Syllabus

November, 2023

Table of Contents

# 1 Population Genetics: Simulation and Visualization

Online reference used in AS.360.111.

# 2 Syllabus: Population Genetics Simulation & Visualization

**Course Number:** AS.360.111 **Schedule**: Fall 2023, Tues 2:30 PM - 4:30 PM **Location** Homewood Campus, UG Teaching Lab (UTL) 289 **Student Hours**: by appointment, please email both instructors; Levi 200

**Instructors:** Andrew Bortvin [abortvi2@jh.edu](mailto:abortvi2@jh.edu) Sara Carioscia [saracarioscia@jhu.edu](mailto:Saracarioscia@jhu.edu)

### 2.0.1 Course Description

Dive into the fascinating world of computational evolutionary biology with our course on “Population Genetics Simulation and Visualization.” This hands-on course is designed to equip students with the tools and knowledge needed to understand and analyze complex evolutionary dynamics using the SLiM (Selection, Linkage, and Mutation) simulation software. Through a blend of theoretical lectures and practical sessions, students will learn how to create and manipulate virtual populations, simulate genetic drift, natural selection, and other evolutionary forces. They will gain proficiency in setting up simulation scenarios, running experiments, and collecting raw data. Leveraging this data, students will explore various data visualization techniques to uncover patterns, trends, and insights in the simulated evolutionary processes. Prior programming experience is not required. Students from all departments and at all levels (including first-year undergraduates) are welcome.

### 2.0.2 Class Structure

**Class Organization** Each class will be divided between lectures covering biological principles, live coding where we teach programming in SLiM, and in-class completion of assignments. As this is a longer class, we will take breaks between sections. Feel free to bring snacks or drinks.

**Assignments** Each day, we will reserve time to work on in-class assignments, which will primarily focus on implementation of simulations and conceptual questions regarding the theory behind population genetics. Assigments will also include additional (optional) questions for students wishing to further develop their models.

Please submit all files via email to both instructors. Please save your .slim files as .txt (export and save as) and your resulting plots as .png files.

You are welcome to work together in small groups, and collaboration is encouraged. Likewise, we encourage you to seek answers online when encountering. However, please refrain from just copying someone else’s code – you should understand and be able to explain every line of code in your scripts.

### 2.0.3 Schedule

| Class Number | Date | Topics Covered |
| --- | --- | --- |
| 1 | 10/31/23 | Intro to SLiM + molecular biology |
| 2 | 11/7/23 | TBD |
| 3 | 11/14/23 | TBD |
| 4 | 11/21/23 | TBD |
| 5 | 11/28/23 | TBD |
| 6 | 12/5/23 | TBD |

### 2.0.4 Grading

This course will be graded Satisfactory/Unsatisfactory, and individual assignments will be graded on reasonable completion (rather than accuracy of results). In each assignment, we will specify items to include in your submission; the assignment score will be the fraction of items completed. To achieve a Satisfactory, you must have an average of 70% completion across the assignments.

All assignments will be due at the end of the course period. However, please try to submit your work each week - we will look at your work in progress, provide written feedback, discuss any questions or opoprtunities for improvement, and let you know estimated percent completion.

### 2.0.5 Student Hours

We will host open student hours (time for students to chat about anything! biology, coding, graduate school, materials directly or slightly less directly related to the course, actual assignments) through the end of the semester. Please email both instructors to schedule a time for either in-person or Zoom.

# 3 SLiM Guide

Notes on usage of SLiM for evolutionary modeling. Adapted from the SLiM manual: <http://benhaller.com/slim/SLiM_Manual.pdf>

## 3.1 A Basic SLiM Simulation

Our first simulation will be a lightly modified version of the script that comes up when we first open the SLiM graphical user interface (GUI). It is a simple simulation tracking one population, with no selection:

// set up a simple neutral simulation  
initialize() {  
 initializeMutationRate(1e-7);  
   
 // m1 mutation type: neutral  
 initializeMutationType("m1", 0.5, "f", 0.0);  
   
 // g1 genomic element type: uses m1 for all mutations  
 initializeGenomicElementType("g1", m1, 1.0);  
   
 // uniform chromosome of length 100 kb with uniform recombination  
 initializeGenomicElement(g1, 0, 99999);  
 initializeRecombinationRate(1e-8);  
}  
  
// create a population of 500 individuals  
1 early() {  
 sim.addSubpop("p1", 500);  
}  
  
2000 late() {   
 sim.outputFixedMutations();   
}

## 3.2 Initliazing a Simulation

The first step in running a simulation is defining all of the parameters that characterize our populations - how many populations are we studying? How big are they? What do individual genomes look like? We use an initialize() block to define any parameters like this.

In almost every simulation, we will define a core set of common parameters. They are:

### 3.2.1 Global Mutation Rates

The mutation rate is set by the command initializeMutationRate(). In its simplest form, we run this function with a single argument, the mutation, as such: initializeMutationRate(1e-7);. Now, when executing the simulation, SLiM will go through each gamete base by base, and will introduce a mutation with a probability of 1e-7.

### 3.2.2 Mutation Types

When running our simulation, we can distinguish between multiple kinds of mutations, each with their own prevalence and impact - for example, a neutral substitution, or a lethal deletion, or a rare variant that confers a selective advantage. To create a mutation type, we use the initializeMutationType() command. For example, this following line of code creates a deleterious mutation:

initializeMutationType("m1", 0.5, "f", -0.02);

Let’s break the arguments to initializeMutationType down:

* the **mutation id**: this can be any integer, or a string in the format m<integer>. This is the name that you will use to keep track of your mutation.
* the **dominance coefficient**: Here, 0.5. This is used when determining how a mutation impacts the fitness of your individual. A mutation with a dominance coefficient of 1.0 is completely dominant; a mutation with a dominance coefficient of 0.0 is recessive. Mutations with values in between represent incomplete dominance - in our case, 0.5 indicates that heterozygotes have half the fitness effect of the mutation
* the last two arguments, "f" and -0.02 denote the **distribution** and **magnitude** of **how our mutation impacts fitness**. f indicates that the effect is *fixed* - that is, the mutation impacts all affected individuals identically. The value -0.02 indicates that this effect is a fitness reduction of 0.02 - any individual homozygous for this allele is expected to have offspring at 0.98 the frequency of an individual without this allele, all else being equal. While this allele has the same effect on all carriers, we can also create alleles with different fitness effect distributions - for example "n" indicates that for each individual with this mutation, we draw the fitness effect from a normal distribution, "e" indicates that we draw fitness effects from the exponential distribution. For some of these, e.g. the normal distribution, we may need to provide more than one numerical parameter.

### 3.2.3 Types of DNA

Now that we have defined the types of mutations we can encounter in our simulation, we can no specify the types of genomic regions that we are working with. For example, we can create genomic regions that represent telomeres, centromeres, introns, exons, etc. And intuitively, we can anticipate that mutations are likely to have different effects in different regions (for example, a SNP in an exon is more likely to have a dramatic phenotypic impact than a SNP in a centromere).

We can define a type of DNA using the initializeGenomicElementType() function. This function takes three argument, in the following order:

* the **name** of the genomic region (in the form "g" + an integer)
* the **kinds of mutations** that can occur in this type of DNA region.
* the **relative frequencies** of differnet mutation types

Here is a simple example of this command:

initializeGenomicElementType("g1", m1, 1.0);

This creates a type of DNA region called "g1", which can only have m1 type mutations. Since there is only one type of mutation, it happens at a frequency of 1.

### 3.2.4 Genomic Regions

Now that we have definied a type of DNA that we can see in our genome, we need to great a specific instance of it. This is done with the initializeGenomicElement() command. This takes as arguments:

* the *type of genomic element*
* the *start coordinate*
* the *stop coordinate*

For example:

initializeGenomicElement(g1, 0, 99999);

This creates a genomic element of type g1, which extends from bases 0 to 99999.

#### 3.2.4.1 Recombination Rates

Lastly, we need to set the recombination rate. This is done using the initializeRecombinationRate() function, which functions similarly to how we defined mutation rates. For this function, we just provide a float defining the per base recombination rate:

initializeRecombinationRate(1e-8);

## 3.3 Running a simulation

Now that we have initialized a simulation, we can define the events that happen during our simulation. Our simulation is organized into generations (used interchangeably with tick cycles), and within each generation, a predefined set of events happen in a fixed order:

1. If you have defined any events as occurring 'early' in a generation, they are executed first
2. Offspring are generated
3. Fixed mutations are removed in the offspring - that is, if all individuals have two copies of a mutation, SLiM stops keeping track of it
4. Parents die, offspring become the new parents
5. If you have defined any events as occurring 'late' in a generation, they are now executed

### 3.3.1 Adding subpopulations

We have used the initialize block to precisely define how the DNA of all of our organisms is structured. Now, we can create a subpopulation. This is conventionaly done as an early event in the first generation (that is, this will be the first thing that happens in your simulation). For example:

1 early()  
{  
 sim.addSubpop("p1", 500);  
}

A few things to note here:

* The first line (1 early()) defines **when** this event occurs. The 1 indicates that this will happen in the first generation. The early() indicates that this is an early event in the generation (i.e. it happens before generation of offspring)
* The command sim.addSubpop("p1", 500); creates a subpopulation with the name "p1", which contains 500 individuals.

### 3.3.2 Events that occur every generation

The above sim.addSubpop() command was only executed in the first generation. What if we want an event that runs every generation? To do this, we simply create an event without specifying a generation number. For example, let’s say that in each generation, we want to print the list of all mutations that have reached fixation. We can do this with the following command:

late() {  
 sim.outputFixedMutations();  
}

In the first line of this command (late()), we do not specify the generation at which this happens. As a result, it will happen as a late event in *every* generation.

## 3.4 Ending a simulation

We can explicitly end a simulation using the command sim.simulationFinished().

10000 late()  
{  
 sim.simulationFinished();  
}

This simply ends the simulation. Typically, we want to output something for the user. We’ll go into more options throughout the course. For now, know that we can output a sample of our population using the following syntax:

10000 late()  
{  
 p1.outputSample(500);  
}

where 500 is the number of individuals to output from population p1. This outputs a lot of information that is typically sent to a downstream application.

A simpler output is to use the function sim.outputFixedMutations(), which outputs a list of all mutations that have reached fixation (are present in two copies in all individuals).

10000 late()  
{  
 sim.outputFixedMutations();  
}

## 3.5 Multiple Mutations, Elements and Regions

So far, we have looked at simulations where we only have one mutation type, one genomic element type, and one genomic region. Now, let’s look at how to create multiple mutation types and genomic regions.

Here is a simulation that creates three mutation types, each with their own fitness effects

initialize(){  
 initializeMutationRate(1e-7);  
   
 // Create mutation types  
 initializeMutationType("m1", 1.0, "f", 0.0); // Neutral mutation  
 initializeMutationType("m2", 1.0, "f", 0.02); // favorable mutation  
 initializeMutationType("m3", 1.0, "f", -0.01); // Deleterious mutation   
   
 // Create Types of DNA   
 initializeGenomicElementType("g1", c(m1, m2, m3), c(50, 1, 10));   
 initializeGenomicElementType("g2", m1, 1);   
   
 // Arrange DNA into a genome  
 initializeGenomicElement(g1, 0, 9999);  
 initializeGenomicElement(g2, 10000, 15000);  
 initializeGenomicElement(g1, 15001, 25000);  
   
 initializeRecombinationRate(1e-8);  
}  
  
1 early(){  
 sim.addSubpop("p1", 500);  
}  
  
10000 early()  
{  
 sim.outputFixedMutations();  
}

### 3.5.1 Making multiple mutations

We can make multiple mutation types using repeated calls to the initializeMutationType() statement. From the script above:

initializeMutationType("m1", 1.0, "f", 0.0); // Neutral mutation  
initializeMutationType("m2", 1.0, "f", 0.02); // favorable mutation  
initializeMutationType("m3", 1.0, "f", -0.01); // Deleterious mutation

Notice that each mutation type has its own name. The three mutations represent neutral, favorable, and deleterious mutations, respectively. These effects are controlled by the fourth argument (0.0, 0.0, -0.01).

### 3.5.2 Genomic Elements with multiple mutations

We can create a genomic element with multiple mutation types. Consider the line below:

initializeGenomicElementType("g1", c(m1, m2, m3), c(50, 1, 10));

Notice that for the mutation type, we can provide multiple mutation types using the syntax: c(m1, m2, m3). Likewise, we now set the ratio of how often they appear in the third argument, c(50, 1, 10).

This essentially says that:

* Genomic element "g1" can have mutations of type m1, m2, and m3
* The chance of an m1 mutation is 50/61. The chance of an m2 mutation is 1/61. The chance of an m3 mutation is 10/61.

Note that the order of mutation types within c(m1, m2, m3) is arbitrary, but it needs to match the order of their frequencies in c(50, 1, 10). The following two lines would work identically:

initializeGenomicElementType("g1", c(m1, m2, m3), c(50, 1, 10)); initializeGenomicElementType("g1", c(m3, m1, m2), c(10, 50, 1));

### 3.5.3 Multiple Genomic Elements Types

We can create multiple genomic element types with multiple uses of initializeGenomicElementType(). From the script above:

initializeGenomicElementType("g1", c(m1, m2, m3), c(50, 1, 10));   
initializeGenomicElementType("g2", m1, 1);

Note that "g2" type genomic elements can only have mutations of type m1, despite multiple mutation types being defined.

### 3.5.4 Multiple Genomic Elements

Now that we have defined multiple genomic element types, we can initialize multiple genomic elements with repeated uses of initializeGenomicElement():

initializeGenomicElement(g1, 0, 9999);  
initializeGenomicElement(g2, 10000, 15000);  
initializeGenomicElement(g1, 15001, 25000);

This creates a genome consisting of a stretch of 10,000 base long stretch of g1-type DNA, a 5,000 base long stretch of g2-type DNA, and a second 10,000 base stretch of g1-type DNA again.

To summarize:

* We have defined three types of mutations with three different fitness effects
* We have defined two types of DNA. One of them can have mutations of type m1, m2, and m3. The other can only have mutations of type m1
* We have arranged the two types of DNA into a genome, in the order g1, g2, g1

### 3.5.5 SLiM GUI

A few interesting things to note about the SLiM GUI in this simulation:

Looking at the genomic region viewer, it is now color-coded to reflect the arrangement of multiple genomic regions of different types (dark and light blue).

Looking at the mutation tracker, we can see different mutation types color-coded by their fitness effects - yellow for neutral mutations, green for favorable mutations, and red for deleterious mutations. Note that in the dark blue region (g2), there are only neutral mutations. This agrees with our setup of the genome, as g2 can only have mutations of type m1.

## 3.6 Week 1 Assignment

### 3.6.1 1.1 - Population Size

A key concept in population genetics is **genetic drift**, or the random fluctuation of allele frequencies from generation to generation. In our neutral simulations, genetic drift is the **only** force that causes alleles to become more or less common with each generation.

The extent that genetic drift can influence allele frequencies is dependent on the population size. We can use our neutral simulation to look further investigate this relationship. Pick a range of population sizes, and run your simulation at least 5 times for each population size. Record the number of fixed mutations for each population size. Briefly comment on how the population size is related to the time to fixation.

### 3.6.2 1.2 - Selection

So far, we focused on simulations where all mutations were neutral. Change the selection coefficient of your mutations. Try making your mutation slightly favorable. How does the number of fixed mutations change when you add favorable mutations? How low a selection coefficiet do you need for the effect to be visible?

Do the same with a slightly deleterious mutation.

### 3.6.3 1.3 - Multiple mutation types; multiple genomic regions

Create two mutation types, one of which is highly (perhaps a selection coefficient around ~0.05) and one of which is deleterious (selection coefficient ~-0.01).

Create a genomic region with both of these mutation types. Recall that we can use the syntax c(m1, m2) to provide multiple items together to an argument. For example, we can initialize a genomic element type with two mutation types at relative frequencies of 0.2 and 0.8 as such: initializeGenomicElementType("g2", c(m1, m2), c(0.2, 0.8));.

Run this a few times and look to see if your deleterious mutation ever reaches fixation. How could you explain this behavior? Does it still happen if you raise your recombination rate higher (e.g. a biologically unrelatstic value of 1e-2)? Why or why not?

Note that the following code block will print the number of fixed mutations of type m2:

2000 late() {   
 print('Number of type 2 fixed mutations:');  
 print(sum(sim.substitutions.mutationType == m2));  
}

## 3.7 Storing Information

In SLiM, we have many different ways of storing information. The optimal way of storing information depends on two questions:

* How long do I need to retain this information?
* Will I need to modify this information?

### 3.7.1 Local variables

Local variables are **temporary** and **mutable**. We can define a local variable using the = operator. For example, in the following lines, I save the current number of chromosomes in my population to a variable nChr and the print it to screen:

400 late(){  
 nChr = size(p1.genomes); // Define number of chromosomes  
 print(nChr);   
}

We can redfine a local variable using the = operator. For example, here we refined nChr to be 3 times nChr:

400 late(){  
 nChr = size(p1.genomes); // Define number of chromosomes  
 nChr = nChr \* 3; // Multiply by 3 and store value   
 print(nChr);   
}

One key feature of local variables is that once you leave the event your are in, **local variables are removed from memory**. (Recall that an event is considered anything surrounded by curly braces {}). Consider the following example:

400 late(){  
 nChr = size(p1.genomes); // Define number of chromosomes  
 nChr = nChr \* 3; // Multiply by 3 and store value   
}  
  
400 late() {  
 print(nChr); // This will cause an error   
}

Running the above code will generate an error. In the first 400 late() event we define nChr and update it’s value to be 3 times itself. In the second 400 late() event, we try to print nChr, but after the first event was run, nChr was deleted and no longer exists. This therefore causes an error. Having = define local variables is a somewhat quirky, SLiM-specific behavior.

### 3.7.2 Constants

Constants are **permanent** and **immutable**. Once a constant is defined, it will remain in memory for the remainder of the simulation, but we will not be able to change its value. We can create constants using the defineConstant() function. This function takes two arguments - the name of our constant, and it’s value.

For example imagine, that we are running multiple iterations of a simulation, and we want to keep track of the trial number. A constant is a good data type to use here, because the trial number will not change within a simulation.

1 early(){  
 defineConstant('trialNumber', 4);  
}  
  
1 early(){  
 print(trialNumber);  
}

This works! A constant is saved for the entirety of your simulation, beyond the current event.

Note that we **can not** modify the value of a constant. The following results in an error:

1 early(){  
 defineConstant('trialNumber', 4);  
 trialNumber = trialNumber + 1;  
}

### 3.7.3 Simulation Values

After the simulation is initialized, we can use the name sim to refer to the entire simulation. We can also attach **values** to this simulation; each value is **permanent** and **mutable**.

We can define a new value using the function: sim.setValue(). The syntax for this function is identical to defineConstant() - we provide a name and associated value.

We can refer to an existing simulation value using the function sim.getValue(), which just takes the name of your value.

For example, in the code below, we define and print a new value initially set to 0 at generation 400.

400 early(){  
 sim.setValue("counter", 0); // define counter to be 0  
 print(sim.getValue("counter"));  
}

Now, in the following event, we update the counter by adding 1 to it in every generation between generations 401 and 500.

401:500 early(){  
 currentGen = sim.getValue("counter"); // get the current value of counter  
 sim.setValue("counter", currentGen + 1); // increment the counter by 1  
 print(sim.getValue("counter")); // print current value of counter  
}

Note that prior to updating counter, we first set a local variable, currentGen equal to the value of counter. This makes the code more readable, but is not necessary; a more concise form of the above event is:

401:500 early(){  
 sim.setValue("counter", sim.getValue("counter") + 1); // increment the counter by 1  
 print(sim.getValue("counter")); // print current value of counter  
}

This functions identically and is one line shorter, at the cost of a more confusing call to sim.setValue().

## 3.8 Introducing Sweep Mutations

This week, we are simulating **selective sweeps**, or events in which a genetic variant rises rapidly in frequency due to positive selection.

We want to mimic a so-called “soft sweep”, or a scenario in which an existing genetic variant becomes favorable (e.g. because of changing environmental pressures).

In practice, a simple way of doing this is to randomly introduce a favorable mutation at a set frequency at a pre-specified generation. This process can be divided into a few sequential steps:

1. Define the mutation type for your sweep
2. At a given generation, sample a predetermined of your individuals
3. To sampled individuals, add the mutation at a predetermined genomic position

### 3.8.1 Defining the Mutation

Let’s start with a familiar initialize statement, where we have one neutral mutation type and one genomic region.

initialize(){  
 initializeMutationRate(1e-7);  
 // Create mutation types  
 initializeMutationType("m1", 0.5, "f", 0.0);  
   
 // Create Types of DNA   
 initializeGenomicElementType("g1", m1, 1);   
   
 // Arrange DNA into a genome  
 initializeGenomicElement(g1, 0, 9999);  
  
 initializeRecombinationRate(1e-8);  
}

To introduce our favorable mutation, let’s add the following to our initialize statement:

initializeMutationType("m2", 0.5, "f", 0.015); m2.mutationStackPolicy="l"; m2.convertToSubstitution = F;

The first line should be familiar - we have a mutation type called "m2", which has a positive fitness effect of 0.015.

The second line, m2.mutationStackPolicy="l"; controls what happens when multiple mutations occur at the same position. By default, SLiM allows multiple mutations at the same position, with additive fitness effects. Since we are modeling a single base substitution, we want to only track the most recent mutation at a given position. setting the mutation stack policy to "l" tells SLiM to only care about the last mutation at a given site.

The final line, m2.convertToSubstitution = F;, controls how our mutation behaves when it reaches fixation. By default, once a mutation reaches fixation, it is converted to a **substitution** - that is, SLiM stops tracking the mutation and only keeps a log of the coordinate where the mutation happened. This is because once the variant is fixed, it has the same impact on all individuals. Here, we want to disable this behavior, since we want to track this specific variant throughout the entire simulation.

### 3.8.2 Sampling individuals

Let’s say that we want to introduce this mutation into our population with an allele frequency of 5%, at position 1000. The following code block does this.

400 late(){  
 p1\_size = size(p1.genomes); // Count how many chromosomes we have   
 initial\_freq = asInteger(p1\_size \* 0.05); // Find 5% of total number of chromosomes  
 target = sample(p1.genomes, initial\_freq); // Randomly sample 5% of chromosomes  
 target.addNewDrawnMutation(m2, 1000); // Add new mutation  
}

Let’s break this down line by line:

* p1\_size = size(p1.genomes);: p1.genomes is a list of all the chromosomes in our population. The size() function gets the length of this list, i.e. the total number of chromosomes. We save this as a local variable, p1\_size.
* initial\_freq = asInteger(p1\_size \* 0.05);. We multiply our number of chromosomes by 0.05 to get the number of chromosomes into which we want to insert the mutation. Depeding on our population size, we might get a non-integer number. Since we can’t add a mutation into, say, 3.2 chromosomes, we convert to an integer, rounding down, using the asInteger() function.
* target = sample(p1.genomes, initial\_freq); The sample() function randomly samples from a set of items. It takes two arguments - what we are sampling from and how many items to sample. This essentially says, we should sample 5% of our genomes.
* target.addNewDrawnMutation(m2, 1000);: The addNewDrawnMutation() function takes two arguments, the mutation type to add, and the genomic position to add it to. Note that we call the function as target.addNewDrawnMutation() - the target here indicates the object we are operating on.

All together, our simulation may look like this:

initialize(){  
 initializeMutationRate(1e-7);  
 // Create mutation types  
 initializeMutationType("m1", 0.5, "f", 0.0);  
 initializeMutationType("m2", 1, "f", 0.015);  
 m2.mutationStackPolicy="l";  
 m2.convertToSubstitution = F;  
   
 // Create Types of DNA   
 initializeGenomicElementType("g1", m1, 1);   
   
 // Arrange DNA into a genome  
 initializeGenomicElement(g1, 0, 9999);  
  
 initializeRecombinationRate(1e-8);  
}  
  
1 early(){  
 sim.addSubpop("p1", 500);  
}  
  
400 late(){  
 p1\_size = size(p1.genomes); // Count how many chromosomes we have   
 initial\_freq = asInteger(p1\_size \* 0.05); // Find 5% of total number of chromosomes  
 target = sample(p1.genomes, initial\_freq); // Randomly sample 5% of chromosomes  
 target.addNewDrawnMutation(m2, 1000); // Add new mutation  
}  
  
2000 late() { sim.outputFixedMutations(); }

# 4 Lecture Slides

* [Week 1](https://docs.google.com/presentation/d/1xdNvB5BIjlm1EAbDGl9JsDbymq2KxQrbLJK9qVbEylE/edit?usp=sharing)

# About the Authors

These credits are based on our [course contributors table guidelines](https://www.ottrproject.org/more_features.html#giving-credits-to-contributors).

| Credits | Names |
| --- | --- |
| **Pedagogy** |  |
| Content Author | Andrew Bortvin |
| Content Author | Sara Carioscia |
| **Website** |  |
| Template | Jeff Leek & The Johns Hopkins Data Science Lab |
| Design Inspiration | Stephanie Yan & Ali Madooei & JHU Data Structures |

## ─ Session info ───────────────────────────────────────────────────────────────  
## setting value   
## version R version 4.0.2 (2020-06-22)  
## os Ubuntu 20.04.5 LTS   
## system x86\_64, linux-gnu   
## ui X11   
## language (EN)   
## collate en\_US.UTF-8   
## ctype en\_US.UTF-8   
## tz Etc/UTC   
## date 2023-11-07   
##   
## ─ Packages ───────────────────────────────────────────────────────────────────  
## package \* version date lib source   
## assertthat 0.2.1 2019-03-21 [1] RSPM (R 4.0.5)   
## bookdown 0.24 2023-03-28 [1] Github (rstudio/bookdown@88bc4ea)   
## cachem 1.0.7 2023-02-24 [1] CRAN (R 4.0.2)   
## callr 3.5.0 2020-10-08 [1] RSPM (R 4.0.2)   
## cli 3.6.1 2023-03-23 [1] CRAN (R 4.0.2)   
## 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 0.20 2023-01-17 [1] CRAN (R 4.0.2)   
## fansi 0.4.1 2020-01-08 [1] RSPM (R 4.0.0)   
## fastmap 1.1.1 2023-02-24 [1] CRAN (R 4.0.2)   
## fs 1.5.0 2020-07-31 [1] RSPM (R 4.0.3)   
## glue 1.4.2 2020-08-27 [1] RSPM (R 4.0.5)   
## hms 0.5.3 2020-01-08 [1] RSPM (R 4.0.0)   
## htmltools 0.5.5 2023-03-23 [1] CRAN (R 4.0.2)   
## jsonlite 1.7.1 2020-09-07 [1] RSPM (R 4.0.2)   
## knitr 1.33 2023-03-28 [1] Github (yihui/knitr@a1052d1)   
## lattice 0.20-41 2020-04-02 [2] CRAN (R 4.0.2)   
## lifecycle 1.0.3 2022-10-07 [1] CRAN (R 4.0.2)   
## magrittr 2.0.3 2022-03-30 [1] CRAN (R 4.0.2)   
## Matrix 1.2-18 2019-11-27 [2] CRAN (R 4.0.2)   
## memoise 2.0.1 2021-11-26 [1] CRAN (R 4.0.2)   
## ottrpal 1.0.1 2023-03-28 [1] Github (jhudsl/ottrpal@151e412)   
## pillar 1.9.0 2023-03-22 [1] CRAN (R 4.0.2)   
## pkgbuild 1.1.0 2020-07-13 [1] RSPM (R 4.0.2)   
## pkgconfig 2.0.3 2019-09-22 [1] RSPM (R 4.0.3)   
## pkgload 1.1.0 2020-05-29 [1] RSPM (R 4.0.3)   
## png 0.1-8 2022-11-29 [1] CRAN (R 4.0.2)   
## 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 1.4.0 2020-10-07 [1] RSPM (R 4.0.2)   
## R6 2.4.1 2019-11-12 [1] RSPM (R 4.0.0)   
## Rcpp 1.0.10 2023-01-22 [1] CRAN (R 4.0.2)   
## readr 1.4.0 2020-10-05 [1] RSPM (R 4.0.2)   
## remotes 2.2.0 2020-07-21 [1] RSPM (R 4.0.3)   
## reticulate 1.28 2023-01-27 [1] CRAN (R 4.0.2)   
## rlang 1.1.0 2023-03-14 [1] CRAN (R 4.0.2)   
## rmarkdown 2.10 2023-03-28 [1] Github (rstudio/rmarkdown@02d3c25)  
## rprojroot 2.0.3 2022-04-02 [1] CRAN (R 4.0.2)   
## rstudioapi 0.11 2020-02-07 [1] RSPM (R 4.0.0)   
## 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 2023-03-28 [1] Github (R-lib/testthat@e99155a)   
## tibble 3.2.1 2023-03-20 [1] CRAN (R 4.0.2)   
## usethis 1.6.3 2020-09-17 [1] RSPM (R 4.0.2)   
## utf8 1.1.4 2018-05-24 [1] RSPM (R 4.0.3)   
## vctrs 0.6.1 2023-03-22 [1] CRAN (R 4.0.2)   
## withr 2.3.0 2020-09-22 [1] RSPM (R 4.0.2)   
## xfun 0.26 2023-03-28 [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

# 5 References