4.0.2 (2020-06-22)  
## os Ubuntu 20.04.3 LTS   
## system x86\_64, linux-gnu   
## ui X11   
## language (EN)   
## collate en\_US.UTF-8   
## ctype en\_US.UTF-8   
## tz Etc/UTC   
## date 2022-12-21   
##   
## ─ Packages ───────────────────────────────────────────────────────────────────  
## package \* version date lib source   
## assertthat 0.2.1 2019-03-21 [1] RSPM (R 4.0.3)   
## bookdown 0.24 2022-02-15 [1] Github (rstudio/bookdown@88bc4ea)   
## callr 3.4.4 2020-09-07 [1] RSPM (R 4.0.2)   
## cli 2.0.2 2020-02-28 [1] RSPM (R 4.0.0)   
## crayon 1.3.4 2017-09-16 [1] RSPM (R 4.0.0)   
## desc 1.2.0 2018-05-01 [1] RSPM (R 4.0.3)   
## devtools 2.3.2 2020-09-18 [1] RSPM (R 4.0.3)   
## digest 0.6.25 2020-02-23 [1] RSPM (R 4.0.0)   
## ellipsis 0.3.1 2020-05-15 [1] RSPM (R 4.0.3)   
## evaluate 0.14 2019-05-28 [1] RSPM (R 4.0.3)   
## fansi 0.4.1 2020-01-08 [1] RSPM (R 4.0.0)   
## fs 1.5.0 2020-07-31 [1] RSPM (R 4.0.3)   
## glue 1.6.1 2022-01-22 [1] CRAN (R 4.0.2)   
## htmltools 0.5.0 2020-06-16 [1] RSPM (R 4.0.1)   
## knitr 1.33 2022-02-15 [1] Github (yihui/knitr@a1052d1)   
## lifecycle 1.0.0 2021-02-15 [1] CRAN (R 4.0.2)   
## magrittr 2.0.2 2022-01-26 [1] CRAN (R 4.0.2)   
## memoise 1.1.0 2017-04-21 [1] RSPM (R 4.0.0)   
## pkgbuild 1.1.0 2020-07-13 [1] RSPM (R 4.0.2)   
## pkgload 1.1.0 2020-05-29 [1] RSPM (R 4.0.3)   
## prettyunits 1.1.1 2020-01-24 [1] RSPM (R 4.0.3)   
## processx 3.4.4 2020-09-03 [1] RSPM (R 4.0.2)   
## ps 1.3.4 2020-08-11 [1] RSPM (R 4.0.2)   
## purrr 0.3.4 2020-04-17 [1] RSPM (R 4.0.3)   
## R6 2.4.1 2019-11-12 [1] RSPM (R 4.0.0)   
## remotes 2.2.0 2020-07-21 [1] RSPM (R 4.0.3)   
## rlang 0.4.10 2022-02-15 [1] Github (r-lib/rlang@f0c9be5)   
## rmarkdown 2.10 2022-02-15 [1] Github (rstudio/rmarkdown@02d3c25)  
## rprojroot 2.0.2 2020-11-15 [1] CRAN (R 4.0.2)   
## sessioninfo 1.1.1 2018-11-05 [1] RSPM (R 4.0.3)   
## stringi 1.5.3 2020-09-09 [1] RSPM (R 4.0.3)   
## stringr 1.4.0 2019-02-10 [1] RSPM (R 4.0.3)   
## testthat 3.0.1 2022-02-15 [1] Github (R-lib/testthat@e99155a)   
## usethis 2.1.5.9000 2022-02-15 [1] Github (r-lib/usethis@57b109a)   
## withr 2.3.0 2020-09-22 [1] RSPM (R 4.0.2)   
## xfun 0.26 2022-02-15 [1] Github (yihui/xfun@74c2a66)   
## yaml 2.2.1 2020-02-01 [1] RSPM (R 4.0.3)   
##   
## [1] /usr/local/lib/R/site-library  
## [2] /usr/local/lib/R/library

# 3 References

*Molecular Population Genetics*, Matthew Hahn (Oxford University Press, 2019)