Population Genetics Modeling

Table of Contents

# 1 Homepage

This is the course homepage and digital textbook for Population Genetics Modeling (AS.020.369).

#### 1.0.0.1 Instructor

Andrew Bortvin, abortvi2[at]jhu.edu

### 1.0.1 Course Updates

* 8/21: Introductory R chapters added under R Guidebook
* 8/21: Syllabus added to website
* 8/21: Course website is live!

# 2 Course Syllabus

**COMPUTATIONAL APPROACHES TO POPULATION GENETICS**  
Instructor: Andrew Bortvin  
Times: Tuesday, Thursday 3:00PM - 4:30PM  
Location: UG Teaching Lab (UTL) 189  
Office Hours: Wednesday 9:00-10:00 AM and 5:00-6:00 PM, or by appointment; Levi 251  
Course website: <https://andrew-bortvin.github.io/popGenModeling/>

### 2.0.1 Course Description

The size, composition, and genetics of populations fluctuate over time. These fluctuations are the product of dynamics between individuals, the interactions between populations, and the context of a population within a broader ecological landscape. The quantitative tools developed to study population genetics allow biologists to discover the simple fundamental principles that govern these complex systems. This course will introduce the basic theory of population genetics while teaching students the fundamental skills of programming in the R programming language, which will allow them to directly implement and visualize theoretical concepts. Students will model and simulate theoretical populations and analyze population-scale genomic data. This course will examine evolution on a variety of scales, ranging from the competition between cells within a single organism, to population dynamics in conservation biology that span decades, to the evolution of contemporary human populations over hundreds of thousands of years.

### 2.0.2 Learning Goals

By the end of this course, students will be able to:

• outline, using biological theory and terms, how populations grow and interact with each other  
• describe the external factors that can cause changes in genetic diversity and use this information to predict how specific demographic scenarios would impact a population  
• manipulate, analyze, and visualize data using the R programming language  
• describe and interpret common formats used to store genomic data, and implement standard analytic protocols used to analyze these data types

### 2.0.3 Grading

30% Participation  
40% Weekly Assignments  
30% Final Project

**Weekly Assignments**

Class sessions will consist of a lecture in which theory is introduced, instructor-led live-coding sessions that implement the models studied in class, and time for students to work independently on assignments that further develop the models designed in class. Most weeks, there will be a take-home assignment that extends concepts studied in class. Students will have a week to complete problem sets. After submission, students will receive feedback from the instructor on each assignment and will have until the end of the semester to submit any revisions necessary.

Each assignment will consist of a set of required exercises that can be completed by students of any coding background. These will be followed by optional, more computationally-focused exercises, which will allow students to examine more intricate evolutionary scenarios and implement more complicated computational models.

Work will be graded on reasonable completion–that is to say, code that demonstrates an understanding of an algorithm and its general implementation will receive full marks, regardless of whether output is exactly correct. Students will also be assessed on the clarity and interpretability of the data visualizations that their code outputs and the accuracy of their responses to short questions prompting biological interpretation of their results.

**Google and AI**

Googling is always an acceptable way to find answers or help, and I encourage you to utilize it extensively. If you adopt a solution following a Google search, make sure you understand what you incorporate, rather than just copy/paste without comprehension of the logic or code. Google is also a good way to learn more about any error messages you encounter in your code.

You may be familiar with ChatGPT and other large language models. After trying each problem/assignment/task on your own, if you’re still running into issues, feel free to use ChatGPT as you would any other online resource (Google, stack overflow, etc.). Learning how to succinctly describe exactly what you want to accomplish is a skillset in itself, so this can be good practice. If you find code that seems to work (e.g., from Google) but you’re not sure how exactly it works, you can also type it into ChatGPT and ask it to explain what’s happening. As always, please do not submit any code if you are not familiar entirely with how it works; flag it and ask an instructor for assistance. Be aware that ChatGPT might confidently offer an answer that is not correct; so always check the output on your own.

### 2.0.4 Schedule

**First Week - Introduction to Population Genetics Modeling**

| Date | Topic | Assignments |
| --- | --- | --- |
| August 27 | Welcome; Course Overview |  |
| August 29 | Introduction to R Programming - Working with Data, Plotting | DUE: Create a Posit account |

**Unit One: Population Biology**  
How do population sizes change? Models for one and two populations. Cooperation, Competition, and predation.

**Week Two: One Population Models**

| Date | Topic | Assignments |
| --- | --- | --- |
| September 3 | The Exponential and Logistic Growth Models |  |
| September 5 | Density-Dependent Growth | Problem Set 1 Assigned |

**Week Three: Multiple Populations**

|  |  |  |
| --- | --- | --- |
| September 10 | Lotka-Volterra dynamics 1: Competition and Cooperation |  |
| September 12 | Lotka-Volterra dynamics 2: Predation and Parasitism | Problem Set 1 Due; Problem Set 2 Assigned |

**Week Four: Advanced Topics in Population Biology**

|  |  |  |
| --- | --- | --- |
| September 17 | Spatial Models |  |
| September 19 | Social Evolution and Game Theory | Problem Set 2 Due; Problem Set 3 Assigned |

**Unit Two: Population Genetics**  
How do we measure the genetic relationships between individuals? Between Populations? Between Species?  
What determines the fate of a genetic element in a population? How does the size and demographic history of a population impact its genetic composition?  
Biological Simulation

**Week Five: The Wright Fisher Model**

|  |  |  |
| --- | --- | --- |
| September 24 | The Wright-Fisher Model: Evolutionary Neutrality |  |
| September 26 | The Wright-Fisher Mode 2: Types of Selection, Selective Sweeps | Problem Set 3 Due; Problem Set 4 Assigned |

**Week Six: Multiple Loci - Measures of Genetic Variation**

|  |  |  |
| --- | --- | --- |
| October 1 | Nucleotide Diversity, the Site Frequency Spectrum |  |
| October 3 | F statistics | Problem Set 4 Due |

**Week Seven: Biological Simulation and Population Size Changes**

|  |  |  |
| --- | --- | --- |
| October 8 | The SLiM Programming Language and slimr | Problem Set 5 Assigned |
| October 10: | Population Bottlenecks, Population Expansion, and Genetic Diversity |  |

**Week Eight: Biological Simulation and Population Size Changes Continued**

|  |  |  |
| --- | --- | --- |
| October 15 | Population Size Changes and the Site Frequency Spectrum | Problem Set 5 Due |
| October 17 | Fall Break |  |

**Week Nine: Multiple Populations and Genetic Relatedness**

|  |  |  |
| --- | --- | --- |
| October 22 | Simulation with Multiple Populations - Migration | Problem Set 6 Assigned |
| October 24 | Simulation with Multiple Populations - Admixture, Local Adaptation |  |

**Unit Three: Analyzing Genetic Data**  
How are population-scale genetic variants represented?  
How do we quantify relatedness between populations?  
Tests for selection, association testing, fine mapping. Phylogeny

**Week Ten: Association Testing**

|  |  |  |
| --- | --- | --- |
| October 29 | The Variant Call Format and population-scale data | Problem Set 6 Due |
| October 31 | GWAS, linkage disequilbrium, Fine Mapping | Problem Set 7 Assigned |

**Week Eleven: Population Structure and Phylogeny**

|  |  |  |
| --- | --- | --- |
| November 5 | Population structure: PCA, STRUCTURE, and clustering methods |  |
| November 7 | The Coalescent - Inferring Timing of Selection | Problem Set 7 Due |

**Week Twelve: Constructing and Interpreting Phylogenies**

|  |  |  |
| --- | --- | --- |
| November 12 | Working with Phylogenetic Trees - the ape and phytools packages | Problem Set 8 Assigned |
| November 14 | Tree Comparison Methods, Advanced trees | DUE: Finalized proposals for independent/small Group projects |

**Week Thirteen: Independent/Small Group Projects**

|  |  |  |
| --- | --- | --- |
| November 19 | Work on Independent/Small Group Projects | Problem Set 8 Due |
| November 21 | Work on Independent/Small Group Projects |  |

**Week Fourteen:**

|  |  |  |
| --- | --- | --- |
| December 3 | Evolutionary Methods in Other Fields (Linguistics, Economics, etc.) Current Directions in Population Genetics |  |
| December 5 | Project Presentations and Discussion | Independent/Small Group Projects Due |

**December 19**: All revisions for weekly assignments due

# 3 Lecture Notes

## 3.1 Class 2: Intro to R

8/27/2024

Today I want to give a pretty broad overview of the basics of using R. There are three big topics that I want to hit upon today. The first is just the basics of the language - what are the standard ways we interface with R? What data can we work with? So on.

Secondly, I want to introduce you to working with tabular data. Data tables are the heart and soul of R, and so today we’ll learn how to make them, read them in from a file, and manipulate them.

And lastly, I want to teach you a bit of the basics of plotting in R.

Every line of code that we write in an R script evaluates, unless we preface it with the hashtag #. In that case, everything in the line of code following the # is a comment and is ignored by the interpreter.

We can run our script by clicking, or with the nice hotkey Shift+Command+Return (Windows: Shift+Control+Enter). We can run our individual lines with Command+Return (Windows: Control+Enter).

### 3.1.1 Data Types, Variables, Math

#### 3.1.1.1 Characters and Variables

In R, we have access to a lot of different data types. Text data is called **Character data** and is denoted by single or double quotes:

"ant"

## [1] "ant"

We’ve just written down that string and not actually done anything with it. Typically, though, we want to save and store our data for future use.

We can do this using **variables**. We assign a variable using the operators <- or =. For our purposes these are interchangeable for assignment; if you look at code online you’ll likely see both. I use <- since it is a little more common.

insect <- "cicada"

Now we have a thing in memory with the name insect. And associated with that name is some data - in this case, the word “ant”. Incidentally, you can see in the top right corner of your screen in the Environment tab there is now an insect variable with the associated word “ant”.

And now, whenever we want to access that data, we can use its variable name. For example, the print() function allows us to, fittingly, print data to the screen. We can print the contents of insect as such:

print(insect)

## [1] "cicada"

And so just one bit of nomenclature - here, print() is the **function**, and **insect** - the thing we give to the function - is the function’s **argument**.

A little more about assignment: we can “overwrite” a variable by assigning new data to it.

insect <- "ant"  
print(insect)

## [1] "ant"

Now, insect is associated with the value "ant". The old value is removed and the new one is added in its place.

I’ve just told you that insect is character data, but if you ever are working with data and you don’t know the type of data it is, you can investigate that using the class() function:

class(insect)

## [1] "character"

##### 3.1.1.1.1 paste

We have two very helpful functions for combining strings paste and paste0.

paste(insect, "hill")

## [1] "ant hill"

We can give paste() a separator to combine with a common delimiter:

paste("eye", "of", "newt", sep = ",")

## [1] "eye,of,newt"

paste0() is a shorthand of writing sep = ""

print(paste(insect, "hill", sep = ""))

## [1] "anthill"

print(paste0(insect, "hill"))

## [1] "anthill"

#### 3.1.1.2 Numbers

In many programming languages, you need to be very careful about different types of data - the difference between integers and numbers with decimal points is often extremely critical. R simplifies this a lot - under the hood, there are differences between these data types, but for our purposes we can broadly treat them as one type of data - **numeric data**.

Math is straightforward:

aNumber <- 3.14  
print(aNumber + 2)

## [1] 5.14

Notice that this **does not** modify the original value of aNumber:

print(aNumber)

## [1] 3.14

To do that, we must use the assignment operator:

aNumber <- aNumber \* 2  
print(aNumber)

## [1] 6.28

Other Math:

print(5 + 2.2) # Notice the accommodation of integers and floating point data

## [1] 7.2

print(7/3) # Division

## [1] 2.333333

print(9\*4) # Multiplication

## [1] 36

print(2\*\*3) # Exponents

## [1] 8

This last point about math is fairly intuitive, so I don’t need to dwell on it too long, but R obeys standard order of operations. Just like in real math, you can use parenthesis to control the order of operations.

print(1 + 3 \* 2)

## [1] 7

print((1 + 3) \* 2)

## [1] 8

#### 3.1.1.3 Logical Data

R can represent whether conditions are true or false using the appropriately named, case-sensitive TRUE and FALSE. These can shortened to T and F.

A really nifty part of these is that they can also be automatically converted to the numeric 1 for TRUE and 0 for FALSE.

T + TRUE + FALSE

## [1] 2

Now that we’ve seen the three major data types, one last thing about the assignment operator. Assignment can change the type of a variable. For example:

testData <- 5  
print(class(testData))

## [1] "numeric"

testData <- "yes"  
print(class(testData))

## [1] "character"

### 3.1.2 Comparing Values

Logical data most commonly appears when we compare values. We compare equality using the double equals sign:

print("egg" == "egg")

## [1] TRUE

print("Egg" == "egg")

## [1] FALSE

We can invert this using the != operator:

print(2 != 3)

## [1] TRUE

We can likewise use > and < to check if things are greater or less than each other.

print(5 > 2.2)

## [1] TRUE

>= and <= are greater than/less or equal to:

print(3 <= 3)

## [1] TRUE

We can also use the exclamation point ! to negate logic in R:

print(!TRUE)

## [1] FALSE

print(!(3 > 2))

## [1] FALSE

#### 3.1.2.1 Multiple Conditions

AND and OR operators are used to evaluate multiple conditions. Any number of conditions connected by AND and OR operators return only one TRUE or FALSE value.

##### 3.1.2.1.1 AND

The logical AND is denoted by the operator &. Two conditions connected by & evaluate as TRUE only if **both** are TRUE:

(5 + 2 ==7) & ('a' != 'b') # True and True

## [1] TRUE

If a single condition is FALSE, the entire statement is FALSE:

(3 + 2 < 4) & (5 > 2) # False and True

## [1] FALSE

##### 3.1.2.1.2 OR

The logical OR is denoted by the operator |. Two conditions connected by | evaluate as TRUE if **either** condition is TRUE:

(7 >= 3) | (2 == 3) # True and False

## [1] TRUE

### 3.1.3 Collections

Most of the time, our data doesn’t come in the form of individual discrete observations. Instead, we usually work with collections of data - for example, the size of a population across many years.

The most basic kind of collection in R is the **vector**, which combines multiple data points of the same type. We create a vector using the function c() (stands for concatenate).

fruits <- c("peach", "plum", "pear")  
print(fruits)

## [1] "peach" "plum" "pear"

If we mix data of multiple types, R automatically converts data, to the best of its ability:

fruits <- c("peach", 5.2, "pear")  
print(fruits)

## [1] "peach" "5.2" "pear"

Notice that the numeric 5.2 is converted to the character "5.2" (see the quotation marks).

#### 3.1.3.1 Adding to vectors

We can use the c() function to add to an existing vector:

fruits <- c(fruits, "apple")  
print(fruits)

## [1] "peach" "5.2" "pear" "apple"

#### 3.1.3.2 Indexing

We can extract from a vector using the brackets []. Inside the brackets, put the position of the item you want to extract:

print(fruits)

## [1] "peach" "5.2" "pear" "apple"

print(fruits[2]) # Get the second item

## [1] "5.2"

print(fruits[length(fruits)]) # Get the last item

## [1] "apple"

We can provide a range or a vector of positions to extract multiple items from our vector:

print(fruits[2:4])

## [1] "5.2" "pear" "apple"

print(fruits[c(1,3,2)])

## [1] "peach" "pear" "5.2"

We can combine indexing and the assignment operator to modify a vector. For example, changing "5.2" back to "plum":

fruits[2] <- "plum"  
print(fruits)

## [1] "peach" "plum" "pear" "apple"

#### 3.1.3.3 Comparisons with vectors

We can use our comparison operators with vectors to compare the entire vector to a value.

myNumbers <- c(4, 10, 8, 2)  
myNumbers > 5

## [1] FALSE TRUE TRUE FALSE

We return a vector of logical values!

#### 3.1.3.4 Two other major types of collections:

##### 3.1.3.4.1 Rep

We often want to repeat elements a set number of times. Writing this out by hand is annoying, so there’s a helpful function to do it for you: rep().

rep(5, times = 4)

## [1] 5 5 5 5

This repeats 5 four times.

We can also repeat vectors:

rep(c('stop', 'go'), times = 4)

## [1] "stop" "go" "stop" "go" "stop" "go" "stop" "go"

What if we want to repeat each element in the vector a set number of times before moving on to the next? e.g. we want to turn c("a", "b") into c("a","a","a", "b","b","b")?

rep(c('a', 'b'), each = 3)

## [1] "a" "a" "a" "b" "b" "b"

So, notice that this the first function we’ve encountered that takes multiple arguments. Multiple arguments are extremely common. For most functions, the first argument will be the input data or a formula or something. It doesn’t need a name - R can figure that out on its own.

After that, though, most arguments are **optional**. So if you give R the code rep(c('stop', 'go'), 4), it won’t automatically know what that 4 means - it could be either times or each in that position. So it’s a good practice to explicitly tell R what your optional arguments are by writing their name.

Incidentally, how do we know what these optional arguments can be? This is something we need to know on a function-by function basis, and the ones for the common functions will be second nature to you sooner than you think. But also, we can look up our functions documentation using ?rep. And if our function is not from the default R language, we can do a broader search using ??rep?

##### 3.1.3.4.2 seq

We often want to work with a series of numbers in order (1,2,3,4,5,…,n). The seq() function allows us to do this.

Let’s look at it in a bit more detail:

seq(5)

## [1] 1 2 3 4 5

We can give seq() multiple numbers to create a start and stop condition:

seq(1, 10)

## [1] 1 2 3 4 5 6 7 8 9 10

seq(7,3)

## [1] 7 6 5 4 3

Two important optional arguments for seq()

seq(5,20, by = 5)

## [1] 5 10 15 20

seq(4,9, length = 12)

## [1] 4.000000 4.454545 4.909091 5.363636 5.818182 6.272727 6.727273 7.181818  
## [9] 7.636364 8.090909 8.545455 9.000000

### 3.1.4 Iteration

An extremely common thing for us is that we will want to perform actions over and over again. Say you have 30 individuals in a population and every generation it gets 10% bigger. We can write that out over and over, but it gets tiring and impractical soon.

for (i in seq(1, 5)){  
 print(i)  
}

## [1] 1  
## [1] 2  
## [1] 3  
## [1] 4  
## [1] 5

So what we do here is take this sequence of 5 items. For each item, we take this temporary variable i, and we perform an action on it. Here, we just print it out.

for (i in seq(1, 5)){  
 i <- 2\*\*i  
 print(i > 6)  
}

## [1] FALSE  
## [1] FALSE  
## [1] TRUE  
## [1] TRUE  
## [1] TRUE

We don’t need to operate on i directly. For example, to implement that population example:

N <- 30  
for (i in seq(1, 8)){  
 N <- N \* 1.1  
 print(N)  
}

## [1] 33  
## [1] 36.3  
## [1] 39.93  
## [1] 43.923  
## [1] 48.3153  
## [1] 53.14683  
## [1] 58.46151  
## [1] 64.30766

So, here we take this variable N and we make it 10% bigger 8 times using our loop.

The last thing I want to touch upon here is that we put a for loop inside a for loop:

for (i in c("b", "c")){  
 for (j in c("at", "ar")){  
 print(paste0(i, j))  
 }  
}

## [1] "bat"  
## [1] "bar"  
## [1] "cat"  
## [1] "car"

### 3.1.5 Data Frames: Tabular Data

So vectors are great, but usually we don’t keep track of just *one* thing - we normally have multiple variables that we track simultaneously - for example, time and population size. A data frame is a tabular data structure that lets us organize a lot of different data in one place.

Each column in a data frame is a vector - so within each column, the data needs to be of the same type. However, each column can be its own data type. So to make this concrete, let’s build a simple data frame with some made up data:

myDf <- data.frame(  
 numbers = c(1, 4, 6, 2, 3.3),  
 logic = c(T, T, F, F, T),  
 words = c("red", "green", "blue", "orange", "turquoise")  
)  
  
myDf

## numbers logic words  
## 1 1.0 TRUE red  
## 2 4.0 TRUE green  
## 3 6.0 FALSE blue  
## 4 2.0 FALSE orange  
## 5 3.3 TRUE turquoise

So now that we’ve made a simple dataset, let’s load in some more elaborate real data.

penguins <- read.csv("penguins.csv")

We have a few ways of quickly looking at our data:

head(penguins)

## species island bill\_length\_mm bill\_depth\_mm flipper\_length\_mm body\_mass\_g  
## 1 Adelie Torgersen 39.1 18.7 181 3750  
## 2 Adelie Torgersen 39.5 17.4 186 3800  
## 3 Adelie Torgersen 40.3 18.0 195 3250  
## 4 Adelie Torgersen NA NA NA NA  
## 5 Adelie Torgersen 36.7 19.3 193 3450  
## 6 Adelie Torgersen 39.3 20.6 190 3650  
## sex  
## 1 MALE  
## 2 FEMALE  
## 3 FEMALE  
## 4   
## 5 FEMALE  
## 6 MALE

tail(penguins)

## species island bill\_length\_mm bill\_depth\_mm flipper\_length\_mm body\_mass\_g  
## 339 Gentoo Biscoe 47.2 13.7 214 4925  
## 340 Gentoo Biscoe NA NA NA NA  
## 341 Gentoo Biscoe 46.8 14.3 215 4850  
## 342 Gentoo Biscoe 50.4 15.7 222 5750  
## 343 Gentoo Biscoe 45.2 14.8 212 5200  
## 344 Gentoo Biscoe 49.9 16.1 213 5400  
## sex  
## 339 FEMALE  
## 340   
## 341 FEMALE  
## 342 MALE  
## 343 FEMALE  
## 344 MALE

and View(penguins), which is analogous to clicking on it in the Environment tab.

So we can only see the first seven columns of our data here - if we ever want to know what all the columns are, we can use the names() function:

names(penguins)

## [1] "species" "island" "bill\_length\_mm"   
## [4] "bill\_depth\_mm" "flipper\_length\_mm" "body\_mass\_g"   
## [7] "sex"

We can also pretty easily get the dimensions of our data:

nrow(penguins)

## [1] 344

ncol(penguins)

## [1] 7

dim(penguins)

## [1] 344 7

#### 3.1.5.1 Subsetting

We can extract columns from our data frame using the dollar sign. So for example if I want to see the island column, I can do:

penguins$island

Which we can now index the usual way:

penguins$island[5:10]

## [1] "Torgersen" "Torgersen" "Torgersen" "Torgersen" "Torgersen" "Torgersen"

We can also index the entire df the conventional way, but now we need to provide a row and column index:

penguins[5, 10]

## NULL

#### 3.1.5.2 Logical Indexing

Let’s say that we want the penguins with big bills.

We can use logical comparisons with the bill\_length\_mm vector to find which ones are 40 mm or longer.

penguins$bill\_length\_mm >= 40

This creates a huge vector of TRUEs and FALSEs. We can give this as a row index to pull out the TRUE rows:

penguins[penguins$bill\_length\_mm >= 40, ]

Notice that the column index is blank - this means that we want to pull out all columns.

Notice that we retain rows where the bill length is NA are retained. The easiest way to address this is to remove rows with NAs. Checking for NA values in R is a bit finicky. The conventional way is to use the function is.na(). This returns TRUE for all NAs. We want the opposite of that, so we add in ! to negate it:

penguins[penguins$bill\_length\_mm >= 40 & !is.na(penguins$bill\_length\_mm), ]

We can also use this type of logical comparisons to answer the question of *how many* penguins have long bills. penguins$bill\_length\_mm >= 40 returns a logical vector of T and F. Recall that these are equivalent to 1 and 0. So to find the number of TRUEs, we can just sum the vector:

sum(penguins$bill\_length\_mm >= 40, na.rm = TRUE)

## [1] 242

Again, we need to remove the NAs for this to work as intended, so the sum function has an optional na.rm argument.

### 3.1.6 Plotting

For plotting, we use the external library ggplot2. This library is installed but because it is not a part of the base R programming language, we need to let R know that we will be using it. This is done with the following line of code:

library(ggplot2)

To create a plot, we need to provide a data set that we will plot, along with defining some aesthetics (axes, colors, groups of data, etc.). Almost always, we will need to define the x and y axes. Let’s start by plotting the bill length and body mass of our penguins.

ggplot(penguins, aes(x = body\_mass\_g, y = bill\_length\_mm))



This already creates a nice plot. Notice that the axes are already scaled around your data.

Now we can add points. To add a geometric, we use the addition operator + to add a type of geometric object to our plots:

ggplot(penguins, aes(x = body\_mass\_g, y = bill\_length\_mm)) +  
 geom\_point()

## Warning: Removed 2 rows containing missing values or values outside the scale range  
## (`geom\_point()`).



Now, we can also color our data by a different variable. This involves adding a color term to aes For example, recoloring by sex:

ggplot(penguins, aes(x = body\_mass\_g,   
 y = bill\_length\_mm,  
 color = sex)) +  
 geom\_point()

## Warning: Removed 2 rows containing missing values or values outside the scale range  
## (`geom\_point()`).



You may notice that some of our penguins are missing sex information. This is coded by an empty string "" rather than an NA. We can filter these out as such:

penguinsFiltered <- penguins[penguins$sex != "", ]

We retain just the rows where the sex column is not equal to ". Plotting this new dataset:

ggplot(penguinsFiltered, aes(x = body\_mass\_g,   
 y = bill\_length\_mm,  
 color = sex)) +  
 geom\_point()



Another way to separate variables is to create multiple subplots. This is mostly easy done by adding facet\_wrap() to our ggplot() statement. To facet\_wrap(), we give the name of the column that we want to split by. For example, splitting the above plot by island: `

ggplot(penguinsFiltered, aes(x = body\_mass\_g,   
 y = bill\_length\_mm,  
 color = sex)) +  
 geom\_point() +   
 facet\_wrap("island")



Notice that for facet\_wrap(), we need to give the column name in quotation marks.

### 3.1.7 Wide to Long

One thing you may have noticed is that when we plot our data, we want our data to be tall rather than wide. To make that a little more concrete, here’s a data set of population sizes for two species:

popSizes <- data.frame(  
 generation = seq(1,5),   
 species1 = c(10, 12, 15, 13, 14),  
 species2 = c(20, 15, 14, 13, 11)  
)  
  
head(popSizes)

## generation species1 species2  
## 1 1 10 20  
## 2 2 12 15  
## 3 3 15 14  
## 4 4 13 13  
## 5 5 14 11

So if we go to plot this, we hit an issue:

ggplot(popSizes, aes(x = generation, y = #????))

There are a few workarounds for this, but one of the best ones is to reshape your data and make it taller.

The reshape2 library has a great function for this is:

library(reshape2)

popSizesTall <- melt(popSizes, id.vars = "generation")  
head(popSizesTall)

We should now rename our columns to be a bit more descriptive. This can be done with the colnames() function:

colnames(popSizesTall) <- c("generation", "species", "size")



Plotting is now straightforward:

ggplot(popSizesTall, aes(x = generation, y = size, color = species)) +  
 geom\_point() +  
 geom\_line() +   
 theme\_classic()



library(ggplot2)  
library(reshape2)

## 3.2 Class 3: Exponential Growth and Age Structure

Today, we’re looking at the exponential growth model, which describes how a population grows in the absence of density dependence - that is, when the growth rate doesn’t change in response to population size. First, we’ll plot exponential growth for one population at a given per capita growth rate.

# Initialize parameters   
N <- 50 # Population Size  
r <- 0.04 # Per capita growth rate  
lambda <- 1 + r  
n\_generations <- 50  
  
# Vector to keep track of population size  
popSize <- c(N)  
  
# Run simulation  
for (i in seq(n\_generations)){  
 N <- N \* lambda # Calculate next generation  
 popSize <- c(popSize, N)  
}  
  
# Organize into data frame  
df <- data.frame(  
 generation = seq(0, n\_generations),  
 popSize = popSize  
)  
  
# Plot   
ggplot(df, aes(x = generation,   
 y = popSize)) +  
 geom\_point() +  
 xlab("Generation") +   
 ylab("Population Size") +  
 ggtitle("Exponential Growth") +  
 theme\_bw()



Next, plotting across multiple growth rates:

# Initialize parameters  
n\_generations <- 50  
  
# Vectors to track variable  
generations <- c()  
popSizeList <- c()  
growthRate <- c() # Tracking growth rate so that we can separate our data by rate when plotting  
  
# Run simulation  
for (growth\_rate in c(-0.1, 0, 0.02, 0.04)){ # Loop through all the growth rates (i.e. run simulation once for each rate)  
 lambda <- 1 + growth\_rate # Get lambda for the current growth rate  
 N <- 50 # Reset starting population size to 50  
 popSize <- c(N) # Track population size for current run of the simulation  
   
 # Run simulation for 50 generations  
 for (i in seq(n\_generations)){  
 N <- N \* lambda  
 popSize <- c(popSize, N)  
 }  
   
 # Update vectors with simulation output  
 generations <- c(generations, seq(0, n\_generations))  
 popSizeList <- c(popSizeList, popSize)  
 growthRate <- c(growthRate, rep(lambda, n\_generations + 1))  
}  
  
# Reorganize into a data frame   
df <- data.frame(  
 generations = generations,  
 popSize = popSizeList,  
 growthRate = growthRate  
)  
  
# Plot  
ggplot(df, aes(x = generations,   
 y = popSize,   
 color = factor(growthRate))) +  
 geom\_point() +  
 xlab("Generation") +   
 ylab("Population Size") +  
 ggtitle("Exponential Growth") +  
 theme\_bw() +   
 labs(color = "Growth Rate") # Change label of the color legend



Notice that our population decreases when lambda < 1, stays constant when lambda = 1, and increases when lambda > 1.

### 3.2.1 Comparison to Real Data

I’ve pulled some data from Yoav et al. 2019, PNAS (<https://doi.org/10.1073/pnas.1902217116>). This data set tracks microbial growth across a series of plates. Reading in the data:

yoavData <- read.csv("yoavData.csv", row.names = "X")  
head(yoavData)



The data has three columns: Time.S - the time in seconds, Plate - the name of the plate that the culture is grown in, and Size - the size of the colony. First, plotting the size of the colony for the first 13,000 seconds:

ggplot(yoavData, aes(x = Time.S, y = Size, color = Plate)) +  
 geom\_point() +  
 geom\_line() +  
 theme(legend.position = "none") +  
 xlab("Time (Seconds)") +  
 ylab("Population Size") +  
 xlim(c(0, 13000)) +  
 ylim(c(0, 300000))



Visually, this looks exponential. We can fit a curve to it. To do this, we can use the function lm() which fits a linear model. The tilde (~) symbol indicates the relationship we want to measure - in other words, this equation measures log(Size) as a function of Time. To make this an exponential fit, we take the log of the dependent variable:

fitData <- yoavData[yoavData$Time.S < 15000, ]  
lm(log(fitData$Size) ~ fitData$Time.S)

We can plot this along the real-world data.

# Creating a dataframe from our function  
xs <- seq(1, 60000) # Set the x values we want to evaluate  
ys <- exp(10.067264) \* exp(0.000182 \* xs) # Get the corresponding values using coefficients from the fit  
fit\_line <-data.frame(Time.S = xs, # Convert to data frame  
 Size = ys,  
 Plate = "fit")  
  
# Notice that the names we give above  
# Match the names of the axes of our plotting data.  
  
# We also give create a column "Plate" for color-coding  
  
# Plot data  
ggplot(yoavData, aes(x = Time.S, y = Size, color = Plate)) +  
 geom\_point() +  
 geom\_line() +  
 xlab("Time (Seconds)") +  
 ylab("Population Size") +  
 geom\_line(data = fit\_line, size = 1, color = "black") + # Plot the line of best fit   
 theme(legend.position = "none") +  
 xlim(c(0, 13000)) +  
 ylim(c(0, 300000))



Notice that when we add the line of best fit, we provide the argument data = fit\_line. This tells ggplot that we are using a second data set not provided in the initial ggplot call.

Now, let’s plot our data extending further in time - this is done by changing the xlim arguments:

xs <- seq(1, 60000)  
ys <- exp(10.067264) \* exp(0.000182 \* xs)  
fit\_line <-data.frame(Time.S = xs,  
 Size = ys,  
 Plate = "fit")  
  
  
ggplot(yoavData, aes(x = Time.S, y = Size, color = Plate)) +  
 geom\_point() +  
 geom\_line() +  
 xlab("Time (Seconds)") +  
 ylab("Population Size") +  
 geom\_line(data = fit\_line, size = 1, color = "black") +  
 theme(legend.position = "none") +  
 xlim(c(0, 35000)) + # Changed x limits to see further into the Time   
 ylim(c(0, 800000))



Notice that as the population grows, the exponential model is an increasingly poor fit - the exponential model predicts continued growth at an increasing rate; the actual data shows growth slowing down until the population hits a plateau. We’ll look at this discrepancy in more detail next class!

### 3.2.2 Age Structure

Next, let’s look at age structured populations - that is a population where we track the ages of individuals and where individuals of different ages can have different reproduction rates and mortalities.

We’ll start with a simplified population of 0 and 1-year olds. In each generation, the number of 1 year olds in the next generation is determined purely by the number of 0 years olds that survive the year, and the number of 0 year olds is determined by the sum of the number 0 year olds and the number of 1 year olds that reproduce.

# Initializing parameters  
# Initial numbers of 0 and 1 year olds  
N0 <- 2   
N1 <- 10  
  
# Birth Rates for 0 and 1 year olds  
f0 <- 0.4  
f1 <- 1.8  
  
# Death Rates   
# Just for 0 year olds - all 1 year olds die  
S0 <- 0.7   
  
n\_generations <- 50  
  
# Empty vectors to store data   
n0\_list <- c()  
n1\_list <- c()  
  
# Run simulation  
for (i in seq(n\_generations)){  
 # Calculate the number of 0 and 1 year olds in the next generation   
 N0\_next\_gneration <- N0 \* f0 + N1 \*f1  
 N1\_next\_generation <- N0 \* S0  
   
 # Update values   
 N0 <- N0\_next\_gneration  
 N1 <- N1\_next\_generation  
   
 # Add to lists  
 n0\_list <- c(n0\_list, N0)  
 n1\_list <- c(n1\_list, N1)  
   
}  
  
# Organize into data frame  
df <- data.frame(  
 n0 = n0\_list,  
 n1 = n1\_list,  
 generation = seq(n\_generations)  
)  
  
# Convert to tall format  
df <- melt(df, id.vars = "generation")  
colnames(df) <- c("Generation", "Age", "PopulationSize")  
  
# Plot  
ggplot(df, aes(x = Generation, y = PopulationSize, color = Age)) +  
 geom\_point() +   
 geom\_line() +   
 theme\_classic() +  
 ylab("Population Size")



This looks like normal exponential growth, with slightly different rates for 0 and 1 year olds. However, let’s zoom in on early time points:

ggplot(df, aes(x = Generation, y = PopulationSize, color = Age)) +  
 geom\_point() +   
 geom\_line() +   
 theme\_classic() +  
 ylab("Population Size") +  
 xlim(c(0, 10)) +   
 ylim(c(0,400))



The last thing we want to do with this data is plot the ratio of 0 to 1 year olds:

# Initializing parameters  
# Initial numbers of 0 and 1 year olds  
N0 <- 2   
N1 <- 10  
  
# Birth Rates for 0 and 1 year olds  
f0 <- 0.4  
f1 <- 1.8  
  
# Death Rates   
# Just for 0 year olds - all 1 year olds die  
S0 <- 0.7   
  
n\_generations <- 50  
  
# Empty vectors to store data   
ratio\_list <- c(N0/N1)  
  
# Run simulation  
for (i in seq(n\_generations)){  
 # Calculate the number of 0 and 1 year olds in the next generation   
 N0\_next\_gneration <- N0 \* f0 + N1 \*f1  
 N1\_next\_generation <- N0 \* S0  
   
 # Update values   
 N0 <- N0\_next\_gneration  
 N1 <- N1\_next\_generation  
   
 # Add to list  
 ratio\_list <- c(ratio\_list, N0/N1) # Calculate the ratio of 0/1 year olds  
}  
  
# Organize into a data frame  
ratioDf <- data.frame(  
 generations = seq(0, n\_generations),  
 ratio = ratio\_list  
)  
  
# Plot  
ggplot(ratioDf, aes(x = generations,   
 y = ratio)) +   
 geom\_point() +  
 geom\_line() +  
 theme\_bw() +   
 xlab("Generation") +   
 ylab("Ratio 0:1 Year Olds")



Notice that after an initial period of chaos, the ratio of age groups stabilizes. This is called the stable age distribution and is a general property of age structured models like this.

### 3.2.3 The Leslie Matrix

The Leslie matrix allows us to very efficiently extend age structured models to an arbitrarily large number of populations.

In the Leslie matrix L, the first row is the fecundity of each category, and the area beneath that has a diagonal arrangement of survival rates for each age category.

Note that for matrix multiplication, we use %\*%.

# Leslie matrix  
L <- matrix(data = c(0, 0, 0.1, 1.1, 1.8, 0.9, 0.4, 0.1,  
 0.4, 0, 0, 0, 0, 0, 0, 0,  
 0, 0.9, 0, 0, 0, 0, 0, 0,  
 0, 0, 0.8, 0, 0, 0, 0, 0,  
 0, 0, 0, 0.9, 0, 0, 0, 0,  
 0, 0, 0, 0, 0.8, 0, 0, 0,  
 0, 0, 0, 0, 0, 0.2, 0, 0,  
 0, 0, 0, 0, 0, 0, 0.1, 0), nrow = 8, ncol = 8, byrow = TRUE)  
  
# Starting population size for each age class  
pop\_matrix <- as.matrix(c(1, 3, 2, 0, 6, 10, 25, 15))  
  
# Starting population size for each age class  
total\_size <- c(sum(pop\_matrix))  
  
n\_generations <- 40  
  
# Run simulation   
for (i in seq(n\_generations)){  
 pop\_matrix <- L %\*% pop\_matrix  
 total\_size <- c(total\_size, sum(pop\_matrix))  
}  
  
# Organize data into a data frame  
df <- data.frame(  
 year = seq(0, n\_generations),  
 size = total\_size  
)  
  
# Plot  
ggplot(df, aes(x=year, y = size)) +  
 geom\_point() +   
 geom\_line() +   
 theme\_bw() +  
 xlab("Time (Year)") +  
 ylab("Population Size")



library(ggplot2)  
library(reshape2)

## 3.3 Class 5: Lotka-Volterra I: Competition

To begin with, a simple implementation of the Lotka-Volterra equation in R, with discrete time points:

# Initialize conditions  
# Population Sizes  
N1 <- 30  
N2 <- 20  
  
# Carrying Capacities  
K1 <- 1000  
K2 <- 400  
  
# Interaction parameters between pops  
alpha <- 0.1  
beta <- 0.9  
  
# Growth rates  
r1 <- 1.5  
r2 <- 1.3  
  
# Simulation time  
nYears <- 20  
  
# Vectors to store population sizes  
n1\_list <- c(N1)  
n2\_list <- c(N2)  
  
# Run simulation  
for (i in seq(nYears)){  
 # Calculate future pop sizes for next time point  
 # Saving N1 as a new variable so that it the current value of N1  
 # can be used to calculate N2   
 N1\_future <- N1 + r1 \* N1 \* ((K1 - N1 - alpha \* N2)/K1)   
 N2\_future <- N2 + r2 \* N2 \* ((K2 - N2 - beta \* N1)/K2)   
   
 # Update N1 and N2   
 N1 <- N1\_future  
 N2 <- N2\_future  
   
 # Save to list   
 n1\_list <- c(n1\_list, N1)  
 n2\_list <- c(n2\_list, N2)  
}  
  
# Convert to data frame  
df <- data.frame(  
 time = seq(length(n1\_list)),  
 n1 = n1\_list,  
 n2 = n2\_list  
)  
  
# Convert to tall format  
df <- melt(df, id.vars = "time")  
colnames(df) <- c("time", "pop", "size")  
   
# Plot  
ggplot(df, aes(x = time,  
 y = size,   
 color = pop)) +  
 geom\_line() +   
 geom\_point() +   
 labs(x = "Time (Generations)",   
 y = "Population Size",  
 color = "Population") +   
 theme\_bw()



A couple things are apparent from this mode:

* Under these conditions, both populations are not able to coexist - after a brief period of growth, population 2 collapses to 0.
* In this model, population 2 dips below 0 at generation 6.

The first behavior is fine - it’s a pretty basic property of this model that many parameter combinations lead to one population outcompeting the other to extinction. This is called **competitive exclusion**.

The second behavior - dipping below zero - is not desirable. While our models are theoretical representations of reality, unrealistic behaviors like this are good to correct.

Here, shooting below zero happens because with a large step size and a large growth rate, our model moves too quickly. In response to this, we can decrease our step size. In fact, if we decrease our step size enough, our model very closely approximates the continuous form of the equation.

# Initialize conditions  
# Population Sizes  
N1 <- 30  
N2 <- 20  
  
# Carrying Capacities  
K1 <- 1000  
K2 <- 400  
  
# Interaction parameters between pops  
alpha <- 0.1  
beta <- 0.9  
  
# Growth rates  
r1 <- 1.5  
r2 <- 1.3  
  
# Simulation time  
nYears <- 20  
  
# Vectors to store population sizes  
n1\_list <- c(N1)  
n2\_list <- c(N2)  
  
# Create a step size  
stepSize <- 0.01  
  
# Run simulation  
for (i in seq(nYears/stepSize)){ # Scale our for loop range by the step size  
 # Calculate future pop sizes for next time point  
 # Saving N1 as a new variable so that it the current value of N1  
 # can be used to calculate N2   
 N1\_future <- N1 + r1 \* N1 \* ((K1 - N1 - alpha \* N2)/K1) \* stepSize # Multiply by step size  
 N2\_future <- N2 + r2 \* N2 \* ((K2 - N2 - beta \* N1)/K2) \* stepSize # Multiply by step size  
   
 # Update N1 and N2   
 N1 <- N1\_future  
 N2 <- N2\_future  
   
 # Save to list   
 n1\_list <- c(n1\_list, N1)  
 n2\_list <- c(n2\_list, N2)  
}  
  
# Convert to data frame  
df <- data.frame(  
 time = seq(length(n1\_list)),  
 n1 = n1\_list,  
 n2 = n2\_list  
)  
  
# Convert to tall format  
df <- melt(df, id.vars = "time")  
colnames(df) <- c("time", "pop", "size")  
   
# Plot  
ggplot(df, aes(x = time \* stepSize, # We multiply time by stepSize to convert back to years  
 y = size,   
 color = pop)) +  
 geom\_line() +   
 labs(x = "Time (Generations)",   
 y = "Population Size",  
 color = "Population") +   
 theme\_bw()



A few changes are needed to make this work:

* The for loop is changed to incorporate step size: for (i in seq(nYears/stepSize)). If, previously, each step represented one generation, each step now represents one generation \* stepSize - just a fraction of the time. In other words, we need to run our model more times to represent an equal amount of real time.
* When calculating population sizes, we incorporate step size: N1\_future <- N1 + r1 \* N1 \* ((K1 - N1 - alpha \* N2)/K1) \* stepSize
* Lastly, when plotting, we convert time back to years by multiplying by stepSize again: ggplot(df, aes(x = time \* stepSize,

So far, we have seen models where one population outcompetes the other. It is worth noting that this is not an inherent property of Lotka-Volterra models:

# Initialize conditions  
# Population Sizes  
N1 <- 30  
N2 <- 20  
  
# Carrying Capacities  
K1 <- 1000  
K2 <- 1000  
  
# Interaction parameters between pops  
alpha <- 0.4  
beta <- 0.9  
  
# Growth rates  
r1 <- 1.5  
r2 <- 1.3  
  
# Simulation time  
nYears <- 20  
  
# Vectors to store population sizes  
n1\_list <- c(N1)  
n2\_list <- c(N2)  
  
# Create a step size  
stepSize <- 0.01  
  
# Run simulation  
for (i in seq(nYears/stepSize)){ # Scale our for loop range by the step size  
 # Calculate future pop sizes for next time point  
 # Saving N1 as a new variable so that it the current value of N1  
 # can be used to calculate N2   
 N1\_future <- N1 + r1 \* N1 \* ((K1 - N1 - alpha \* N2)/K1) \* stepSize # Multiply by step size  
 N2\_future <- N2 + r2 \* N2 \* ((K2 - N2 - beta \* N1)/K2) \* stepSize # Multiply by step size  
   
 # Update N1 and N2  
 N1 <- N1\_future  
 N2 <- N2\_future  
   
 # Save to list   
 n1\_list <- c(n1\_list, N1)  
 n2\_list <- c(n2\_list, N2)  
}  
  
# Convert to data frame  
df <- data.frame(  
 time = seq(length(n1\_list)),  
 n1 = n1\_list,  
 n2 = n2\_list  
)  
  
# Convert to tall format  
df <- melt(df, id.vars = "time")  
colnames(df) <- c("time", "pop", "size")  
   
# Plot  
ggplot(df, aes(x = time / stepSize, # We multiply time by stepSize to convert back to years  
 y = size,   
 color = pop)) +  
 geom\_line() +   
 labs(x = "Time (Generations)",   
 y = "Population Size",  
 color = "Population") +   
 theme\_bw()



Note that when both populations coexist, they do not hit their carrying capacities:

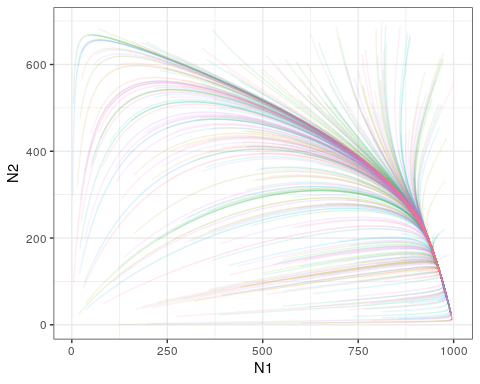
# Initialize conditions  
# Population Sizes  
N1 <- 30  
N2 <- 20  
  
# Carrying Capacities  
K1 <- 1000  
K2 <- 900  
  
# Interaction parameters between pops  
alpha <- 0.4  
beta <- 0.9  
  
# Growth rates  
r1 <- 1.5  
r2 <- 1.3  
  
# Simulation time  
nYears <- 20  
  
# Vectors to store population sizes  
n1\_list <- c(N1)  
n2\_list <- c(N2)  
  
# Initialize variables to store population size for the logistic model  
# We start at the same point as the LV model  
N1\_logistic <- N1  
N2\_logistic <- N2   
  
# Lists to store   
n1\_list\_logistic <- c(N1\_logistic)  
n2\_list\_logistic <- c(N2\_logistic)  
  
# StepSize  
stepSize <- 0.01  
  
for (i in seq(nYears/stepSize)){  
 # Calculate LV dynamics  
 N1\_future <- N1 + r1 \* N1 \* ((K1 - N1 - alpha \* N2)/K1) \* stepSize  
 N2\_future <- N2 + r2 \* N2 \* ((K2 - N2 - beta \* N1)/K2) \* stepSize  
   
 # Update N1 and N2  
 N1 <- N1\_future  
 N2 <- N2\_future  
   
 # Save to list   
 n1\_list <- c(n1\_list, N1)  
 n2\_list <- c(n2\_list, N2)  
   
 # Calculate logistic dynamics  
 N1\_logistic <- N1\_logistic + r1 \* (1 - N1\_logistic/K1) \* N1 \* stepSize  
 N2\_logistic <- N2\_logistic + r2 \* (1 - N2\_logistic/K2) \* N2 \* stepSize  
   
 # Save to list  
 n1\_list\_logistic <- c(n1\_list\_logistic, N1\_logistic)  
 n2\_list\_logistic <- c(n2\_list\_logistic, N2\_logistic)  
}  
  
# Store LV data as a data frame  
df <- data.frame(  
 time = seq(length(n1\_list)),  
 n1 = n1\_list,  
 n2 = n2\_list  
)  
  
# Reorganize to a tall dataframe  
df <- melt(df, id.vars = "time")  
colnames(df) <- c("time", "pop", "size")  
# Add a column with the model name  
df["Model"] = "Lotka-Volterra"   
  
# Store Logistic data as a data frame  
df\_logistic <- data.frame(  
 time = seq(length(n1\_list\_logistic)),  
 n1 = n1\_list\_logistic,  
 n2 = n2\_list\_logistic  
)  
  
# Reorganize to a tall dataframe  
df\_logistic <- melt(df\_logistic, id.vars = "time")  
colnames(df\_logistic) <- c("time", "pop", "size")  
# Add a column with the model name  
df\_logistic["Model"] = "Logistic"  
  
# Combine LV and logistic dataframes into one for plotting  
df\_plot <- rbind(df, df\_logistic)  
  
# Reorganize the factor levels of our plotting data  
# This determines which population is plotted in dotted lines  
df\_plot$Model <- factor(df\_plot$Model,   
 levels = c("Lotka-Volterra", "Logistic"))  
   
ggplot(df\_plot, aes(x = time / stepSize,  
 y = size,  
 color = pop,   
 linetype = Model)) +  
 geom\_line() +   
 labs(x = "Time (Generations)",   
 y = "Population Size",  
 color = "Population") +   
 theme\_bw()



### 3.3.1 Equilibirum from multiple starting points

Now, starting from multiple conditions:

# Initialize conditions  
# Carrying Capacities  
K1 <- 1000  
K2 <- 700  
  
# Interaction parameters between pops  
alpha <- 0.3  
beta <- 0.6  
  
# Growth Rates  
r1 <- 1.5  
r2 <- 1.3  
  
# Length of population  
nYears <- 20  
  
  
# Lists to store population sizes, trial numbers, and time within each trial  
n1\_all\_trials <- c()  
n2\_all\_trials <- c()  
trialNumber <- c()  
time <- c()  
  
# Step Size  
step <- 0.01  
  
# Number of trials we are running  
nTrials <- 300  
  
# Run simulation  
for (trial in seq(nTrials)){  
 # Randomly generate starting population sizes  
 N1 <- sample(seq(K1), 1)  
 N2 <- sample(seq(K2), 1)  
   
 # Create a list for population sizes  
 n1\_list <- c(N1)  
 n2\_list <- c(N2)  
   
 # Run the simulation  
 for (i in seq(nYears/step)){  
 # Calculate population size  
 N1\_future <- N1 + r1 \* N1 \* ((K1 - N1 - alpha \* N2)/K1) \* step  
 N2\_future <- N2 + r2 \* N2 \* ((K2 - N2 - beta \* N1)/K2) \* step  
   
 N1 <- N1\_future  
 N2 <- N2\_future  
   
 # Add to lists  
 n1\_list <- c(n1\_list, N1)  
 n2\_list <- c(n2\_list, N2)  
 }  
   
 # Add simulation results to overall results  
 n1\_all\_trials <- c(n1\_all\_trials, n1\_list)  
 n2\_all\_trials <- c(n2\_all\_trials, n2\_list)  
 trialNumber <- c(trialNumber, rep(trial, time = length(n1\_list)))  
 time <- c(time, seq(length(n1\_list)))  
}  
  
# Store as data frame  
df <- data.frame(  
 n1 = n1\_all\_trials,   
 n2 = n2\_all\_trials,  
 trial = trialNumber,  
 time = time  
)  
  
# Plot  
ggplot(df, aes(x = n1,   
 y = n2,   
 color = factor(trial), # Factor lets R know that trial is a category, not a continuous number  
 alpha = time)) + # Transparency is controlled by time within trial - dark points are the last times within a trial  
 geom\_path() +   
 guides(color="none",   
 alpha = "none") + # Remove color and transparency scales  
 xlab("N1") +   
 ylab("N2") +  
 theme\_bw()



library(ggplot2)  
library(reshape2)

## 3.4 Class 7: Infectious Diseases

Before getting into any biology, let’s learn about making our own functions. Throughout the course, we’ve made use of R functions such as print(), c(), and ggplot(); defining our own functions allows us to

Defining our own functions allows us to perform actions over and over on multiple sets of inputs.

Below, let’s practice creating a function that takes one input, a list, and returns the number of things in it:

# A vector containing the contents of my fridge   
refrigerator <- c("Apples", "Pears", "Bread", "Eggs", "Cheddar",  
 "Brie", "Lettuce", "Shallots", "Cabbage")  
  
# The function definition   
countFood <- function(fridge){ # Fridge is a local variable - it does not exist outside the function   
 numItems <- length(fridge)  
 print(paste("I have", numItems, "things in my fridge."))  
}  
  
# Two more fridges   
myFriendsPantry <- c("olives")  
fridge <- c("veal", "steak", "halloumi")  
  
# Run the fridge function on all three sets of inputs:   
countFood(refrigerator)

## [1] "I have 9 things in my fridge."

countFood(myFriendsPantry)

## [1] "I have 1 things in my fridge."

countFood(fridge)

## [1] "I have 3 things in my fridge."

The countFood() function prints out a value, but doesn’t actually store anything to memory. Very often, we want to use the output from our function for downstream analysis. To save the oputput of our function, we need to include a return() statement.

For example, the function below generates a poly A tail sequence by pasting together a series of "A"’s, taking as input the length of the sequence to return:

# Define function  
polyAtail <- function(length){  
 polyA <- rep("A", times = length) # Create a vector of A's  
  
 return(paste(polyA, collapse = "")) # Convert that vector into a string  
}  
  
# Run polyAtail() function on two inputs  
polyA <- polyAtail(63)  
polyA2 <- polyAtail(17)  
  
# Print results  
print(polyA)

## [1] "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"

print(polyA2)

## [1] "AAAAAAAAAAAAAAAAA"

### 3.4.1 The SIR Model

In the function below, we implement the simplest form of the SIR model. The SIR model tracks the numbers of **S**usceptible, **I**nfected, and **R**ecovered individuals in a population, assuming a constant birth/death rate. :

SIR <- function(S, I, R, # Starting number of individuals, as a fraction of total population size  
 mu, B, gamma, # death/birth rate, contact lengths, recovery rate  
 simTime, step){  
   
 # Initialize total population size  
 N <- 1  
   
 # lists to store population sizes  
 sList <- c(S)  
 iList <- c(I)  
 rList <- c(R)  
   
 # loop through times  
 for (time in seq(simTime/step)){  
 # Calculate rates of change   
   
 dS <- (mu \* (N - S) - B \* I \* S / N) \* step  
 dI <- (B \* I \* S / N - (mu + gamma) \* I) \* step  
 dR <- (gamma \* I - mu \* R) \* step  
   
 # Update population sizes based on rate of change  
 S <- S + dS  
 I <- I + dI   
 R <- R + dR  
   
 # Update lists  
 sList <- c(sList, S)  
 iList <- c(iList, I)  
 rList <- c(rList, R)  
 }  
 # Store results to data frame  
 df <- data.frame(  
 time = seq(length(sList)) \* step,  
 S = sList,   
 I = iList,  
 R = rList  
 )  
   
 return(df)  
}

The function above returns a data frame containing time and population size for individuals of all categories.

We would also like to plot our data. Since we may want to do this over and over again for different inputs, it also makes sense to plot through a function. Notice that because we just want to create a plot here, we don’t need to save anything to memory. Therefore, this function does not have a return() statement:

# Define plotting function   
plotSIR <- function(SIR\_data){  
 # Convert to tall format  
 SIR\_data\_tall <- melt(SIR\_data, id.vars = "time")  
 # Fix the column names  
 colnames(SIR\_data\_tall) <- c("Time", "Population", "Size")  
   
 # Plot, coloring by population   
 ggplot(SIR\_data\_tall, aes(x = Time,  
 y = Size,   
 color = Population)) +   
 geom\_line() +   
 theme\_bw()  
}

We can also make a function to just plot the infected individuals. This is identical to the function above, except that we subset our individuals while plotting (SIR\_data\_tall[SIR\_data\_tall$Population == "I", ]):

# Define plotting function   
plotInfected<- function(SIR\_data){  
 # Convert to tall format  
 SIR\_data\_tall <- melt(SIR\_data, id.vars = "time")  
 # Fix the column names  
 colnames(SIR\_data\_tall) <- c("Time", "Population", "Size")  
   
 # Plot, coloring by population   
 ggplot(SIR\_data\_tall[SIR\_data\_tall$Population == "I", ], aes(x = Time,  
 y = Size,   
 color = Population)) +   
 geom\_line() +   
 theme\_bw()   
}

Now that we’ve defined our functions, we can run them on a set of data:

# Initial fraction of populations in each category  
S <- 0.19  
I <- 0.01  
R <- 0.8  
  
mu <- 1/(50 \* 52) # Birth rate   
B <- 2 # Contact lengths   
gamma <- 0.5 # Recovery rate   
  
simTime <- 52 \* 100  
step <- 0.1  
  
SIR\_data <- SIR(S, I, R, mu, B, gamma,simTime, step)  
plotSIR(SIR\_data)

plotInfected(SIR\_data)

As with other multi-population models, we can the trajectories of multiple populations as they relate to each other:

ggplot(data=SIR\_data, aes(x = S, y = I)) +  
 geom\_path() +theme\_classic()

Note that in this model, there is a cyclical oscillation approaching an equilibrium as time goes on. The new spikes in case number are a result of new births in the population increasing the susceptible population.

### 3.4.2 SEIR

The SEIR model extends the standard SIR model by including a new phase - **E**xposed individuals. Exposed individuals have contracted the infection but are not yet symptomatic.

We’ve also added a couple new terms to make this model more realistic:

1. The parameter p now specifies the at birth vaccination rate. As a result, some fractions of individuals in each generation immediately enter the **R** category rather than **S** at birth.
2. The parameter alpha specifies the rate of death induced by the disease. This quantity of individuals is removed from the **I** population. Because alpha can in fact lead to population size change, we now need to recalculate N with each time step, reflecting the changing total population size.

SEIR <- function(S, E, I, R, # Starting number of individuals  
 mu, B, gamma, sigma, alpha, p, # death/birth rate, contact lengths, recovery rate, disease progression rate, death rate from disease, vaccination proportion  
 simTime, step){  
   
 # Initialize the total population size (start at 100%)  
 N <- 1  
   
 # lists to store population sizes  
 sList <- c(S)  
 eList <- c(E)  
 iList <- c(I)  
 rList <- c(R)  
   
 # loop through times  
 for (time in seq(simTime/step)){  
 # Calculate rates of change   
 dS <- (mu \* (N \* (1-p) - S) - B \* I \* S / N) \* step # (1-p) reflects vaccination  
 dE <- (B \* I \* S / N - (mu + sigma) \* E) \* step  
 dI <- (sigma \* E - (mu + gamma + alpha) \* I) \* step   
 dR <- (gamma \* I - mu \* R + mu \* N \* p) \* step # mu \* N \* p reflects vaccination  
   
 # Update population sizes based on rates of change  
 S <- S + dS  
 E <- E + dE   
 I <- I + dI   
 R <- R + dR  
   
 # Update lists  
 sList <- c(sList, S)  
 eList <- c(eList, E)  
 iList <- c(iList, I)  
 rList <- c(rList, R)  
   
 # Recalculate total population size  
 N <- sum(S, E, I, R)  
 }  
 # Store results to data frame  
 df <- data.frame(  
 time = seq(length(sList)) \* step,  
 S = sList,   
 E = eList,   
 I = iList,  
 R = rList  
 )  
   
 return(df)  
}

Now, running on a population:

# Initializing proportion of individuals in each population   
S <- 0.19  
E <- 0.0  
I <- 0.01  
R <- 0.8  
  
mu <- 1/(50 \* 52) # Birth rate  
B <- 2 # Contact lengths   
sigma <- 0.2 # Time spent exposed prior to symptoms   
gamma <- 0.5 # Recovery rate   
alpha <- 0.1 # Death rate from sickness   
p <- 0.2 # At Birth Vaccination Rate  
  
simTime <- 52 \* 50  
step <- 0.1  
  
# Run Simulation  
SEIR\_data <- SEIR(S, E, I, R, mu, B, gamma, sigma, alpha, p, simTime, step)  
# Plot   
plotSIR(SEIR\_data)

Now, plotting just the infected population:

plotInfected(SEIR\_data)

And lastly, overlaying the infected and exposed populations on the same figure:

ggplot(SEIR\_data, aes(x = S, y = I)) +  
 geom\_path() +   
 geom\_path(data = SEIR\_data, aes(x = S, y = E), color = "plum") +   
 theme\_classic()

### 3.4.3 SEIHFR

You aren’t responsible for being able to reproduce the code below, but I showed this model in class and am including the code below.

The SEIHFR model was developed to model Ebola outbreaks. In addition to the categories of individuals included in the SEIR model, it also considers:

# 4 R Guidebook

This section of the website will contain a concise texbook form introduction to the R programming language, tailored to topics covered in class.

## 4.1 Data Types

R allows us to work with many different types of data. The crucial ones for us are:

**Numeric** data are all numbers. For example:

• 78  
• 3.2  
• -9.8

**Characters** are any text enclosed by single or double quotation marks. For example:

• My Grandpa's deck has no pathetic cards  
• "85.3"

Note that 85.3 - no quotes - is numeric data; "85.3" is character data.

**Logical** data has three possible values: TRUE, FALSE, and NA for missing data. These are case sensitive and lack quotes ("TRUE" would be character data). TRUE and FALSE can also be shortened to T and F.

**Factors** are characters with associated **levels**, i.e. an order. For example, imagine that you have the set of characters "Bach", "Purcell", "Handel" and you always want them sorted in *that specific order*, rather than the default alphabetical order. We can use factors to manually create such an order for our data. If this is a bit abstract right now, it’ll make more sense in practice once we work with real data!

## 4.2 Variables

Variables allow us to store data in memory. When we store data in memory, we can also give it a name.

We create a variable using the assignment operator <- or =:

variable\_name <- value

For our purposes, these are interchangeable. For example:

• composer <- "buxtehude"  
• year = 1637

The variable name can be almost anything. Here are some general rules to consider when naming a variable:

• Variable names must start with a letter and consist of letters, numbers, ., and \_  
• Some words are considered “reserved” - i.e. they are already used by R to mean something and therefore cannot be a variable name. These include TRUE, if, NULL, etc. For a full list, check [here](https://www.datamentor.io/r-programming/reserved-words).

### 4.2.1 Overwriting Variables

In the following code example, we assign two different values to the same variable name:

myData <- 3  
myData <- "marigold"

What is the value of myData? It will be "marigold". In the first line of code, we set myData equal to 3, but then in the next line, we overwrite that value and set myData equal to "margiold". The previous value of myData is erased and it is set to the new value.

Note that in doing so, we change not just the value of myData, but also the *data type* - it goes from numeric to character data.

We can also do this to update the value of a variable:

a\_number <- 5   
a\_number <- a\_number + 10  
print(a\_number)

## [1] 15

Here, we set the variable a\_number equal to 5. In the next line, we add 10 to its value, storing the result under the same name. a\_number is now equal to 15.

## 4.3 Math

A lot of mathematical operations in R are straightforward. Here are some of the basic operations we can perform:

• + and -: addition and subtraction  
• \* and /: multiplication and division  
• \*\*: exponents

We can perform mathematical operations on values directly:

print(2 + 3)

## [1] 5

Or we can operate on variables:

x <- 5   
print(x\*\*3)

## [1] 125

We can also save the output of an expression as a variable:

my\_product <- 2 \* 10  
print(my\_product)

## [1] 20

### 4.3.1 Order of Operations

R follows the usual mathematical order of operations. And like in math, we can use parentheses () to enforce a specific order.

print(2 \* (2 + 2))

## [1] 8

## 4.4 Comparisons

In R, comparisons will always return Logical data, i.e. either TRUE or False. We can use the following syntax to compare values:

• < and >: greater than, less than  
• <= and >=: greater than or equal to, less than or equal to

For example:

print(5.3 < 17)

## [1] TRUE

print(4 >= (8/2))

## [1] TRUE

• == checks if two values are equal. != checks if two values are different.

print(2 == (10 - 8))

## [1] TRUE

Note that a single equals sign = can be used to assign values. However, a double equals sign == is used to compare values.

### 4.4.1 AND and OR

We can evaluate multiple conditions using the logical AND and OR operators.

#### 4.4.1.1 AND

AND statements are represented using the operator &. Two AND statements evaluate as TRUE only if **both** are TRUE. For example:

(5 > 2) & (7 + 2 == 9)

## [1] TRUE

If **either** statement is FALSE, the entire expression is FALSE:

(5 > 2) & (12 == 2)

## [1] FALSE

#### 4.4.1.2 OR

OR statements are represented using the operator |. Two OR statements evaluate as TRUE if **either** statement is TRUE. For example:

(5 > 2) | (7 + 2 == 9)

## [1] TRUE

If **either** statement is TRUE, the entire expression is TRUE:

(5 > 2) | (12 == 2)

## [1] TRUE

## 4.5 Vectors

Usually, we don’t want to work with just a single data point - we will typically have multiple values that we want to store together. The most convenient way of doing this in R is using **vectors**. A vector stores multiple data points, preserving their order.

We create a vector using the c() (short for “concatenate”) function. For example:

plants <- c("Feverfew", "Ivy", "Willow")  
print(plants)

## [1] "Feverfew" "Ivy" "Willow"

The data within a vector may be of any type, but all elements of a vector **must be of the same data type**. What happens if we try to create a vector with multiple data types?

beetles <- c("Weevil", "Firefly", 5)  
print(beetles)

## [1] "Weevil" "Firefly" "5"

Here, we mix character and numeric data. This isn’t allowed, so the numeric 5 is converted to the string "5".

### 4.5.1 Indexing

We will often want to take a larger vector and extract specific data points from it. To do this, we **index** our vector using the general syntax:

vectorName[itemPosition]

The position of the first item in the list is 1 (in other words, R is 1-indexed). Let’s try indexing using our plants vector, made above.

print(plants)

## [1] "Feverfew" "Ivy" "Willow"

To extract "Ivy", we would do:

plants[2]

## [1] "Ivy"

We can also use a colon to extract multiple subsequent elements:

plants[1:2]

## [1] "Feverfew" "Ivy"

We can also provide a vector to index multiple values:

plants[c(1,3)]

## [1] "Feverfew" "Willow"

We often want to extract elements near the end of a vector. plants is short and we can count to the end of it easily, but most of the data we will work with is a lot longer. One easy way to index items near the end of a vector is to use the length() function. We can index the final entry in plants as so:

plants[length(plants)]

## [1] "Willow"

length(plants) is 3, so writing plants[length(plants)] is equivalent to writing plants[3]

Likewise, we can index the second element by doing some math:

plants[length(plants) - 1]

## [1] "Ivy"

### 4.5.2 Logical Indexing

We often want to subset our data not by the position of elements, but based on whether or not they meet a certain criterion. Below, I have generated a short list of numbers:

myNumbers <- c(1, 54, 12.2, 70, 18, 24, 94)

Let’s say we want to extract just the values that are greater than 15 from this list. We can use any of our comparative operators with a vector to compare *all* values within the vector:

myNumbers > 15

## [1] FALSE TRUE FALSE TRUE TRUE TRUE TRUE

This generates a logical vector. We can provide this vector as an index to myNumbers to pull out the elements that are TRUE.

myNumbers[myNumbers > 15]

## [1] 54 70 18 24 94

We can use the logical AND (&) and OR (|) operators to combine conditions. For example, extracting values greater than 15 and less than 30:

myNumbers[(myNumbers > 15) & (myNumbers < 30)]

## [1] 18 24

### 4.5.3 Modifying Vectors

Once we point to elements within a vector, we can modify them using the assignment operator. For example, making the second item in myNumbers equal to 200:

myNumbers[2] <- 200  
print(myNumbers)

## [1] 1.0 200.0 12.2 70.0 18.0 24.0 94.0

We can also modify multiple elements at once. For example, making every value less than 50 equal to 0:

myNumbers[myNumbers < 50] <- 0  
print(myNumbers)

## [1] 0 200 0 70 0 0 94

### 4.5.4 Adding to Vectors

We can add to vectors using the concatenate function:

plants <- c(plants, "Philodendron")  
print(plants)

## [1] "Feverfew" "Ivy" "Willow" "Philodendron"

## 4.6 Data Frames

The Data Frame is a tabular data structure which can contain data of multiple types. It is conceptually similar to an Excel spreadsheet and is by far the most important data structure in R programming.

In a dataframe, each column is a vector. This is to say, all elements within a column will be of the same data type. However, different columns can be of different data types.

Here’s a data frame with a handful of writers, their birth years, and whether or not they were poets.

writers <- data.frame(  
 Name = c("Plath", "Tolstoy", "Milton", "Woolf", "Farid ud-Din Attar"),  
 BirthYear = c(1932, 1828, 1608, 1882, 1145),  
 Poet = c(TRUE, FALSE, TRUE, FALSE, TRUE)  
)  
  
print(writers)

## Name BirthYear Poet  
## 1 Plath 1932 TRUE  
## 2 Tolstoy 1828 FALSE  
## 3 Milton 1608 TRUE  
## 4 Woolf 1882 FALSE  
## 5 Farid ud-Din Attar 1145 TRUE

Note that when we set the names of our columns, we *must* use the equals sign - we cannot use the assignment <- as that is only used for variable assignment (i.e. we cannot do Poet <- c(TRUE, FALSE, TRUE, FALSE, TRUE))

### 4.6.1 Subsetting Data Frames

When we indexed vectors, we used the bracket notation vector[index] to extract information. We can do the same for data frames, but now we must provide two values - one for the row index and one for the column index, so the syntax is dataFrame[row, column]. For example, to pull out the value 1608 from writers, we would do:

writers[3,2]

## [1] 1608

As with vectors, we can extract multiple elements at once:

writers[c(2,3), c(1, 2)]

## Name BirthYear  
## 2 Tolstoy 1828  
## 3 Milton 1608

What if we want to subset the rows, but keep all the columns of our data frame? We can leave a field blank to not subset it at all. For example, pulling out all columns for rows 2 and 3:

writers[c(2,3), ]

## Name BirthYear Poet  
## 2 Tolstoy 1828 FALSE  
## 3 Milton 1608 TRUE

However, there is an easier way of extracting infromation from a data frame - we can take advantage of row names. We can pull out individual vectors from a data frame using the syntax dataFrame$columnName. For example, we can extract the Name vector from writers using:

writers$Name

## [1] "Plath" "Tolstoy" "Milton"   
## [4] "Woolf" "Farid ud-Din Attar"

And then we can index the Name as we would any other vector:

writers$Name[2]

## [1] "Tolstoy"

### 4.6.2 Logical Indexing

As with vectors, we can use logic and comparison operators to subset data frames. For example, we can subset our data frame just to writers who are poets:

poetsVector <- writers$Poet == TRUE # Get the positions of writers who are poets  
writers[poetsVector, ] # Subset our data

## Name BirthYear Poet  
## 1 Plath 1932 TRUE  
## 3 Milton 1608 TRUE  
## 5 Farid ud-Din Attar 1145 TRUE

Notice how here, we save the logical vector as its own variable (poetsVector). We’re doing the equivalent of writers[writers$Poet == TRUE], but you may find dividing this process into multiple lines easier, especially as logic gets more complex.

## 4.7 Plotting

The base R programming language comes with some pretty robust tools for plotting. However, most R users instead use the external library ggplot2, which simplifies plotting and allows for more elaborate data visualization.

Your Posit cloud accounts have ggplot2 pre-installed. However, at the start of each script where you plot, you must let R know that you will use an external package with the line:

library(ggplot2)

For a simple dataset to plot, I’ve saved the temperature in Fahrenheit in Baltimore, MD and Wommels, Netherlands for the next seven hours:

temperatures = data.frame(  
 hours = seq(1,7),  
 temperature = c(74, 73, 72, 71, 75, 79, 87, 59, 60, 62, 65, 69, 72, 67),  
 city = c(rep("Baltimore", 7), rep("Wommels", 7))  
)  
  
head(temperatures)

## hours temperature city  
## 1 1 74 Baltimore  
## 2 2 73 Baltimore  
## 3 3 72 Baltimore  
## 4 4 71 Baltimore  
## 5 5 75 Baltimore  
## 6 6 79 Baltimore

We can initialize a blank canvas with a call to the ggplot() function. The first argument that this function takes is the dataset you wish to plot, which in our case is temperatures.

ggplot(temperatures)



Next, we have to define the axes, groups of data, and other aesthetics. This is done with the aes argument. For our data, it makes sense to have time be the x axis and temperature to be the y axis:

ggplot(temperatures, aes(x = hours, y = temperature))



We now have a blank plot with axes. Notice that the scales of our axes are automatically set by the scales of our data (hours ranges from 0 to 7; temperature ranges approximately from 60 to 90).

Now to add some data to our plot. Because ggplot already knows our input data and how the axes are laid out, all we need to do is to say that type of visualization we want (e.g. points, lines, etc.). Let’s start by adding points to our plot:

ggplot(temperatures, aes(x = hours, y = temperature)) +   
 geom\_point()



This plots our data correctly, but notice that we can’t distinguish between the two cities visually. We can group our data points by coloring them differently. To do this, we can add another argument (color) to our aes statement:

ggplot(temperatures, aes(x = hours, y = temperature, color = city)) +   
 geom\_point()



This is a pretty clear, interpretable plot! We can pretty easily make a few changes to make it look nicer, though. First, let’s also add a line by adding a geom\_line() statement:

ggplot(temperatures, aes(x = hours, y = temperature, color = city)) +   
 geom\_point() +   
 geom\_line()



ggplot will automatically generate axis labels based on the column names of the input data. However, we can also make more descriptive x and y axis labels along with an overall figure title using xlab(), ylab(), and ggtitle(), respectively:

ggplot(temperatures, aes(x = hours, y = temperature, color = city)) +   
 geom\_point() +   
 geom\_line() +  
 xlab("Time (Hours)") +   
 ylab("Temperature (F)") +  
 ggtitle("Projected Temperatures for the Next Six Hours")



Lastly, it is worth mentioning that we can add theme() statements to modify almost any aspect of our figure’s aesthetics. Along with manually defined theme() statements, ggplot comes with a handful of predefined themes:

ggplot(temperatures, aes(x = hours, y = temperature, color = city)) +   
 geom\_point() +   
 geom\_line() +  
 xlab("Time (Hours)") +   
 ylab("Temperature (F)") +  
 ggtitle("Projected Temperatures for the Next Six Hours") +   
 theme\_classic()



ggplot(temperatures, aes(x = hours, y = temperature, color = city)) +   
 geom\_point() +   
 geom\_line() +  
 xlab("Time (Hours)") +   
 ylab("Temperature (F)") +  
 ggtitle("Projected Temperatures for the Next Six Hours") +   
 theme\_bw()



# 5 Lecture Slides

Slides from all class session will be hosted on this page.

[**Lecture 1 (8/27)**](https://docs.google.com/presentation/d/16A7M2S_77kYN3ev2zu5wXeZbEBs8p5NfObUaqHGqkaA/edit?usp=sharing)

[**Lecture 3 (9/3)**](https://docs.google.com/presentation/d/1vd8UfvifcDCSQyhmN6c6R0NcgcNMC7XDK41xzl7khaA/edit?usp=sharing)

[**Lecture 4 (9/5)**](https://docs.google.com/presentation/d/1nu3VC_57NAI0sW3SKwjjjorE0Wvpbg-DW284XiYcXeM/edit?usp=sharing)

[**Lecture 5 (9/10)**](https://docs.google.com/presentation/d/1555l5ge-m2QpIvU_fZM6_7eJnsap8Tgtiqx-aFourpY/edit?usp=sharing)

[**Lecture 6 (9/12)**](https://docs.google.com/presentation/d/1KQEQ2aI9xWmccL2OqRQd1vzrc9CTMArh2GPwHA0yHrM/edit?usp=sharing)

[**Lecture 7 (9/17)**](https://docs.google.com/presentation/d/16ijM_9369inJwd11xA2R_ul4txr0bnEZtwFevL29iy8/edit?usp=sharing)

# Authors

| Credits | Names |
| --- | --- |
| **Pedagogy** |  |
| Instructor | [Andrew Bortvin](https://andrew-bortvin.github.io/) |
| Content Author | [Andrew Bortvin](https://andrew-bortvin.github.io/) |
| **Website** |  |
| Template | [Jeff Leek](https://jtleek.com/) & [The Johns Hopkins Data Science Lab](https://jhudatascience.org/index.html) |
| Design Inspiration | [Stephanie Yan](https://stephaniemyan.github.io/) & [Ali Madooei](https://engineering.jhu.edu/faculty/ali-madooei/) & [JHU Data Structures](https://cs226sp22.github.io/) |
| **Funding** |  |
| JHU Dean’s Teaching Fellowship |  |

## ─ Session info ───────────────────────────────────────────────────────────────  
## setting value  
## version R version 4.3.2 (2023-10-31)  
## os Ubuntu 22.04.4 LTS  
## system x86\_64, linux-gnu  
## ui X11  
## language (EN)  
## collate en\_US.UTF-8  
## ctype en\_US.UTF-8  
## tz Etc/UTC  
## date 2024-09-19  
## pandoc 3.1.1 @ /usr/local/bin/ (via rmarkdown)  
##   
## ─ Packages ───────────────────────────────────────────────────────────────────  
## package \* version date (UTC) lib source  
## askpass 1.2.0 2023-09-03 [1] RSPM (R 4.3.0)  
## bookdown 0.39.1 2024-06-11 [1] Github (rstudio/bookdown@f244cf1)  
## cachem 1.0.8 2023-05-01 [1] RSPM (R 4.3.0)  
## cli 3.6.2 2023-12-11 [1] RSPM (R 4.3.0)  
## colorspace 2.1-0 2023-01-23 [1] RSPM (R 4.3.0)  
## devtools 2.4.5 2022-10-11 [1] RSPM (R 4.3.0)  
## digest 0.6.34 2024-01-11 [1] RSPM (R 4.3.0)  
## dplyr 1.1.4 2023-11-17 [1] RSPM (R 4.3.0)  
## ellipsis 0.3.2 2021-04-29 [1] RSPM (R 4.3.0)  
## evaluate 0.23 2023-11-01 [1] RSPM (R 4.3.0)  
## fansi 1.0.6 2023-12-08 [1] RSPM (R 4.3.0)  
## farver 2.1.1 2022-07-06 [1] RSPM (R 4.3.0)  
## fastmap 1.1.1 2023-02-24 [1] RSPM (R 4.3.0)  
## fs 1.6.3 2023-07-20 [1] RSPM (R 4.3.0)  
## generics 0.1.3 2022-07-05 [1] RSPM (R 4.3.0)  
## ggplot2 \* 3.5.0 2024-02-23 [1] RSPM (R 4.3.0)  
## glue 1.7.0 2024-01-09 [1] RSPM (R 4.3.0)  
## gtable 0.3.4 2023-08-21 [1] RSPM (R 4.3.0)  
## highr 0.10 2022-12-22 [1] RSPM (R 4.3.0)  
## hms 1.1.3 2023-03-21 [1] RSPM (R 4.3.0)  
## htmltools 0.5.7 2023-11-03 [1] RSPM (R 4.3.0)  
## htmlwidgets 1.6.4 2023-12-06 [1] RSPM (R 4.3.0)  
## httpuv 1.6.14 2024-01-26 [1] RSPM (R 4.3.0)  
## httr 1.4.7 2023-08-15 [1] RSPM (R 4.3.0)  
## jsonlite 1.8.8 2023-12-04 [1] RSPM (R 4.3.0)  
## knitr 1.47.3 2024-06-11 [1] Github (yihui/knitr@e1edd34)  
## labeling 0.4.3 2023-08-29 [1] RSPM (R 4.3.0)  
## later 1.3.2 2023-12-06 [1] RSPM (R 4.3.0)  
## lifecycle 1.0.4 2023-11-07 [1] RSPM (R 4.3.0)  
## magrittr 2.0.3 2022-03-30 [1] RSPM (R 4.3.0)  
## memoise 2.0.1 2021-11-26 [1] RSPM (R 4.3.0)  
## mime 0.12 2021-09-28 [1] RSPM (R 4.3.0)  
## miniUI 0.1.1.1 2018-05-18 [1] RSPM (R 4.3.0)  
## munsell 0.5.0 2018-06-12 [1] RSPM (R 4.3.0)  
## openssl 2.1.1 2023-09-25 [1] RSPM (R 4.3.0)  
## ottrpal 1.2.1 2024-06-11 [1] Github (jhudsl/ottrpal@828539f)  
## pillar 1.9.0 2023-03-22 [1] RSPM (R 4.3.0)  
## pkgbuild 1.4.3 2023-12-10 [1] RSPM (R 4.3.0)  
## pkgconfig 2.0.3 2019-09-22 [1] RSPM (R 4.3.0)  
## pkgload 1.3.4 2024-01-16 [1] RSPM (R 4.3.0)  
## profvis 0.3.8 2023-05-02 [1] RSPM (R 4.3.0)  
## promises 1.2.1 2023-08-10 [1] RSPM (R 4.3.0)  
## purrr 1.0.2 2023-08-10 [1] RSPM (R 4.3.0)  
## R6 2.5.1 2021-08-19 [1] RSPM (R 4.3.0)  
## Rcpp 1.0.12 2024-01-09 [1] RSPM (R 4.3.0)  
## readr 2.1.5 2024-01-10 [1] RSPM (R 4.3.0)  
## remotes 2.4.2.1 2023-07-18 [1] RSPM (R 4.3.0)  
## rlang 1.1.4 2024-06-04 [1] CRAN (R 4.3.2)  
## rmarkdown 2.27.1 2024-06-11 [1] Github (rstudio/rmarkdown@e1c93a9)  
## rstudioapi 0.15.0 2023-07-07 [1] RSPM (R 4.3.0)  
## scales 1.3.0 2023-11-28 [1] RSPM (R 4.3.0)  
## sessioninfo 1.2.2 2021-12-06 [1] RSPM (R 4.3.0)  
## shiny 1.8.0 2023-11-17 [1] RSPM (R 4.3.0)  
## stringi 1.8.3 2023-12-11 [1] RSPM (R 4.3.0)  
## stringr 1.5.1 2023-11-14 [1] RSPM (R 4.3.0)  
## tibble 3.2.1 2023-03-20 [1] CRAN (R 4.3.2)  
## tidyselect 1.2.0 2022-10-10 [1] RSPM (R 4.3.0)  
## tzdb 0.4.0 2023-05-12 [1] RSPM (R 4.3.0)  
## urlchecker 1.0.1 2021-11-30 [1] RSPM (R 4.3.0)  
## usethis 2.2.3 2024-02-19 [1] RSPM (R 4.3.0)  
## utf8 1.2.4 2023-10-22 [1] RSPM (R 4.3.0)  
## vctrs 0.6.5 2023-12-01 [1] RSPM (R 4.3.0)  
## withr 3.0.0 2024-01-16 [1] RSPM (R 4.3.0)  
## xfun 0.44.4 2024-06-11 [1] Github (yihui/xfun@9da62cc)  
## xml2 1.3.6 2023-12-04 [1] RSPM (R 4.3.0)  
## xtable 1.8-4 2019-04-21 [1] RSPM (R 4.3.0)  
## yaml 2.3.8 2023-12-11 [1] RSPM (R 4.3.0)  
##   
## [1] /usr/local/lib/R/site-library  
## [2] /usr/local/lib/R/library  
##   
## ──────────────────────────────────────────────────────────────────────────────