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

# 2 Course Syllabus

**COMPUTATIONAL APPROACHES TO POPULATION GENETICS**  
Instructor: Andrew Bortvin  
Times: Tuesday, Thursday 3:00PM - 4:30PM  
Office Hours: To be determined by when2meet at the start of the semester, or by appointment  
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 1: 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 | Problem Set 1 Assigned | |September 5 | Density-Dependent Growth | |

**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

**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

**Week Six: Multiple Loci - Measures of Genetic Variation**  
October 1: Nucleotide Diversity, the Site Frequency Spectrum

October 3: F statistics

**Week Seven: Biological Simulation and Population Size Changes**  
October 8: The SLiM Programming Language and slimr

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

October 17: Fall Break

**Week Nine: Multiple Populations and Genetic Relatedness**  
October 22:Simulation with Multiple Populations - Migration

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

October 31: GWAS, linkage disequilbrium, Fine Mapping

**Week Eleven: Population Structure and Phylogeny**  
November 5: Population structure: PCA, STRUCTURE, and clustering methods

November 7: The Coalescent - Inferring Timing of Selection

**Week Twelve: Constructing and Interpreting Phylogenies**  
November 12: Working with Phylogenetic Trees - the ape and phytools packages

November 14: Tree Comparison Methods, Advanced trees

**Week Thirteen: Independent/Small Group Projects**  
November 19: Work on Independent/Small Group Projects

November 21: Work on Independent/Small Group Projects

**Week Fourteen:**  
December 3: Semester Retrospective Discussion  
Evolutionary Methods in Other Fields (Linguistics, Economics, etc.)  
Current Directions in Population Genetics

December 5: Project Presentations and Discussion

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

# 3 Lecture Notes

## 3.1 Introduction to R

R is a programming language developed primarily for use by statisticians, and it is a common tool in the field for data exploration, analysis, visualization, etc. We will be using it throughout the course to manipulate, view, and interpret data.

In this module, we will learn the basics of R by manipulating some real-world data. In particular, we’ll look at data from the Hudson Bay looking at the size of the hare and lynx populations.

We’ll talk a lot more about this dataset when we study predator-prey models in a few weeks, but for now, let’s make some exploratory plots.

Data adapted from here: <http://people.whitman.edu/~hundledr/courses/M250F03/M250.html>

Please note that input data is available on the course Posit cloud page - please feel free to work alongside this page!

### 3.1.1 Data import

The bread and butter of R is the data frame, a tabular data structure which can contain data of multiple types.

R has a few built-in functions that allow for easy and efficient import of data. As we go on, we’ll learn a lot of nuances about how these work, but for now we’ll start with a tidy, well-behaved dataset that is easy to work with.

We would like for R to import our data and save it into memory. We also want to give it a name, so that we can refer to it later.

In R, we use the assignment operator <- to associate a name with some data. The general syntax for this is:

variableName <- value

Where **left - name right -> value**

**if you know python or c or something, “+” is equivalent**

**lets test it out**

populationData <- read.table("hare\_lynx.csv", sep = ",", header = TRUE)

Let’s break down what’s happening here.

On the left of the <-, we have the text populationData

head(populationData)

## Year Species Count  
## 1 1845 Hare 19.58  
## 2 1846 Hare 19.60  
## 3 1847 Hare 19.61  
## 4 1848 Hare 11.99  
## 5 1849 Hare 28.04  
## 6 1850 Hare 58.00

### 3.1.2 The Data Frame

R has the ability to manipulate multiple different types of data. The primary data types we will use in this course are:

## 3.2 Lecture Notes

In this module, we will learn the basics of R by manipulating some real-world data. In particular, we’ll work with data from the Hudson Bay looking at the size of historical hare and lynx populations.

We’ll talk a lot more about this dataset and the biology it represents when we study predator-prey models, but for now let’s use it make some exploratory plots.

Data adapted from here: <http://people.whitman.edu/~hundledr/courses/M250F03/M250.html>

### 3.2.1 Data import

Our data is available as a comma-delimited text file, or csv. Here’s the top of it as it appears in a text editor:



The first thing we have to do to work with it is to import it into R. R has a few built-in functions that allow for easy and efficient import of data. As we go on, we’ll learn a lot of nuances about how these work, but for now we’ll start with a tidy, well-behaved dataset that is easy to work with. The read.table function will be our workhorse for reading in data.

Let’s try it out. To run this function, we need to provide one **argument**: the location of the data we want to load.

read.table("hare\_lynx.csv")

***ADD IMAGE HERE***

A couple things went wrong here:

1. The data is imported as a single column - we want R to know that there are multiple columns delimited by commas
2. The header (Year,Species,Count) was imported as the first row of our data - we would instead like R to recognize that this is a header rather than a row of data

How do we fix this? Along with the essential argument of the file location, we can provide read.table with many **optional arguments**. Because there are many possibilities for what our second argument may be, we need to explicitly reference the name of our optional arguments.

Let’s inform R that our data is split by commas. We can do this using the sep (short for “separator”) argument.

read.table("hare\_lynx.csv", sep = ",")

***ADD IMAGE***

Now we correctly have three columns of data! We now have to tell R that the first row is a header. There is an optional argument header which can be set to TRUE or FALSE (case sensitive - more on this later).

read.table("hare\_lynx.csv", sep = ",", header = TRUE)

***ADD IMAGE***

Looks good! Now, a couple nuances:

1. The order **named arguments** (here, sep and header doesn’t matter. The following lines of code are equivalent:

read.table("hare\_lynx.csv", sep = ",", header = TRUE)  
read.table("hare\_lynx.csv", header = TRUE, sep = ",")

1. For better legibility, we can add line breaks after commas, like so:

read.table("hare\_lynx.csv",   
 sep = ",",   
 header = TRUE)

This does not impact how data is evaluated.

Now that we read in the data correctly, the last thing to do is to save it as a **variable**. This means that rather than just reading in the data and printing it out, we truly save it in the computer’s memory and assign it a name.

The general syntax for saving a variable in R is:

variableName <- value

For example:

hareLynxData <- read.table("hare\_lynx.csv",   
 sep = ",",   
 header = TRUE)

Now, when we refer to hareLynxData, we are referring to the data we have just imported.

For example we can use the head function, which displays the first 10 rows of our data:

head(hareLynxData)

## Year Species Count  
## 1 1845 Hare 19.58  
## 2 1846 Hare 19.60  
## 3 1847 Hare 19.61  
## 4 1848 Hare 11.99  
## 5 1849 Hare 28.04  
## 6 1850 Hare 58.00

### 3.2.2 DataFrames

### 3.2.3 Data Manipulation

### 3.2.4 Subsetting

### 3.2.5 Logical subsetting

library(ggplot2)

# 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).

**Factors** are characters with associated **levels**, i.e. an order. When we work with numeric data, it is sorted in ascending/descending order. When we work with character data, there is no comparable way to order our data so we can use factors to manually create 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, associated with 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 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  
• \*\*: exponentials

We can perform mathematical operations on values directly:

print(2 + 3)

## [1] 5

Or we can operate on variables:

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

## [1] 125

We can 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. Notice that when we print the vector, 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, and so on (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. For example, 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.

# 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 2024-08-21  
## 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  
##   
## ──────────────────────────────────────────────────────────────────────────────